GEORGIA INSTITUTE OF TECHNOLOGY OFFICE OF CONTRACT ADMINISTRATION

PROJECT ADMINISTRATION DATA SHEET

☒ ORIGINAL □ REVISION NO. ______

Project No. G-36-661 (continuation of G-36-636) DATE 10/29/81

Project Director: R. A. Demillo School/Lab ICS

Sponsor: Office of Naval Research; Arlington, VA

Type Agreement: Contract No. N00014-79-C-0231, Modification No. P00002

Award Period: From 9/1/81 To 1/31/~~83~~ 84 (Performance) 3/31/83 (Reports)

Sponsor Amount: $489,887 (Mod. 2 only) Contracted through:

Cost Sharing: N/A GTRI/~~GIT~~

Title: Software Test and Evaluation Study Phases I and II: Survey and Analysis

ADMINISTRATIVE DATA

OCA Contact Leamon R. Scott

1) Sponsor Technical Contact:

Dr. Robert Grafton
ONR 715
Broadway
New York, N.Y. 10003

2) Sponsor Admin/Contractual Matters:

Office of Naval Research
206 O'Keefe Bldg.
Georgia Tech
Atlanta, GA 30323
Attn: Tom Bryant

Defense Priority Rating: DO-C9

Security Classification: N/A

RESTRICTIONS

See Attached Government Supplemental Information Sheet for Additional Requirements.

Travel: Foreign travel must have prior approval — Contact OCA in each case. Domestic travel requires sponsor approval where total will exceed greater of $500 or 125% of approved proposal budget category.

Equipment: Title vests with Government; except that items costing less than $1K vest w/GIT upon acquisition if prior approval to purchase is obtained from C.O. items costing $1K or more may vest w/GIT as determined by the ACO (ONR).

COMMENTS:

COPIES TO:

Administrative Coordinator
Research Property Management
Accounting
Procurement/EES Supply Services

Research Security Services
Reports Coordinator (OCA) ✓
Legal Services (OCA)
Library

EES Public Relations (2)
Computer Input
Project File
Other ______

GEORGIA INSTITUTE OF TECHNOLOGY   OFFICE OF CONTRACT ADMINISTRATION

R— 5X7782-1

SPONSORED PROJECT TERMINATION/CLOSEOUT SHEET

Date June 12, 1985

Project No. G-36-661   School/~~Lab~~ ICS

Includes Subproject No.(s) A-2568 (Established Under Prior Project G-36-636)

Project Director(s) Dr. R.A. Demillo   GTRC / ~~GIT~~

Sponsor Office of Naval Research

Title Software Test and Evaluation Study Phases I and II: Survey and Analysis

Effective Completion Date: 1/31/84 (Performance) 3/31/84 (Reports)

Grant/Contract Closeout Actions Remaining:

□ None

□ Final Invoice or Final Fiscal Report

☒ Closing Documents

☒ Final Report of Inventions

☒ Govt. Property Inventory & Related Certificate

□ Classified Material Certificate

□ Other

Continues Project No. G-36-636   Continued by Project No.

COPIES TO:

Project Director
Research Administrative Network
Research Property Management
Accounting
Procurement/GTRI Supply Services
Research Security Services
[✓ Reports Coordinator (OCA)]
Legal Services

Library
GTRC
Research Communications (2)
Project File
Other Heyser
Jones

G-36-66

OSD/DDT&E SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II FINAL REPORT

VOLUME 1

REPORT AND RECOMMENDATIONS

BY

R. A. DeMILLO

R. J. MARTIN

SUBMITTED TO
THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION
AND
THE OFFICE OF NAVAL RESEARCH
ONR CONTRACT NUMBER N00014-79-C-0231

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

Volume 1
Final Report and Recommendations

by

R. A. DeMillo
and
R. J. Martin

SUBMITTED BY
GEORGIA INSTITUTE OF TECHNOLOGY

TO

THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION

AND

THE OFFICE OF NAVAL RESEARCH

FOR

ONR CONTRACT NO. N00014-79-C-0231
Subcontract 2G36661

June, 1983

FOREWORD

This volume is one of a set of reports on Software Test and Evaluation prepared by the Georgia Institute of Technology for The Office of the Secretary of Defense/Director Defense Test and Evaluation under Office of Naval Research Contract N00014-79-C-0231.

Comments should be directed to: Director Defense Test and Evaluation (Strategic, Naval, and C^3I Systems), OSD/OUSDRE, The Pentagon, Washington, D.C. 20301.

Volumes in this set include:

Volume 1: Final Report and Recommendations
Volume 2: Software Test and Evaluation: State-of-the-Art Overview
Volume 3: Software Test and Evaluation: Current Defense Practices Overview
Volume 4: Transcript of STEP Workshop, March 1982
Volume 5: Report of Expert Panel on Software Test and Evaluation
Volume 6: Tactical Computer System Applicability Study

PREFACE AND ACKNOWLEDGEMENTS

This series of documents represents the final report and recommendations for the Software Test and Evaluation Project support contract. This contract is funded by the Office of the Secretary of Defense/Director Defense Test and Evaluation and administered by the Office of Naval Research. The prime contractor is the Georgia Institute of Technology. Subcontracts to support this effort have been let to Control Data's Atlanta Research Facility, Clemson University, and a number of independent consultants. The contractors' efforts began in the Fall of 1981 and were completed in the Spring of 1983.

Admiral Isham Linder, Director Defense Test and Evaluation, Mr. Charles Watt, Deputy Director Defense Test and Evaluation, Mr. Donald Greenlee of DDT&E, and Dr. Robert Grafton of ONR have provided support and encouragement. Dr. Edith Martin, Deputy Under Secretary of Defense for Research and Advanced Technology, not only participated in the initial planning for STEP, but has made the resources of her office available.

In addition to these individuals, several organizations have made generous contributions to the progress of STEP. The National Security Industrial Association (NSIA) has been an essential partner. General Wallace Robinson and the staff of NSIA headquarters provided support and sponsorship of a National Conference on Software Test and Evaluation. Mr. Ralph San Antonio of NSIA and Dynamics Research Corporation served as chairman of the conference. Mr. San Antonio also played a key role in coordinating NSIA participation in the data gathering phase.

A number of industrial organizations made important contributions by opening their facilities to the study team surveying contractor practices. A guarantee of anonymity prevents acknowledgement of these organizations and individuals by name. However, it is unlikely that our goals could have been met without the collegial and professional cooperation of these groups. We thank them for their time and patience. In addition, dozens of DoD and military personnel and civilian professionals were interviewed during the data gathering effort. Their hospitality and spirit of cooperation eased a difficult task for the survey teams.

Dr. Richard A. DeMillo, Principal Investigator

Volume 1

Final Report and Recommendations

Table of Contents

Page

Executive Summary 1

1. Introduction 8

2. Rationale 10

3. The Software Test and Evaluation Project 15

4. State-Of-The-Art Assessment 16

4.1. Test Methodologies 17
4.1.1. Test Strategies 17
4.1.2. Testing Techniques 17

4.2. Testing Tools 21

4.3. New Technology 23

4.4. Summary of the State-of-the-Art 24

5. Assessment Of Current Defense Practices 25

5.1. Survey Methodology 26

5.2. Survey Results 27
5.2.1. HQ and Development Commands 27
5.2.2. Project Offices 28
5.2.3. OT&E Agencies 29
5.2.4. Defense Contractors 29

5.3. Summary of Current Defense Practices 33

6. Findings And Recommendations 34

6.1. Modification of DoDD 5000.3 35

6.2. Other Recommendations to DDT&E 38

6.3. Recommendations Addressed to the Military Services 39
6.3.1. State-of-the-Art Improvements 39
6.3.2. Lifecycle Integration 41
6.3.3. Test Planning 41
6.3.4. The Testing Process 44
6.3.5. Test Evaluation 50

6.4. Tri-Service Recommendations 55

7. Concluding Remarks 56

Appendix: Project Organization and Milestones 58

STEP - Final Report and Recommendations

EXECUTIVE SUMMARY

The Software Test and Evaluation Project (STEP) was initiated in 1981 by the Director Defense Test and Evaluation. The primary objective of STEP is to develop new DoD guidance and policy for the test and evaluation of computer software for mission-critical applications. A number of subsidiary goals have also been established for STEP. Principal subgoals include the stimulation of tool development, the support of policy development, and the identification of research issues and directions in the area of software testing.

STEP is conceived in four phases: information gathering, analysis, assessment of feasibility, and policy development. The chief goal of Phases I and II has been to determine the feasibility of modifying and reformulating Defense policy for the test and evaluation (T&E) of software. In support of the feasibility assessment, a broad overview of the state-of-the-art and the current state of Defense practices in software T&E was constructed. Input was sought from DoD components, industrial representatives, selected experts and consultants, and specially convened workshop and symposium participants. In addition, extensive surveys of both the software T&E literature and vendors of automated software T&E tools were prepared. These sources provided a consistent view of software T&E needs and capabilities. Phases III and IV of STEP are yet to be completed. Phase III consists primarily of the assessment of whether new policy guidance can be formulated. Phase IV represents the actual development of policy statements and implementation strategies.

RECOMMENDATIONS

We propose 28 specific recommendations for improvements in software test and evaluation. These recommendations have been formulated by the information gathering and analysis mechanisms described above and have been influenced by the widest possible participation from industry, academia and the Military Services.

MODIFICATIONS OF DODD 5000.3

The first three recommendations below address modifications to DoDD 5000.3 and the Test and Evaluation Master Plan (TEMP). The effects of the recommendations are to (1) establish a chain of T&E plans and evaluation criteria that begins at the level of system test objectives and proceeds through the detailed testing of software components within development organizations, (2) insert existing technology into the T&E process using software that represents the highest decision risk as the focus of the software test plan, and (3) establish the TEMP as the major planning document for software testing and ensure the early incorporation of software test issues into the overall test program.

1. DoDD 5000.3 (Section D, POLICIES AND RESPONSIBILITIES, Part 6 "Test and Evaluation of Computer Software") should be modified to include the following requirements:

a. Software components implementing critical functions shall be identified. These components shall be tested throughout the development/integration portion of the software lifecycle. Results of tests shall be objective, repeatable, available to subsequent test groups, and interpretable in terms of overall system objectives.

b. The level of test of software components that implement critical functions shall be sufficient to demonstrate that the appropriate software evaluation criteria goals for that component are met or exceeded. The level of tests for these components should be sufficient to achieve a balanced risk with the hardware on which they are implemented in an operational environment.

2. DoDD 5000.3 (Enclosure 2) should be modified to require the incorporation of software-specific test and evaluation issues in the TEMP for systems with mission-critical software components. Deviations from the software-specific portion of the TEMP should be subjected to critical review. The portions of the TEMP which should include software-specific information are:

Part I - Description, 2. System, a. Key functions: Should also include a mission/function matrix relating the primary functional capabilities of each critical software component that must be demonstrated by testing to the mission(s) to be performed and concept(s) of operation.

Part I - Description: Should include a new section entitled Required Software Characteristics following Required Operational and Technical Characteristics (Items 3 and 4). This section should contain a list of the key software characteristics, goals, and thresholds.

Part I - Description, 5. Critical T&E Issues: Should include a new sub-section, c. Software Issues. This sub-section should briefly describe key software issues that must be addressed by testing.

Part II - Program Summary, 1. Management: Should also highlight arrangements between participants for software test data sharing.

Part II - Program Summary, 2. Integrated Schedule: Events to be displayed on the schedule should also include key software subsystem demonstrations and software testing tools availability dates.

Software T&E Outline: This new part should follow Part IV - OT&E Outline. This part should discuss all planned software T&E, for software components which implement critical functions, in similar format and detail as that described in the DT&E Outline (Part III). The Software T&E to Date section, which sets the stage for discussion of the planned software T&E, should summarize the software T&E already conducted and emphasize software events and results related to required software characteristics and critical software issues. This section and the Future Software T&E section should discuss the degree to which the test environment is representative of the expected operational environment. The section on Software T&E Objectives should present the major objectives that, when achieved, will demonstrate that the software development effort is progressing satisfactorily. The objectives either should be presented in terms of, or related to, the software characteristics. The Software T&E Events/Scope of Testing/Basic Scenarios section should relate the testing to be performed to the Software T&E Objectives. The Critical Software T&E Items section should highlight all items the availability of which are critical to the conduct of adequate software T&E prior to the next decision point. If appropriate, these critical items should be displayed on the Integrated Schedule. When the required software T&E information is contained in Parts III and/or IV, references may be made to those sections, as appropriate.

Part VI - Special Resource Summary, 1. Test Articles: Should also identify as test articles each software component that is identified in the mission/function matrix and key software subsystems shown in the Integrated Schedule.

Part VI - Special Resource Summary, 2. Special Support Requirements: Should also identify software test tools (including simulators) required, justify each tool identified (describe how the tool supports the software test objectives, achieves a specified level of test, etc.), and briefly describe the steps being taken to acquire each tool.

3. DoDD 5000.3 (Enclosure 1) should be modified to include the following terms and concepts:

Software Lifecycle. Extends from requirements definition and design through operation and maintenance.

Level of Test. Used in conjunction with a systematic software test methodology and is used to rank the thoroughness of a test with respect to the goals set for the evaluation criteria (e.g., 95% statement coverage vs. 50% statement coverage).

Software Evaluation Criteria. Standards by which achievement of required software characteristics, or resolution of software issues may be judged.

Required Software Characteristics. Software parameters that are primary indicators of conformance to written requirements/specifications and operational suitability and effectiveness.

OTHER RECOMMENDATIONS TO DDT&E

Whereas the implementation of the recommendations presented above can be accomplished in the near-term with few changes in the T&E process and only slight modification of the TEMP, the next three recommendations address issues that cannot be resolved so easily. However, if we are to realize the full benefits of the modifications to DoDD 5000.3, these recommendations must also be implemented.

4. DDT&E should initiate or participate in an on-going program of software testing tools development, packaging, evaluation, distribution, and support to provide a warehouse, catalog, or test environment of approved testing tools which can be referenced in the software portion of TEMP without acquisition or further approval.

5. DDT&E should define a model of the software testing process which is well-integrated with the software development lifecycle. In the event that software T&E cannot be accommodated by the DT&E/OT&E/PAT&E structure, a separate software T&E program should be developed.

6. DDT&E should define software evaluation criteria for software in the following categories: (1) necessary testing on support software, (2) risk reduction on software that is required for system operation but does not directly implement mission-critical functions, (3) testing of other software design components. This definition should form the basis for a quantitative risk model of the software T&E process to be used in the evaluation of the overall software testing effort.

RECOMMENDATIONS ADDRESSED TO THE MILITARY SERVICES

Implementation of the recommendations listed above requires a coordinated examination of software T&E technology and practice by DDT&E and the Military Services.

The following additional recommendations are intended to (1) implement DD&TE recommendations, (2) support DDT&E recommendations in areas that are not addressed directly by DoDD 5000.3, and (3) improve the software testing process.

State-of-the-Art Improvements

7. Major initiatives to improve software technology should include early provisions for software test and evaluation.

8. The Services should continue research funding at an accelerated pace for software test and evaluation methodologies and the tools to support these methodologies. Research should also concentrate on establishing usage contexts for the methodologies, cost/benefit analyses, and experimental determination of error detection capabilities.

9. A major focus of military organizations responsible for software development environments should be the identification, qualification, and distribution of tools which implement state-of-the-art testing techniques.

10. AJPO And the affected Military Services should begin now to modify and expand APSE development plans to include substantial provisions for test support environments. Test support tools should be made available in the first generation of APSE's that are used to develop software-intensive systems.

Test Planning

11. Program Offices should encourage and support the development of written test plans for tests to be conducted during early phases of software development. These plans should (1) contain a specification of what constitutes an acceptable approach to testing, (2) explain how the approach adopted supports objectives of the higher level tests, (3) be adhered to rigorously by Program Managers, (4) be critically reviewed for deficiencies, and (5) reflect a realistic, worst-case estimate of the scope and extent of the required testing effort.

12. Project offices should require documentation of unit and module tests. Documentation requirements should include resource requirements, simulation requirements for inputs, analysis requirements for outputs, test case cross references to system requirements and sufficient supporting information to allow the reconstruction and repetition of tests.

13. Project offices should ensure that provisions are made for regression testing in all test plans. In the absence of a major improvement in the state-of-the-art in regression testing, auditing and retesting procedures for all software, specification, and requirements modifications and updates should be required.

14. When IV&V is required by the project office, the involvement of the IV&V contractors should be planned and integrated into the overall testing effort. Project offices should ensure that test plans contain provisions for IV&V involvement.

The Testing Process

15. Project offices should set goals for the testing of the total software system, including those components not specified in the TEMP. These goals should be incorporated into a written test plan as a set of software test objectives. The nature and extent of the testing required for these components should be sufficient to achieve a balanced risk with mission-critical components.

16. Development test organizations should resolve major software deficiencies before the start of dedicated OT&E.

17. Operational test documentation and results should be an integral part of the overall software test database.

18. Project offices should ensure that unit and module tests exercise critical functions with a systematic test methodology. In selecting a test methodology, primary considerations should be the appropriateness of the methodology, known cost/benefit ratios, established error detection capabilities of the methodology, and the extent to which test results are interpretable in terms of software test objectives set forth in the TEMP. The relationship between tests performed and the errors to be discovered must be explicit in the test methodology. This relationship should be a principal consideration in determining the appropriateness of the test.

19. The Military Services should encourage and support the development of testing techniques that take into account quality measurements other than correctness.

20. Implementation of effective practices for software T&E require the Military Services to initiate on-going programs to develop, package, evaluate and maintain testing tools. Included in this effort should be a program to identify and qualify tools for early use in the development cycle. The qualification requirements should specify usage contexts for specific tools and comparative analyses of costs and effectiveness of individual tools should be provided. Provisions should be made for generalizing and improving tools which implement state-of-the-art test techniques and strategies.

21. The Military Services through their project offices should consider reevaluating contract funding patterns to allow special purpose tools developed in support of the contract deliverables to become deliverable items under the same contract.

22. DDT&E in coordination with the management of STARS and the Military Services should investigate the possibility of including the software T&E tools warehouse in one or more STARS task areas. In particular, the process of identifying, packaging, qualifying and distributing test tools for use in support of test plan requirements should be a key role for the STARS Software Enginerring Institute or its functional equivalent.

Test Evaluation

23. The Military Services and DDT&E should develop quantitative indices of software testing progress during development. Quantification should treat both costs and risk: (1) Reliable cost/benefit measures for testing software should be developed and the cost/effectiveness of testing tools should be established. (2) Quantitative risk analysis techniques for software errors should be developed. (3) Cost and risk should be used as essential factors in determining quantitative indices.

24. Military labs should expand their efforts to provide an improved data gathering, reduction, and measurement capability to project offices and developers. Automated data logging and data base systems should be developed to track and record errors on software-intensive systems. The relationship between measurable characteristics of software products and the processes used to produce them should be validated. Measurable characteristics which are reliable predictors of software quality should be applied to enhance the evaluation process.

25. An effective software quality assurance standard should be developed.

26. The Military Services should determine the cost/benefit aspects of IV&V and recommend the conditions under which IV&V should be required.

27. DDT&E and the Service Program Offices should begin now to develop an integrated decision support system for software T&E that combines functionally organized test information and evaluations with data that is required for major programmatic decision points.

Tri-Service Recommendations

28. The Military Services should develop tri-service standards to make unified approaches to software development, testing and evaluation possible.

CHAPTER 1

INTRODUCTION

The Software Test and Evaluation Project (STEP) was initiated in 1981 by the Director Defense Test and Evaluation. The primary objective of STEP is to develop new DoD guidance and policy for the test and evaluation of computer software for mission-critical applications. A number of subsidiary goals have also been established for STEP. Principal subgoals include the stimulation of tool development, the support of policy development, and the identification of research issues and directions in the area of software testing.

STEP is conceived in four phases: information gathering, analysis, assessment of feasibility, and policy development. This report is Volume 1 of a six volume series prepared under a contract to the Georgia Institute of Technology in support of the first two phases. The remaining volumes in the series are the following:

Volume 2:	Software Test and Evaluation: State-of-the-Art Overview
Volume 3:	Software Test and Evaluation: Current Defense Practices Overview
Volume 4:	Transcript of STEP Workshop, March 1982
Volume 5:	Report of Expert Panel on Software Test and Evaluation
Volume 6:	Tactical Computer System Applicability Study

The organization and contents of the overall report will be discussed in the sequel.

In this volume, we will present the rationale for seeking improved DoD guidance in software test and evaluation, the organization of STEP and its support contracts, a summary of our data and information gathering efforts, and Phase I and II findings and recommendations.

The chief goal of Phases I and II has been to determine the feasibility of modifying and reformulating Defense policy for the test and evaluation (T&E) of software. In support of a feasibility assessment, a broad overview of the state-of-the-art and the current state of Defense practices in software T&E was constructed. Georgia Tech and its subcontractors, Control Data and Clemson University, sought input from DoD components, industrial representatives, selected experts and consultants, and specially convened workshop and symposium participants. In addition, extensive surveys of both the software T&E literature and vendors of automated software T&E tools were prepared. These sources provided a consistent view of software T&E needs and capabilities.

Our findings support the view that there is a need for modified and improved policy guidance for the testing of software systems destined for embedded and mission-critical military applications. Test and evaluation is a risk reducing activity. The Director Defense Test and Evaluation (DDT&E) is charged with overall responsibility for testing in the Department of Defense. DDT&E reviews the results of testing on major weapon and support systems, assesses the adequacy of tests and planned tests and conveys these assessments to decision makers in the DoD. The assessment of test adequacy is translated into statements of risk that are used to determine whether a system is ready to advance from one programmatic phase to the next. Assessing the adequacy of tests of more traditional system components is already a difficult activity. Software -- a critical system component for which there is no accepted quantitative test/risk model -- presents even more fundamental problems in planning tests to comply with guidance and reducing the results of tests to an objective body of information which can be used to assess past and planned testing.

Improved technology makes a more systematic and rigorous approach to software testing feasible. In order to be most effective, this approach must be applied in a planned and coordinated fashion at all phases of the software development process, beginning at the earliest design stages and proceeding through operational testing of the integrated system. On the other hand, like all of the software sciences, software T&E is in its infancy. With appropriate attention and support, the state-of-the-art in software testing can be improved considerably. Finally, the acquisition environment in which software components are developed can be improved. Like most T&E activities, software testing is subjected to budget and schedule constraints that often compress testing unrealistically. Alternative development cycles and acquisition strategies that force the examination of software components in proportion to their importance to the overall objectives of the system will give some near-term relief in these areas.

We recommend below that principal DoD guidance policy, and associated standards, regulations, and Service practices be modified along these lines, and that appropriate support be considered for a major technology upgrade to implement these modifications.

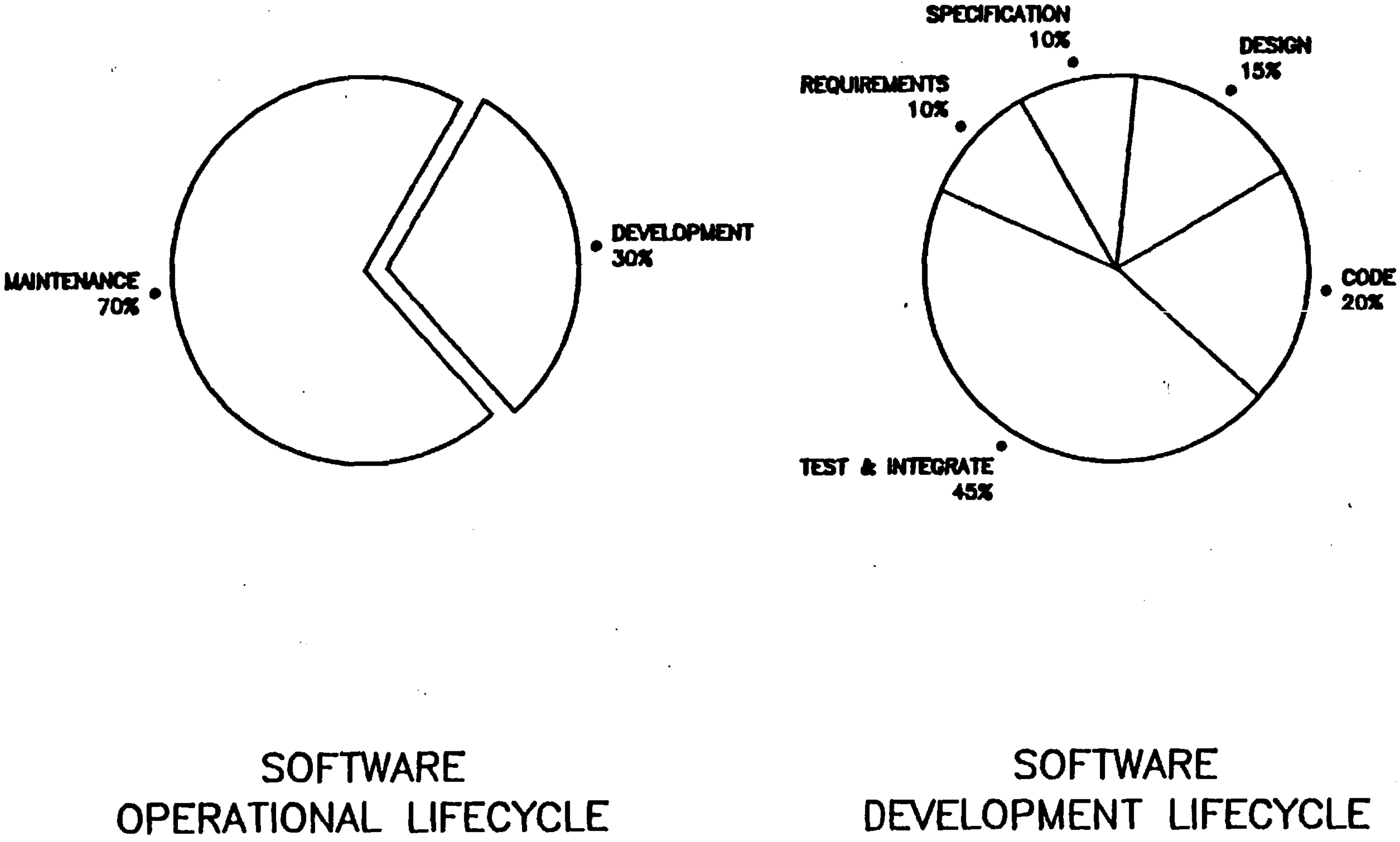
CHAPTER 2

RATIONALE

The role of software in escalating the cost and driving down the reliability of military systems has been very visible in recent years. Virtually every major defense system planned or fielded over the last decade has at least one subsystem consisting of an embedded computer controlling some mission-critical function. The applications literally cover the domain of computer applications. Critical functions of the F/A-18 aircraft are controlled by an avionics suite and the M-1 tank by a fire control system. Complex information gathering, processing and retrieval networks are major components of the AEGIS fleet air defense system and the TACFIRE ground tactical system. The guidance and control functions of such systems as the MAVERICK air-to-ground missile are driven by embedded computers. The range of technology spanned by these applications is also broad. PATRIOT contains microprocessors, while WWMCCS consists of distributed mainframe computers suitable for general purpose computation. In each of these instances, the computer subsystems contain two components of equal importance to the overall operation of the system: hardware and software.

Although hardware and software contribute in equal measure to the successful implementation of system functions, there are relative imbalances in their treatment during system development. In 1974, the Defense Science Task Force on Test and Evaluation observed: "Whereas the hardware development was ... monitored, tested, and regularly evaluated, the software development was not." Since software is essential to the overall objectives of so many important Defense systems, inadequacies in evaluating software components -- especially when the failure of a software component contributes to the failure of a major system -- tend to be highly visible. So visible, in fact, that the test and evaluation of software has attracted attention at the highest levels. For example, the Secretary of Defense has directed the services to "... give priority to development of tools and techniques for testing of embedded computers and software... Testing of software should be sufficient to achieve a balanced risk with the hardware of the same system ..." The Secretary has also stated that "These advances are required if the activities are to provide realistic assessments of system operational capability ..."

Current estimates of increased software costs arising from incomplete testing help to illustrate the dimensions of the problem (see Figure 1). Averaged over the operational lifecycle of an embedded computer system, development costs comprise approximately 30% of the total costs. The remaining 70% of the lifecycle costs are absorbed in maintenance. Maintenance activities can include both system enhancements and the repair of errors. These are errors that might have been



SOFTWARE
OPERATIONAL LIFECYCLE

SOFTWARE
DEVELOPMENT LIFECYCLE

FIGURE 1
EMBEDDED SOFTWARE LIFECYCLE COST DISTRIBUTION

uncovered by more complete testing during earlier phases. In general, observers agree that the cost of finding and fixing errors is an increasing function of the elapsed time from the start of the development process. As shown in Figure 2, the relative cost of repairing errors in software rises dramatically between requirements and specification phases and the maintenance phase. One investigator estimates the cost multiplier to be 10**(2d), where d is the elapsed time of the development effort expressed as a percentage of total development time.

According to the data in Figure 1, costs in the development phase are distributed as follows: requirements and specification development, 20%; design and coding, 35%; test and integration, 45%. Thus, assuming that half of all maintenance costs are incurred in the repair of previously undetected errors, approximately one half of the operational lifecycle costs for embedded applications can be traced directly to testing activities; that is, either these costs are incurred by testing or are due to errors left undiscovered by testing. Of course, there are other cost implications of undetected errors in military systems. The mission-critical nature of software in many modern systems means that software which fails during system operation can pose considerable risk to both the success of the mission and the safety of personnel.

Primary DoD guidance for test and evaluation derives from DoD Directive 5000.3. This directive applies to both hardware and software components of military systems and sets forth the framework within which more specific military regulations and standards must operate. Three provisions of DoDD 5000.3 are particularly relevant to software testing. First, DoDD 5000.3 states that "Quantitative and demonstrable performance objectives and evaluation criteria shall be established for computer software during each system acquisition phase ... Decisions to proceed from one phase of software development to the next will be based on quantitative demonstration of adequate software performance through appropriate test and evaluation." Second, DoDD 5000.3 requires that software be operationally tested using "typical operator personnel." Third, DoDD 5000.3 requires that operational test and evaluation (OT&E) agencies "participate in software planning and development to ensure consideration of the operational environment and early development of the operational test objectives."

For a variety of reasons, existing guidance statements have not had the desired effect. For example, a key factor is the vagueness of the concept of "adequate software performance" and the perceived unavailability of techniques which provide the requisite "quantitative demonstration." While testing of hardware components may result in a database of quantitative test results against which reliability and risk models may be applied, software components are seldom accompanied by objective evidence of the effectiveness of the testing effort. The critical issue of "how much testing is enough?" for software and how that testing should be conducted, reported, and integrated into the key phases of a major system development has simply not been resolved to a useful degree. The difficulties in planning tests to comply with

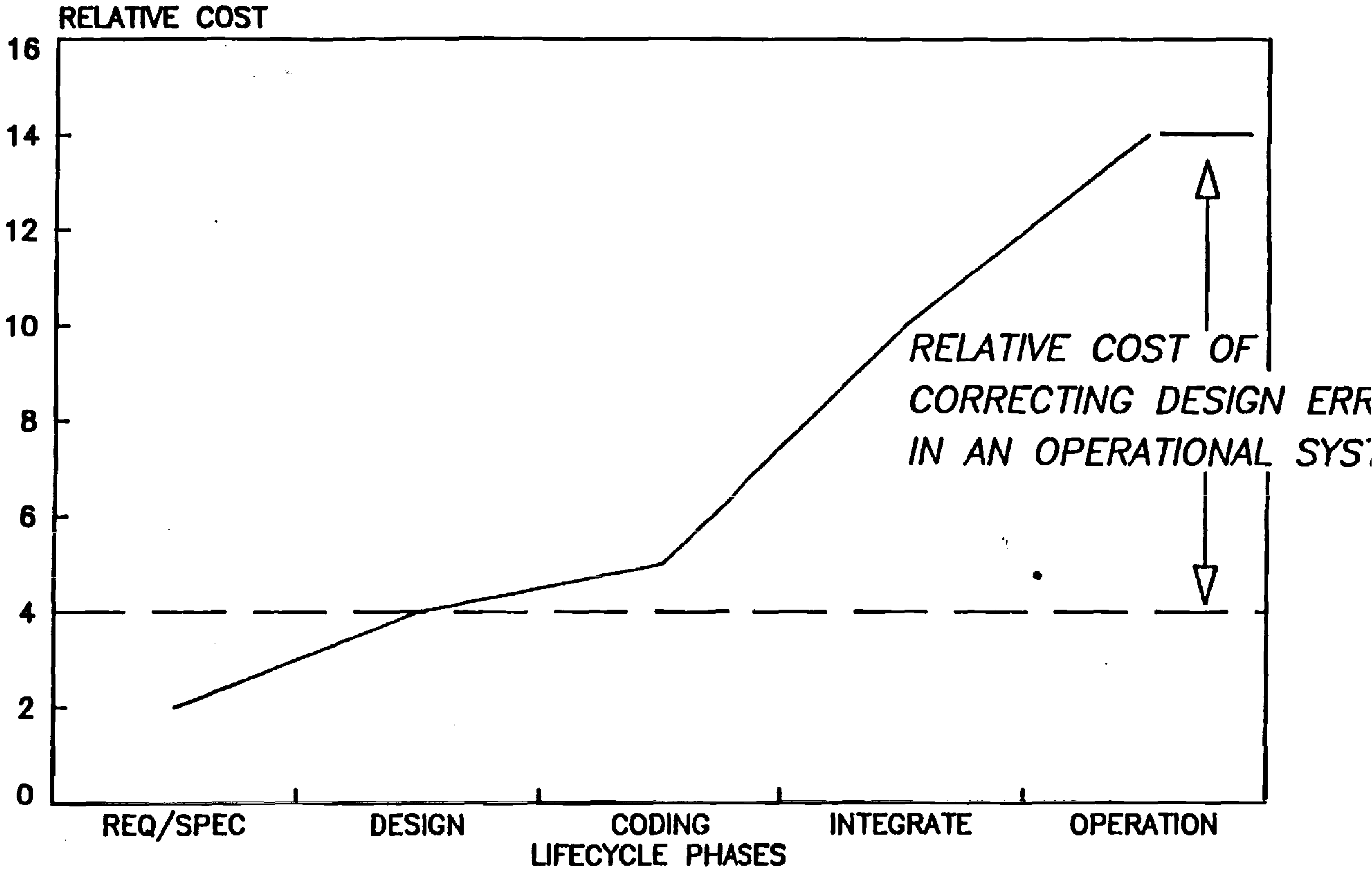


FIGURE 2

RELATIVE COST OF ERROR CORRECTION

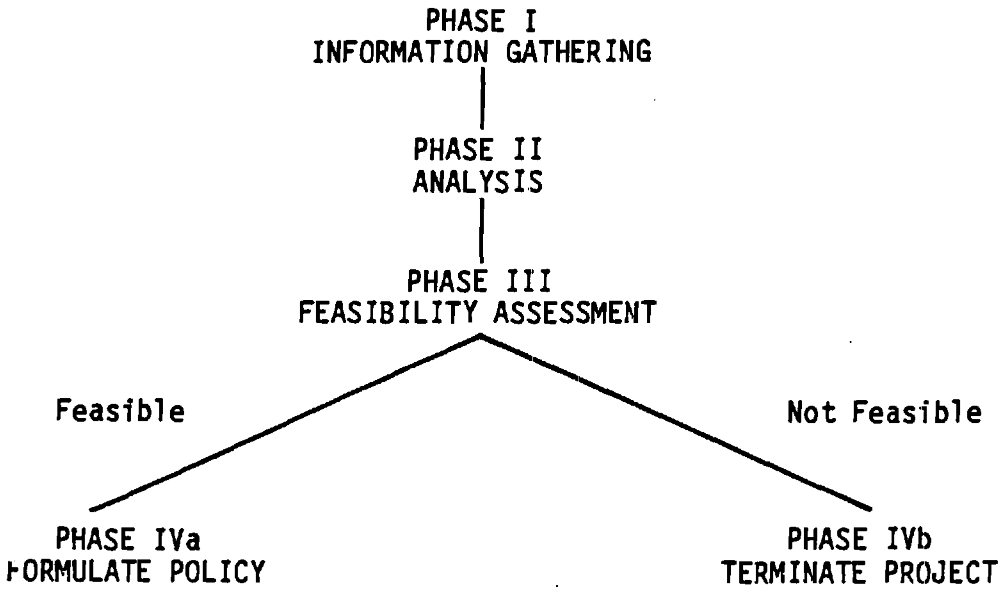
existing guidance, standards and regulations increase with the number and complexity of embedded software systems. These difficulties are highlighted in acquisition environments in which schedules and budgets squeeze the T&E effort.

In recognition of these problems and the developing software technology which can address them, STEP was initiated by the Director Defense Test and Evaluation.

CHAPTER 3

THE SOFTWARE TEST AND EVALUATION PROJECT

STEP consists of four phases intended to lead to improved DoD guidance for software test and evaluation. The current report represents the results of the primary support contracts for STEP Phases I and II. Phase I was an information gathering effort aimed at assessing the state-of-the-art and the state of current practice in software T&E. During Phase I, an extensive survey of known techniques and tools was compiled. The assessment of the state of current practice was made by surveying DoD agencies, the Military Services, program offices, independent test organizations, and selected Defense contractors. Phase II consisted of an analysis of the information gathered in Phase I and the formulation of the recommendations which appear in this report. Phases III and IV of STEP are yet to be completed. Phase III consists primarily of the assessment of whether new policy guidance can be formulated. Phase IV represents the actual development of policy statements and implementation strategies. Thus, the overall structure of STEP may be represented as shown in the diagram below.



The Appendix describes the organization of the support contracts, the principal milestones of the Phase I and II support efforts, and the contents of the remaining volumes of this report.

CHAPTER 4

STATE-OF-THE-ART ASSESSMENT

Current research in software testing centers almost solely on testing for correctness; that is, on techniques that raise the users' confidence that the software functions in accordance with its specifications. "Testing" refers specifically to the activity of executing software on data (the test sets) designed to either reveal the presence of errors or ensure their absence. Therefore, software testing is distinguished from other activities aimed at increasing software reliability (such as structured design techniques, formal program proving, and statistical reliability modelling).

Three aspects of extant research efforts in software testing are relevant for assessing the state-of-the-art: testing methodologies (i.e., methodologies for either generating tests or determining the quality of previously generated test sets), testing tools (i.e., automated systems which implement one or more testing methodologies), and new hardware and software technologies which impact system reliability. In the subsections below, we will briefly outline the state-of-the-art in each of these three areas. A more detailed treatment of each of these topics can be found in Volume 2.

4.1. TEST METHODOLOGIES

A test methodology consists of two (not always distinct) components. The first is a strategy which guides the overall testing effort, while the second is a testing technique which is applied within the framework of a test strategy.

4.1.1. TEST STRATEGIES

Module testing is the process of testing logical units of a program and integrating the individual module tests to evaluate the overall system. Main considerations in module testing are the design of test cases and the coordination of testing multiple modules. Test cases may be constructed from specifications or by analyzing the module code. Testing strategies corresponding to these approaches are called black-box and white-box strategies, respectively. There are two approaches to combining module analysis: incremental and non-incremental. Top-down and bottom-up testing are two incremental approaches. Thread testing is another strategy based on system requirements. Strategies have also been proposed for testing software throughout its development. Finally, several new strategies have been proposed based on an "evolutionary" view of the software lifecycle. In one approach, systems are constructed as working subsystems corresponding to critical functions, and these subsystems are subjected to development and operational tests.

4.1.2. TESTING TECHNIQUES

A variety of testing techniques have been proposed in the literature (see, e.g., Volume 2). These techniques can be classified as follows: static analysis, symbolic testing, program instrumentation, program mutation, input space partitioning, functional program testing, algebraic program testing, random testing, grammar-based testing, data-flow guided testing, and real-time testing.

STATIC ANALYSIS. In static analysis, the requirements, design documents, and program code are analyzed without actually executing the code. Only limited analysis of programs containing dynamic data types and structures is possible using static analysis. Experimental evaluation of code inspections and walk-throughs has found these techniques to be effective in detecting from 30% to 70% of the logic design and coding errors in typical programs.

SYMBOLIC TESTING. To test a program symbolically, input data and program variable values are given formal or "symbolic" values. The possible executions of a program are also characterized formally. The execution of the program is then simulated by a symbolic evaluator which interprets the formal representation of the program and data. The techniques for building expressions which describe the state of the symbolic execution of a program lean heavily on techniques developed for proving program correctness. Studies describing the effectiveness of symbolic analysis for detecting errors indicate that it may be an effective technique for moderately large modules.

PROGRAM INSTRUMENTATION. Programs can be instrumented by statements or routines that do not affect the functional behavior of the program, but record properties of the executing program. Additional output statements, assertion statements, monitors, and history-collecting subroutines may be used to instrument programs. Experimental evaluations of instrumentation techniques indicate that experienced testers can decrease the debugging time for even complex programs using these techniques.

PROGRAM MUTATION. Program mutation is a technique for the measurement of test data adequacy. Test adequacy refers to the ability of the data to ensure that certain errors are not present in the program under test. In mutation testing, test data is applied to the program being tested and its "mutants" (i.e., programs that contain one or more likely errors). If a program passes a mutation test, then either the program is correct or it contains an improbable error. Experimental evaluation of mutation testing indicates that the results of mutation testing are good predictors of operational reliability.

INPUT SPACE PARTITIONING. A path in a program consists of a possible flow of control. In path analysis techniques, the input space of a program is partitioned into path domains: those subsets of the program input domain that cause execution of the paths. Path analysis can detect computation, path, and missing path errors. Domain testing detects many path selection errors by considering test data on or near the boundaries of path domains. In partition analysis, the specification of a program is partitioned into subspecifications. The subspecifications are then matched with domain partitions to increase the sensitivity of the test. All of these techniques have been shown theoretically and experimentally to be generators of high quality test data, although current technology limits their use to programs which have a small number of input variables.

FUNCTIONAL TESTING. In functional testing, the specification of a program is viewed as an abstract description of its design. Function and data abstractions are used as guides to identify the abstract functions of a program and to generate the functional test data. Functional testing requires the specification of domains for each input and output variable of the program. Extremal and special values are the most important values in the domain of a variable. In a study of errors that occurred in a release of a major software package, functional testing was effective in detecting 38 out of 42 known errors.

ALGEBRAIC TESTING. In algebraic testing, program correctness is viewed as an equivalence problem. Since the general equivalence problem is undecidable, programs to which this technique is applicable must fall in a restricted class of programs for which execution on a small test set is sufficient to infer equivalence. Applications of algebraic testing to array manipulation programs, polynomial evaluation programs, and other mathematical programs have appeared in the literature. Monte Carlo methods exist for algebraic testing procedures which make the technique tractable for many problems.

RANDOM TESTING. Random testing is essentially a black-box testing technique in which a program is tested by randomly sampling inputs. Depending on the sensitivity of the analysis desired, the sampling technique may be independent of the actual distribution of inputs or may attempt to accurately reflect the distribution of the operational environment. Random testing is useful in making operational estimates of software reliability and has some connection to problems arising in operational testing.

GRAMMAR-BASED TESTING. Formal specifications of some software systems can be given by state diagrams. By considering the state diagram to be a description of an automaton, classical machine identification experiments can be conducted to determine whether or not a program implementing the automaton does so correctly.

DATA-FLOW GUIDED TESTING. Data-flow analysis is a method for obtaining structural information about programs which has found wide applicability in compiler design and optimization. One result of data-flow analysis is a set of dynamically meaningful relationships among program variables. Control flow information about the program is then used to construct test sets for the paths to be tested.

REAL-TIME TESTING. The characteristic phases of real-time software testing occur during development (on the development "host") and operational testing (on the operation "target"). Systematic techniques for testing real-time software during development, for the most part, do not make essential use of the fact that the software is real-time. Testing an integrated system on a development host requires an environment simulator and devices for controlling on-going processes. In testing real-time software on target machines, overall test objectives for the hardware/software system are used, and performance becomes a key observable factor in assessing the result of the tests. While the literature contains very few systematic techniques for real-time testing, some studies of large-scale, real-time software systems tests have been published, and these experiences may generalize to other applications.

4.2. TESTING TOOLS

Testing tools may be classified by whether they carry out static or dynamic analysis of the program under test. Static analyzers are systems that manipulate source code to reveal global aspects of program logic, structural errors, syntactic errors, variations in coding style, and interface consistency. Static analyzers consist of front end language processors, data bases, error analyzers, and report generators. Basic operations include data collection, error analysis, and error report generation. Existing static analyzers differ in terms of their scope of error analysis, the flexibility of user command languages, and the nature of error descriptions. Static analyzers have been used in many reported software development efforts. Dynamic analyzers, in addition to implementing many of the techniques described above, are used to generate test data, provide a convenient test environment, and compare program test output with expected output.

SYMBOLIC EVALUATORS. Symbolic evaluators implement the symbolic evaluation testing technique. They provide the user with the ability to input loop and control point assertions and symbolic values for input variables. They also allow the user to monitor the symbolic execution of the program.

TEST DATA GENERATORS. A test data generator is a tool which assists the user in the preparation of test sets. Three types of generators have been described in the literature: pathwise test generators, specification-based generators, and random generators. Pathwise test generators have four basic operations: program construction, path selection, symbolic execution, and test data generation. Specification-based generators provide the user with a language for constructing test case specifications; the system carries out the actual generation of test files from the test specifications. Random test generators choose random values from the input domain according to statistical parameters set by the user.

PROGRAM INSTRUMENTERS. These systems gather execution data to reveal characteristics of a program's internal behavior and performance. In practice, instrumentation tools are the principal tools used to detect errors that cannot be detected by static analysis. Systems exist which provide coverage analysis, monitor assertions, and detect data-flow anomalies. In addition, instrumentation subsystems can be found in several other types of testing tools.

MUTATION TOOLS. An automatic mutation system is a test entry, execution, and data evaluation system that evaluates the quality of test data based on the results of program mutation. In addition to a mutation "score" that indicates the adequacy of the test data, a mutation system provides an interactive test environment and reporting and debugging operations which are useful for locating and removing errors.

AUTOMATIC TEST DRIVERS. Automatic test drivers are software systems that simulate an environment for running module tests. They may provide standard notation for specifying test cases and automating the testing process. Some systems also compare the resulting output with the expected output and report discrepancies. Some test drivers operate on object modules, while others operate on source modules. Since the automation of the testing process is an integral part of most test tools, automatic test drivers appear in some form in most systems.

COMPARATORS. A comparator is a system that compares two versions of data to identify differences. Comparators are used in the validation process to limit the scope of re-testing of revised software. The main differences among comparators lie in the form of the data and the flexibility in specifying tolerances for each comparison.

Volume 2 contains a catalog of existing tools in each of these categories and a summary of their availability and support. Generally, however, it appears that testing tools which are available as supported, nonproprietary packages are rare. It is more common that testing tools are systems that are constructed and customized to a single software development project. Generalization, documentation, marketing, and support of such custom tools is capital intensive and seldom carried out.

4.3. NEW TECHNOLOGY

Two aspects of new technological developments are relevant to software testing. First, there are new technologies that hold some hope for improving the programming process. New languages such as Ada[1], new views of the software lifecycle, prototyping, and reusable software all give software developers new tools and concepts to work with. Modern operating systems and programmer support environments give programmers collections of tools which will aid in the testing effort. Standard architectures ease the transition from host environments to target environments. It has also become possible to "freeze" certain critical system components in custom hardware. While the problems of determining correctness of design remain in transitions to hardware implementation, the static nature of hardware and the visibility of hardware interfaces may reduce the severity of many testing problems.

Second, new technology presents many new reliability problems. New applications such as distributed computing and communications rely on complex interactions of concurrent processes. These systems have thus far been as resistent to systematic testing techniques as older, real-time applications. Since many of these systems come equipped with stringent reliability requirements, new testing techniques are clearly needed. Customized hardware designs, in addition to providing benefits such as those mentioned above, also present new difficulties. As the density of functions that can be placed on a single chip increases, so does the complexity of the testing effort needed to determine that the designs are correctly implemented in the hardware. Existing hardware design verification techniques do not appear to be mature.

[1] Ada is a Registered Trademark of the Ada Joint Program Office - U.S. Government

4.3. NEW TECHNOLOGY

Two aspects of new technological developments are relevant to software testing. First, there are new technologies that hold some hope for improving the programming process. New languages such as Ada[1], new views of the software lifecycle, prototyping, and reusable software all give software developers new tools and concepts to work with. Modern operating systems and programmer support environments give programmers collections of tools which will aid in the testing effort. Standard architectures ease the transition from host environments to target environments. It has also become possible to "freeze" certain critical system components in custom hardware. While the problems of determining correctness of design remain in transitions to hardware implementation, the static nature of hardware and the visibility of hardware interfaces may reduce the severity of many testing problems.

Second, new technology presents many new reliability problems. New applications such as distributed computing and communications rely on complex interactions of concurrent processes. These systems have thus far been as resistent to systematic testing techniques as older, real-time applications. Since many of these systems come equipped with stringent reliability requirements, new testing techniques are clearly needed. Customized hardware designs, in addition to providing benefits such as those mentioned above, also present new difficulties. As the density of functions that can be placed on a single chip increases, so does the complexity of the testing effort needed to determine that the designs are correctly implemented in the hardware. Existing hardware design verification techniques do not appear to be mature.

[1] Ada is a Registered Trademark of the Ada Joint Program Office - U.S. Government

CHAPTER 5

ASSESSMENT OF CURRENT DEFENSE PRACTICES

The testing and evaluation performed on software developed for DoD applications is influenced by a variety of organizations and guidance documents. The primary guidance which exists with respect to software T&E resides in DoDD 5000.3. The Services implement this directive in regulations which provide further guidance to their activities. In addition, the Development Commands of the Services may supplement the Headquarters' guidance in regulations, instructions, or pamphlets with which their subordinate Commands must comply. The final responsibility for adherence to the guidance rests with the project offices which monitor the activities of the Defense contractors. The Services' independent test and evaluation organizations are responsible for the operational test and evaluation of the systems produced. In order to assess the current Defense practices, the functional groups mentioned above were surveyed on subjects related to software test and evaluation.

5.1. SURVEY METHODOLOGY

The survey methodology consisted of conducting interviews with selected representatives of the military and industrial sectors. These groups included the HQ and Development Commands for the Army, Navy, and Air Force, project offices for selected programs, OT&E agencies, and Defense contractors. The subjects discussed during the interviews spanned the areas of military regulations and standards, reviews and inspections, testing techniques, tools, quality assurance, independent verification and validation, and risk assessment. Although the interviews covered a variety of topics, all were related to the software development process, and therefore, the quality of the final software product.

The survey was not a random sampling of Defense organizations and no attempt has been made to give statistical interpretations to the results. Rather, the study team was guided to selected project offices by the HQ and Development Commands and by OSD. Defense contractors were selected by the study team in consultation with NSIA. Several considerations helped to determine the mix of organizations selected for interviews. These considerations included the size of the organization and the type of software activity. The overall goal of the interview selection process was to give the most representative picture possible of current contractor practices. The interview results showed a high degree of similarity. The lack of significant deviation in the responses of these organizations is evidence that if, in some cases, current practices do differ significantly from what is described, those differences are most likely unique to the specific circumstances of the program or contractor involved rather than representative of the norm in the testing and evaluation being performed on military software systems today.

To aid in the data gathering effort, a set of data gathering guides was developed, consisting of one guide for each functional group being interviewed. The guides ensured that the same basic information was gathered during interviews with representatives of each functional group. The use of personal interviews rather than the mass mailing of questionnaires helped circumvent the problems of differing terminologies and low response rates.

5.2. SURVEY RESULTS

In the following subsections, we will describe the information requested from each of the functional groups during the interview process, and present highlights of the information gathered.

5.2.1. HQ & DEVELOPMENT COMMANDS

Interviews were conducted with representatives of the Headquarters and Development Commands for the Army, Navy, and Air Force. The primary purpose of these interviews was to determine what guidance the Headquarters receive from the Department of Defense with respect to software T&E, what guidance they pass on to the Development Commands, and how the Development Commands assist the individual project offices.

The primary guidance given to the DoD components regarding software test and evaluation is DoDD 5000.3. Each of the Military Services has implemented DoDD 5000.3 in regulations applicable to their specific circumstances. The key regulations of interest to us are Army Regulation 70-10, the Navy TADSTAND's, and Air Force Regulations 80-14 and 800-14.

Military Standards also exist for use by contractors who are developing software for military applications. These include:

MIL-STD-1679 (Navy)	Weapons Systems Software Development
MIL-S-52779A	Software Quality Assurance Program Requirements
MIL-STD-1521A (USAF)	Technical Reviews and Audits
MIL-STD-490	Specification Practices
MIL-STD-483 (USAF)	Configuration Management Practices

For a summary of these standards and other guidance documents, see Volume 3.

In addition to the existing standards, the Joint Logistics Commanders have been directing efforts to produce tri-service standards. This has resulted, in part, in MIL-STD-SDS on "Defense System Software Development". MIL-STD-SDS establishes requirements with respect to software requirements analysis, design, code, test, configuration management, quality programs, and project planning and control. It should be noted that although MIL-STD-SDS is currently in the review process, some contractors are requesting waivers to use it as an alternative to other standards. The potential benefits of MIL-STD-SDS include that it addresses the entire software lifecycle, provides uniform terminology and definitions, and is for use by all of the Military Services.

5.2.2. PROJECT OFFICES

Interviews were also conducted with representatives of specific project offices for major systems which are currently under development. During these interviews, information was gathered on project status and history, military regulations and standards invoked, reviews conducted, development test and evaluation, acceptance testing, quality assurance programs, independent verification and validation activities, operational test and evaluation, and risk assessment.

One result of these interviews was the discovery of the complete faith which the military acquisition organizations place in their contractors. This is evidenced by the lack of formal procedures for tracking progress during the coding, module testing, and integration testing phases of the software development life cycle, and a lack of effective government involvement in the software development process. The distance maintained between the software development contractors and the project offices may, in fact, be due to the shortage of personnel in the project offices who are "software-qualified". In addition, the turnover of personnel creates problems with continuity in the knowledge of projects. In some cases, IV&V contractors are hired as an extension of the project offices to support their efforts.

Although it is recognized that software development is expensive, the resources allocated to development and testing are seldom sufficient. When problems arise, activities compressed relate to testing and quality assurance.

To compound these problems, contracts levy few reasonable requirements for software testing and evaluation. With no objective evidence to indicate an effective testing strategy for a given application, it is unrealistic to specify any particular testing strategy. Problems have also been attributed to the apparent weakness of MIL-S-52779A, "Software Quality Assurance Program Requirements".

Many of the difficulties encountered can be traced to inadequacies in the requirements definition process. When adequate requirements are provided, experience suggests that systems can be developed within cost and schedule constraints. However, given the complexity of today's systems and the susceptability of the requirements to change, complete, accurate requirements are rare.

The government's primary involvement with the system development prior to acceptance takes the form of the Preliminary and Critical Design Reviews (PDR's and CDR's). The usefulness of these reviews for the software has been questioned since, in many cases, there are 100 cr more participants. When software acceptance tests are finally performed, they are often a combination of selected tests previously conducted by the contractor. Government personnel may witness these tests, however, care must be exercised since the contractor has advance access to the tests.

Detailed assessments of the potential risks when software failures occur are only conducted when the application has nuclear implications. In other cases, past experience dictates the amount of testing considered to be necessary and sufficient for a given software application.

5.2.3. OT&E AGENCIES

Each of the Military Services has an organization which has been given the mission to operationally test and evaluate new and modified systems. These OT&E Agencies are the Operational Test and Evaluation Agency (OTEA - Army), the Operational Test and Evaluation Force (OPTEVFOR - Navy), and the Air Force Test and Evaluation Center (AFTEC). Due to the special section in DoDD 5000.3 on Test and Evaluation of Computer Software, groups which specialize in software T&E have been formulated within each organization.

Early and continued involvement of OT&E Agency software specialists in the software development process is encouraged and emphasized. When possible, this involvement includes attending Computer Resource Working Group Meetings, PDR's, and CDR's. In some cases, acceptance testing is witnessed.

There is a widespread belief that the "real" problems with software are best found in the operational environment. OT&E personnel are interested in software quality measures other than correctness, i.e., the software's operational effectiveness and suitability. Along those lines, the Software Evaluation Element of AFTEC has developed a set of handbooks to aid in this type of assessment.

Unfortunately, during operational testing, the software is usually only singled out on an exception basis. In addition, since OT&E takes place after the completion of software development, any errors detected may be extremely expensive to correct. The desired early and continued involvement is not always possible due to personnel shortages similar to those which constrain the project offices. In any case, the effectiveness of PDR's, CDR's, and acceptance testing was discussed earlier.

5.2.4. DEFENSE CONTRACTORS

Interviews were conducted with twelve Defense contractors. These contractors are involved in the development of applications software, the development of support software, and the independent verification and validation of military software systems.

APPLICATIONS SOFTWARE DEVELOPERS. Six contractors were interviewed with respect to their efforts toward developing applications software for embedded or mission-critical computer systems. The customers dealt with spanned the Military Services and many other DoD components. The subjects discussed included military and internal standards; requirements, design, and code analysis techniques; the levels of testing performed; tools; quality assurance; independent verification and validation; and risk assessment.

Changing requirements are a constant frustration to Defense contractors. Orderly, structured approaches to software development are easily frustrated by the modifications which are requested throughout the life cycle. Baselining and configuration management techniques are essential for dealing with fluid requirements. Moreover, contract performance seems to be tightly coupled to the effectiveness of these techniques.

Plans for testing begin early in the software development life cycle, usually prior to or during the design phase. Unfortunately, these planning activities often fall victim to the pressures of the immediate development phase.

Similar to the faith which the government places in its contractors is the faith which the contractors demonstrate in their programmers. The greatest opportunity for thorough testing of the software exists at the unit or module level. This testing is seldom subjected to formal requirements for coverage or documentation. A module is tested until the programmer is "satisfied" that it is ready for integration. The prevalent testing strategy is one which tests each functional requirement. Tests which may be necessary due to design or implementation peculiarities are often neglected.

Some contractors implement internal testing standards and practices. However, audit procedures to ensure that the contractors' standards and practices are followed and effective are lacking. The documentation which is delivered is often inadequate. Though very important, it is usually produced after the fact, rather than as the software development process progresses.

The concept of endurance testing as espoused by MIL-STD-1679, "Weapons System Software Development", is criticized by some Defense contractors. The most common complaints center on the cost and appropriateness of endurance testing for software. In addition, although acceptable occurences of errors are assigned according to the criticality of failure, testing requirements do not differentiate between software modules which do or do not implement critical functions.

Regression testing is also an area of concern for many contractors. The need for human intervention during software system level tests makes regression testing time consuming and expensive. "How much is enough?" is a question of primary importance. The completeness and correctness of test sets used during regression testing is maintained as a by-product of the software trouble reporting system. Although this method may ensure that new functionality will be tested, it is not necessarily effective for recognizing the existence of obsolete and incorrect test cases.

Contractors recognize the influence that tools and people have on the success of a software testing effort. Unfortunately, testing tools have traditionally suffered due to a lack of investment. Those which are developed and used are predominately project-specific and rarely examined for possible application on other programs. Extensive waste of resources occurs as a result of the repeated "reinvention of the wheel". In addition, the talented, creative personnel needed for testing are often assigned to and prefer development activities.

Finally, some Defense contractors question the value of software quality assurance (SQA) as it is currently practiced. SQA is viewed as a non-technical function which could be much more useful if it were enchanced to meet the technical need of evaluating the effectiveness of the development and testing techniques employed.

SUPPORT SOFTWARE DEVELOPERS. Two organizations which develop support software were also interviewed. Although the subject areas discussed were identical to those discussed with the applications software developers, the interviews conducted with these contractors centered upon the development and certification of compilers. The major difference between the testing of applications software and support software is the degree of automation used. In each case, a standard and extensive set of certification tests are run prior to each release. Very little human intervention is needed either when running these tests or when checking the results.

IV&V ORGANIZATIONS. Independent verification and validation (IV&V) is a risk reducing technique which is applied to many major programs under development today. Four industry contractors whose primary function is to conduct an independent evaluation of the software development efforts of another contractor were interviewed. Due to the high cost of IV&V, the activities described were usually only performed for a portion of any software system. The information gathered during these interviews pertained to military regulations and standards, the scope of the IV&V effort and the time of initial involvement, the relationship to the project office and development contractors, requirements, design, and code analysis techniques, independent testing, tools, metrics, and risk assessment.

The IV&V organizations stressed the importance of early involvement. Their role is that of a technical resource reporting directly to the project office. Since, requirements analysis and risk assessment are the most critical activities performed by the IV&V organizations, their effectiveness is limited if they are brought on board a project "after the fact". As the development effort progresses, IV&V involvement typically decreases due more to the expense of continued involvement rather than a lack of need.

5.3. SUMMARY OF CURRENT DEFENSE PRACTICES

Many problems combine to limit the effectiveness of current Defense practices in software test and evaluation. First of all, the resources allocated to software development and testing may fall short of what is necessary to ensure required operational capabilities. Second, the project offices, tasked with tracking Defense contractor progress, may lack sufficient software-qualified personnel. Third, there is little objective evidence available to aid Defense contractors in selecting the testing strategy which is best suited to a given application. Fourth, few testing tools which could help ensure the quality of the final software product are readily available for use. Fifth, the evaluation of early development testing is difficult since the results of these tests are rarely documented or reported. All of these shortcomings are amplified by frequent modifications to requirements. And, of course, when budgets are cut or schedules slip, testing and quality assurance activities are the first casualties.

CHAPTER 6

FINDINGS AND RECOMMENDATIONS

This section proposes 28 specific recommendations for improvements in software test and evaluation. These recommendations have been formulated by the information gathering and analysis mechanisms described above and have been influenced by the widest possible participation in the STEP support contractors' activities. While these recommendations do not represent a consensus of the data and opinions solicited, there are multiple sources of support for each recommendation. The recommendations have been organized as follows:

1. Recommendations to DDT&E for Modification of DoDD 5000.3.

2. Other Recommendations to DDT&E.

3. Recommendations addressed to the Military Services.

4. Tri-Service Recommendations.

The central areas of concern addressed by these recommendations include the following:

1. Test Planning: recommendations which relate to the planning and reporting of test activities and the setting of test goals and objectives for software.

2. Test Technology: recommendations which exploit or aim at developing the technical aspects of testing at each programmatic stage of development.

3. Test Tools and Environments: recommendations which relate to the development, qualification, packaging and distribution of automated tools for software T&E.

4. Test Evaluation: recommendations which relate to the translation of test results into quantitative statements of decision risk and software quality.

5. Technology Improvement: recommendations for support of development and basic research which have long-term benefits for software T&E.

6.1. MODIFICATION OF DODD 5000.3

The state-of-the-art in software testing has progressed to a stage of maturity in which systematic testing of software can be planned, documented, and evaluated. DoDD 5000.3 gives little specific guidance in this regard, and the inclusion of test issues which are peculiar to mission-critical software components is not common. Modifications to DoDD 5000.3 are clearly needed. The recommendations below can be incorporated into DoDD 5000.3 with few changes in the T&E process and with only slight modification of the Test and Evaluation Master Plan (TEMP). The effects of the modifications recommended here are to (1) establish a chain of T&E plans and evaluation criteria that begins at the level of system test objectives and proceeds through the detailed testing of software components within development organizations, (2) insert existing technology into the T&E process using software that represents the highest decision risk as the focus of the software test plan, and (3) establish the TEMP as the major planning document for software testing and ensure the early incorporation of software test issues into the overall test program.

DoDD 5000.3 (Section D, POLICIES AND RESPONSIBILITIES, Part 6 "Test and Evaluation of Computer Software") should be modified to include the following requirements:

a. Software components implementing critical functions shall be identified. These components shall be tested throughout the development/integration portion of the software lifecycle. Results of the tests shall be objective, repeatable, available to subsequent test groups, and interpretable in terms of overall system objectives.

b. The level of test of software components that implement critical functions shall be sufficient to demonstrate that the appropriate software evaluation criteria goals for that component are met or exceeded. The level of tests for these components should be sufficient to achieve a balanced risk with the hardware on which they are implemented in an operational environment.

DoDD 5000.3 (Enclosure 2) should be modified to require the incorporation of software-specific test and evaluation issues in the TEMP for systems with mission-critical software components. Deviations from the software-specific portion of the TEMP should be subjected to critical review. The portions of the TEMP which should include software-specific information are:

Part I - Description, 2. System, a. Key functions: Should also include a mission/function matrix relating the primary functional capabilities of each critical software component that must be demonstrated by testing to the mission(s) to be performed and concept(s) of operation.

Part I - Description: Should include a new section entitled Required Software Characteristics following Required Operational and Technical Characteristics (Items 3 and 4). This section should contain a list of the key software characteristics, goals, and thresholds.

Part I - Description, 5. Critical T&E Issues: Should include a new sub-section, c. Software Issues. This sub-section should briefly describe key software issues that must be addressed by testing.

Part II - Program Summary, 1. Management: Should also highlight arrangements between participants for software test data sharing.

Part II - Program Summary, 2. Integrated Schedule: Events to be displayed on the schedule should also include key software subsystem demonstrations and software testing tools availability dates.

Software T&E Outline: This new part should follow Part IV - OT&E Outline. This part should discuss all planned software T&E, for software components which implement critical functions, in similar format and detail as that described in the DT&E Outline (Part III). The Software T&E to Date section, which sets the stage for discussion of the planned software T&E, should summarize the software T&E already conducted and emphasize software events and results related to required software characteristics and critical software issues. This section and the Future Software T&E section should discuss the degree to which the test environment is representative of the expected operational environment. The section on Software T&E Objectives should present the major objectives that, when achieved, will demonstrate that the software development effort is progressing satisfactorily. The objectives either should be presented in terms of, or related to, the software characteristics. The Software T&E Events/Scope of Testing/Basic Scenarios section should relate the testing to be performed to the Software T&E Objectives. The Critical Software T&E Items section should highlight all items the availability of which are critical to the conduct of adequate software T&E prior to the next decision point. If appropriate, these critical items should be displayed on the Integrated Schedule. When the required software T&E information is contained in Parts III and/or IV, references may be made to those sections, as appropriate.

Part VI - Special Resource Summary, 1. Test Articles: Should also identify as test articles each software component that is identified in the mission/function matrix and key software subsystems shown in the Integrated Schedule.

Part VI - Special Resource Summary, 2. Special Support Requirements: Should also identify software test tools (including simulators) required, justify each tool identified (describe how the tool supports the software test objectives, achieves a specified level of test, etc.), and briefly describe the steps being taken to acquire each tool.

DoDD 5000.3 (Enclosure 1) should be modified to include the following terms and concepts:

Software Lifecycle. Extends from requirements definition and design through operation and maintenance.

Level of Test. Used in conjunction with a systematic software test methodology and is used to rank the thoroughness of a test with respect to the goals set for the evaluation criteria (e.g., 95% statement coverage vs. 50% statement coverage).

Software Evaluation Criteria. Standards by which achievement of required software characteristics, or resolution of software issues may be judged.

Required Software Characteristics. Software parameters that are primary indicators of conformance to written requirements/specifications and operational suitability and effectiveness.

6.2. Other Recommendations to DDT&E

Whereas the implementation of the recommendations presented above can be accomplished in the near-term with few changes in the T&E process and only slight modification of the TEMP, the next three recommendations address issues that cannot be resolved so easily. However, if we are to realize the full benefits of the modifications to DoDD 5000.3, these recommendations must also be implemented.

DDT&E should initiate or participate in an on-going program of software testing tools development, packaging, evaluation, distribution, and support to provide a warehouse, catalog, or test environment of approved testing tools which can be referenced in the software portion of a TEMP without acquisition or further approval. These tools should be accompanied by usage contexts which can be used to guide inclusion of tools in the software portion of a TEMP. These usage contexts should define the applications, programming languages, software evaluation criteria and level of test interpretations of the tool output.

Software development and T&E is not correlated with the decision milestones of the system acquisition process. Furthermore, software testing activities do not correlate well with the DT&E/OT&E/PAT&E division of responsibility. DDT&E should define a model of the software testing process which is well-integrated with the software development lifecycle. In the event that software T&E cannot be accomodated by the DT&E/OT&E/PAT&E structure, a separate software T&E program should be developed. Separate software T&E should not replace system DT&E/OT&E but should rather support DT&E/OT&E and serve to focus on the special software testing issues which are not adequately addressed by either DT&E or OT&E.

The relationship between required T&E on mission-critical components and (1) necessary testing on support software, (2) risk reduction on software that is required for system operation but does not directly implement mission-critical functions, (3) testing of other software design components is as yet unspecified. DDT&E should define software evaluation criteria for software in categories (1), (2), and (3). This definition should form the basis for a quantitative risk model of the software T&E process to be used in the evaluation of the overall software testing effort. Software T&E issues and activities required to evaluate the complete software system should be mentioned in the software portion of the TEMP.

6.3. RECOMMENDATIONS ADDRESSED TO THE MILITARY SERVICES

Implementation of the recommendations listed above requires a coordinated examination of software T&E technology and practice by DDT&E and the Military Services.

The following additional recommendations are intended to (1) implement DD&TE recommendations, (2) support DDT&E recommendations in areas that are not addressed directly by DoDD 5000.3, and (3) improve the software testing process. The arguments which support each recommendation appear interspersed with the recommendations themselves (which appear highlighted in upper case text).

6.3.1. STATE-OF-THE-ART IMPROVEMENTS

Software test and evaluation is an integral component of software technology. In particular, state-of-the-art improvements in software technology can be expected to improve software quality. New and enhanced software test methodologies should keep pace with advances in the field. In recent years, a number of DoD initiatives have been proposed with a view toward quantum improvements in the state of software engineering and practice. The focus of these efforts has been on design-oriented problems, standardization, and problem areas peculiar to the embedded and mission-critical computer application environments. The corresponding focus for software T&E improvements has not been as visible. As discussed in considerable detail in Volume 3 of this report, software testing issues are frequently given less than adequate treatment during the early stages of software project planning and specification. In planning for technological improvements in software engineering practice, early incorporation of initiative goals which directly address software T&E would serve two important purposes. First, it would provide for the improvement in test and evaluation technology needed to keep pace with advances in other areas. Second, it would be symbolic of the importance which adequate software test and evaluation has to overall software system quality; it would be a signal to the development communities that the test and evaluation of software for Defense systems is inseparably linked to the development process. MAJOR INITIATIVES TO IMPROVE SOFTWARE TECHNOLOGY SHOULD INCLUDE EARLY PROVISIONS FOR SOFTWARE TEST AND EVALUATION.

The DoD Software Technology for Adaptable, Reliable Systems (STARS) Program is the most recent such initiative - and the one which has received the most widespread attention from DoD, industrial, and academic sectors. At every front, the problems encountered in improving software test tech' iogy parallel the problems addressed by the STARS program. Problem in education, technology insertion, and leveraging existing R&D res urces and technology to improve the software development environme it each have their counterparts in the software testing.

Research initiatives should concentrate on enhancing the effectiveness of existing test techniques and on developing new techniques which address central problems in the test and evaluation of mission-critical software systems. THE SERVICES SHOULD CONTINUE RESEARCH FUNDING AT AN ACCELERATED PACE FOR SOFTWARE TEST AND EVALUATION METHODOLOGIES AND THE TOOLS TO SUPPORT THESE METHODOLOGIES. RESEARCH SHOULD ALSO CONCENTRATE ON ESTABLISHING USAGE CONTEXTS FOR THE METHODOLOGIES, COST/BENEFIT ANALYSES, AND EXPERIMENTAL DETERMINATION OF ERROR DETECTION CAPABILITIES.

Most existing and proposed methodologies are directly suited to automated testing tools. A MAJOR FOCUS OF MILITARY ORGANIZATIONS RESPONSIBLE FOR SOFTWARE DEVELOPMENT ENVIRONMENTS SHOULD BE THE IDENTIFICATION, QUALIFICATION, AND DISTRIBUTION OF TOOLS WHICH IMPLEMENT STATE-OF-THE-ART TESTING TECHNIQUES.

A major technology upgrade can result from the continued development of a new generation of higher order languages (HOL's) suitable for use in embedded computer applications. It is a widespread opinion that HOL development and standardization is not proceeding fast enough. This opinion is supported by the number and variety of programming languages encountered in the studies reported in Volumes 2 and 3 of this report. Many languages in use on major systems are tailored to specific hardware requirements. Others are "standard" for a class of systems and applications, but are used almost nowhere else. Language levels range from absolute non-relocatable machine code to HOL's such as Fortran and JOVIAL. Programming environments are similarly diverse. They range from environments that include little more than compilers and linking loaders to UNIX (trademark of Western Electric) environments with an array of programming support tools. Tools and support systems generated in one project are seldom transported to other projects, even within the same development organization.

There is considerable faith in the development communities, project offices, and test organizations that Ada will be the catalyst for broad improvements stemming from a HOL upgrade. In many quarters, Ada is seen to offer improvements at the language level through its support of modern design methodologies. The development of Ada programming environments and run time environments offers the opportunity for integrating test, simulation, and management techniques. It may also ease the technology transfer and insertion problems and make successful support tools available to a broad range of Ada user environments, resulting in standardized tools and techniques for certain classes of programming tasks. The development of tools for testing, simulation and system integration and the definition of relationships between development environments and run time environments have been neglected in current planning. Existing plans for Ada Programming Support Environments (APSE's) contain vague provisions for "debugging" tools and systems, suggestions for inclusion of data gathering and automatic metrics calculation facilities, and support of simulations of run time environments at the development level. Detailed plans for substantial support subsystems to aid in T&E have not been forthcoming. Existing APSE development efforts do not address T&E issues at all, and AJPO does not project the inclusion of such facilities in the first generation of APSE's. It is unlikely that usable test support environments will be available until well after Ada adoption by the Services. AJPO AND THE AFFECTED MILITARY SERVICES SHOULD BEGIN NOW TO MODIFY AND EXPAND APSE DEVELOPMENT PLANS TO INCLUDE SUBSTANTIAL PROVISIONS FOR TEST SUPPORT ENVIRONMENTS. TEST SUPPORT TOOLS SHOULD BE MADE AVAILABLE IN THE FIRST GENERATION OF APSE'S THAT ARE USED TO DEVELOP SOFTWARE-INTENSIVE SYSTEMS.

6.3.2. LIFECYCLE INTEGRATION

The most costly software errors are those that are created early and discovered late. Therefore, quality assessments of software should be performed during all phases of the software lifecycle. As yet, there is no acceptable model for integrating software test activities into the various phases of software development. During the earliest stages of software development, test teams will have access to the software that is not possible during later stages of development. Systematic attempts should be made to identify errors while the software is still available for direct manipulation.

As we have already noted, the division of software testing activities into the standard DT&E/OT&E/PAT&E framework is at best an artificial one. The correspondence between critical stages of software development and programmatic milestones will certainly be clarified with further study. In the meantime, however, the Services should refine and investigate approaches to software T&E. Techniques should be identified and practices established to allow assessments of risk due to critical software components that are useful predictors of overall software system reliability and effectiveness. Such problems can be approached without assuming any particular lifecycle testing model by concentrating on the following aspects of the T&E process: (1) test planning, (2) testing, (3) test evaluation and risk assessment.

6.3.3. TEST PLANNING

The effectiveness of software T&E depends in large measure on thorough test planning and adherence to written test plans. In the development of software-intensive Defense systems, software testing is frequently the victim of inadequate planning. When programs do not include specific plans for the test and evaluation of software, program offices, independent test organizations and contractors are left to interpret the goals and objectives of software-oriented tests. Although test plans and procedures exert a strong influence on the quality of the overall software test program, a number of factors combine to make detailed planning for and evaluation of software tests especially difficult. One factor is the perception in acquisition, development, and test communities that T&E is given inadequate time and dollar resources. While other phases of system development also feel squeezed by schedules and budgets, testing is often viewed as the phase which is most compressible. One reason for this perception is the nature of testing; since testing examines the product it is often viewed as not contributing to the development. Another reason is that testing activities usually fall toward the end of the design and implementation effort. This part of the problem can be approached through the acquisition cycle. Acquisition agencies should ensure that the time and money allocated to test and evaluation are protected. This may be especially difficult in light of current methods of allocating resources to the overall system development. However, in recent policy memos to the Services, OSD has directed that certain test resources be protected; these concepts should be widened to include test resources for software T&E.

Another contributing factor is the that test planners often use optimistic assumptions about the nature and extent of the software testing effort. Slippage in schedules or budgetary shortfalls are therefore not accommodated gracefully. This problem can, however, be solved by planning according to more realistic or worst-case scenarios.

Finally, another significant problem is that written software test plans are seldom required until software system integration. Yet the most productive software testing typically occurs during development and integration. Furthermore, the test results from the earliest stages of software system development are often the most essential element in evaluating the progress of the testing effort. These tests should be guided by a software test plan which spells out test criteria and objectives and sets goals for software T&E.

The test approaches ultimately adopted should relate to and support the objectives of the TEMP. If the TEMP specifies a fixed set of software test objectives then the results of early testing should lead to evaluations in terms of these objectives. By the same token, if the TEMP provides for test criteria, environments and tools that can be applied during early software testing, the requirements of the TEMP will determine the technical nature of the early software tests. Allowing development teams to improvise in the choice of test approaches, to change the objectives of early tests, and to conduct tests without reasonable accountability, frustrates the overall test program. Therefore, PROGRAM OFFICES SHOULD ENCOURAGE AND SUPPORT THE DEVELOPMENT OF WRITTEN TEST PLANS FOR TESTS TO BE CONDUCTED DURING EARLY PHASES OF SOFTWARE DEVELOPMENT. THESE PLANS SHOULD (1) CONTAIN A SPECIFICATION OF WHAT CONSTITUTES AN ACCEPTABLE APPROACH TO TESTING,

(2) EXPLAIN HOW THE APPROACH ADOPTED SUPPORTS OBJECTIVES OF THE HIGHER LEVEL TESTS, (3) BE ADHERED TO RIGOROUSLY BY PROGRAM MANAGERS, (4) BE CRITICALLY REVIEWED FOR DEFICIENCIES, AND (5) REFLECT A REALISTIC, WORST-CASE ESTIMATE OF THE SCOPE AND EXTENT OF THE REQUIRED TESTING EFFORT.

An important implication of assessing quality throughout the software lifecycle is that early testing during software design and implementation is critical. The earliest software testing activities occur during unit and module testing. It is common practice for the development team to conduct unit and module tests according to internal standards and procedures - frequently, under the direct and exclusive control of the programmers. Since these tests are seldom conducted under the guidance of a written test plan, the objective of unit and module tests may be to "get by" rather than to uncover errors. In the typical waterfall lifecycle, there is little upward flow of test information from the development groups to later test teams. By the same token, there are few audit trails which can be applied to evaluate the effectiveness and progress of these tests. The accountability is hampered by a lack of emphasis on documentation during unit and module tests. The results of early testing can be a valuable resource to later testing phases and should constitute the initial component of a test support data base which contains results of software testing activites. Since the results of these testing phases should be interpretable in terms of the TEMP and written test plans for early software testing, PROJECT OFFICES SHOULD REQUIRE DOCUMENTATION OF UNIT AND MODULE TESTS. DOCUMENTATION REQUIREMENTS SHOULD INCLUDE RESOURCE REQUIREMENTS, SIMULATION REQUIREMENTS FOR INPUTS, ANALYSIS REQUIREMENTS FOR OUTPUTS, TEST CASE CROSS REFERENCES TO SYSTEM REQUIREMENTS AND SUFFICIENT SUPPORTING INFORMATION TO ALLOW THE RECONSTRUCTION AND REPETITION OF TESTS.

Another implication of testing throughout the software lifecycle is that during later project milestones, much of the software is actually in its maintenance phase. Therefore, software testing during later phases of system development, integration and test is actually retesting or regression testing of the software. PROJECT OFFICES SHOULD ENSURE THAT PROVISIONS ARE MADE FOR REGRESSION TESTING IN ALL TEST PLANS. IN THE ABSENCE OF A MAJOR IMPROVEMENT IN THE STATE-OF-THE-ART IN REGRESSION TESTING, AUDITING AND RETESTING PROCEDURES FOR ALL SOFTWARE, SPECIFICATION, AND REQUIREMENTS MODIFICATIONS AND UPDATES SHOULD BE REQUIRED.

In some cases, independent verification and validation (IV&V) organizations are involved in the test and evaluation effort. WHEN IV&V IS REQUIRED BY THE PROJECT OFFICE, THE INVOLVEMENT OF THE IV&V CONTRACTORS SHOULD BE PLANNED AND INTEGRATED INTO THE OVERALL TESTING EFFORT. Effective IV&V efforts require the validation of user needs and verification of the design and implementation. This is turn requires that the technical focus of the IV&V be incorporated into test plans and that an efficient transferral of project information takes place. PROJECT OFFICES SHOULD ENSURE THAT TEST PLANS CONTAIN PROVISIONS FOR IV&V INVOLVEMENT.

6.3.4. THE TESTING PROCESS

Test planning should be conducted with a view toward achieving technical goals. Technical goals are in turn set by combining software test objectives with a balanced and informed evaluation of the state-of-the-art. The following findings and recommendations are addressed to these issues. In order to plan the testing effort and evaluate results, it is necessary to decide what software is to be tested, the test technology to be applied, and the degree of automation desired in the test.

The recommendations for modification of DoDD 5000.3 include provisions for system level test planning for software that implements one or more mission-critical functions. Therefore, in a typical TEMP, non-critical software components will not be addressed at all. However, test plans should address the testing effort for the total software system.

In addition to software components specified in a TEMP there may be other components of an embedded software system that are transparent to the critical system functions. An example of such a component is a system monitor which is responsible for achieving timely response to operator commands. Such components are necessary to correct and reliable system operation but are seldom tested against overall system requirements. There are also components that are necessary to meet certain operational goals. For instance, logistic support may require the availability of support software (e.g., a specialized compiler or telecommunications package). In such cases, the operational availability of the mission-critical software depends on the correctness and reliability of software that is not strictly part of the system. When such components are tested at all, they are usually tested as "black boxes" which deliver services to critical functions. PROJECT OFFICES SHOULD SET GOALS FOR THE TESTING OF THE TOTAL SOFTWARE SYSTEM, INCLUDING THOSE COMPONENTS NOT SPECIFIED IN THE TEMP. THESE GOALS SHOULD BE INCORPORATED INTO A WRITTEN TEST PLAN AS A SET OF SOFTWARE TEST OBJECTIVES. THE NATURE AND EXTENT OF THE TESTING REQUIRED FOR THESE COMPONENTS SHOULD BE SUFFICIENT TO ACHIEVE A BALANCED RISK WITH MISSION-CRITICAL COMPONENTS.

Development test and evaluation (DT&E) tests system components against requirements and specifications. Operational test and evaluation (OT&E) tests overall system capabilities. Since the development cycle for software spans the DT&E/OT&E activities in unexpected ways, the relationship between DT&E and OT&E is especially critical. Since they are both testing activities, DT&E and OT&E are subject to many of the same pressures. DT&E is an essential prerequisite to OT&E. Without careful contingency planning, slippages, failures and inadequacies in DT&E can negatively affect OT&E. Test environments for DT&E and OT&E must be coordinated. During DT&E operational environments are generally simulated and the use of prototype systems is acceptable. During OT&E, simulated run time environments are sometimes used, but the nature of an operational test requires that the simulated environment be faithful to the characteristics of the actual environment. Even though concurrent DT&E and OT&E is sometimes conducted, operational test groups are sensitive to the fact that software problems left unresolved during DT&E can be masked or untraceable during OT&E. DEVELOPMENT TEST ORGANIZATIONS SHOULD RESOLVE MAJOR SOFTWARE DEFICIENCIES BEFORE THE START OF DEDICATED OT&E.

OT&E does not evaluate the software by itself but rather as part of the total system. In OT&E, the focus is on operational characteristics, and test scenarios are designed against operational test objectives. It is important to note that operational tests are user-oriented and operational failures may be decided by users. This is contrasted with DT&E; during DT&E, failures are decided by conformance to requirements and specifications. Nevertheless, software components may present special problems during operational testing, and efforts to ensure that software is thoroughly tested in an operational environment may be desirable. Early OT&E involvement in software test planning, carefully drawn test plans, test management which avoids deviation from plans, simulated hardware failures and tests of user interfaces can all be accommodated within existing test organizations if adequate resources are made available. As a further step toward isolating software errors that are incompatible with actual operating conditions, OT&E organizations could adopt alternative approaches which would essentially operationally test software subsystems in simulated operational environments. Evolutionary development, rapid prototyping, and build-test-build lifecycles have been proposed as techniques for ensuring early involvement of OT&E in the development process. Furthermore, operational tests can reveal the presence of design errors and inefficiences that can form the basis for the redesign and modification of software. Therefore, OPERATIONAL TEST DOCUMENTATION AND RESULTS SHOULD BE AN INTEGRAL PART OF THE OVERALL SOFTWARE TEST DATABASE.

Major improvements in the testing process can be achieved by addressing the early phases of development. This is critical since detailed software testing does not occur throughout the total system development effort for most embedded applications. Special problems arising from software errors that may be masked after system integration should be identified and addressed. Therefore, extensive testing at the unit and module levels must occur before system integration.

Detailed test plans for all critical software components should include provisions for testing at the unit and module levels. Unit and module tests should be complete enough to ensure that critical functions have been exercised. At the unit and module levels, exercising critical functions is complicated along two dimensions. First, the test methodology must be sensitive to the system-critical errors that are most likely to be encountered in operation. Second, units and modules must be "driven" by simulated test environments since actual operational inputs are probably not available until after system integration. The expense incurred in these tests should be balanced with the criticality of the components being tested. However, the record of software failures during system integration, operational testing, and post-deployment operations is filled with examples of expensive errors that would have been uncovered by almost any systematic test strategy applied at the unit or module level. PROJECT OFFICES SHOULD ENSURE THAT UNIT AND MODULE TESTS EXERCISE CRITICAL FUNCTIONS WITH A SYSTEMATIC TEST METHODOLOGY. IN SELECTING A TEST METHODOLOGY, PRIMARY CONSIDERATIONS SHOULD BE THE APPROPRIATENESS OF THE METHODOLOGY, KNOWN COST/BENEFIT RATIOS, ESTABLISHED ERROR DETECTION CAPABILITIES OF THE METHODOLOGY, AND THE EXTENT TO WHICH TEST RESULTS ARE INTERPRETABLE IN TERMS OF SOFTWARE TEST OBJECTIVES SET FORTH IN THE TEMP.

The issue of appropriateness of the test to the current test objective is an especially important one - it is often this issue alone that determines the test methodology. For example, if the purpose of a test is to determine the behavior of a system under stress or heavy load conditions, it is essential that the softwaare test produce extreme values, boundary values, and special values for affected parameters. Random testing is, therfore, not appropriate to that test objective. In many instances, appropriateness of a test depends on the application. Furthermore, neglecting the specialized needs of an application may result in extreme test requirements that do little to meet test objectives. For example, MIL-STD-1679 requires endurance testing of software. Endurance testing is motivated by system reliability requirements, especially when operational reliability estimates are stated in time dependent form. Mathematical formulations of these estimates are derived from material failures, and their applicability to software failures is unclear. It should be determined whether endurance testing of software is necessary to test functionality or whether shorter tests of greater variety can be substituted without sacrificing test quality. THE RELATIONSHIP BETWEEN TESTS PERFORMED AND THE ERRORS TO BE DISCOVERED MUST BE EXPLICIT IN THE TEST METHODOLOGY. THIS RELATIONSHIP SHOULD BE A PRINCIPAL CONSIDERATION IN DETERMINING THE APPROPRIATENESS OF THE TEST.

A number of formal testing methodologies which are potentially valuable in unit and module testing are not applied in practice. The most common complaint about systematic test methodologies is that they do not concentrate on the system objectives and errors which are critical. For instance, a test methodology which requires statement coverage (execution of all statements) with little or no regard for the cost/benefit aspects of the tests is viewed with suspicion by a number of development groups. This situation is most apparent at later stages of testing, particularly software integration testing and regression testing. In these areas, there is relatively little published documentation on the effectiveness and cost of techniques.

Most approaches to integration and regression testing are based on experience with similar applications. Principles to support evaluations of the tests are not well-established, and test approaches encourage improvisation. Integration testing is hampered by the lack of an underlying theory of functional testing that treats the total system. Military research organizations should encourage software testing research that offers near-term solutions to this problem. Regression testing occurs at all stages of the software development process, and is widely recognized to be inadequately treated by current methodologies and practices. Regression testing is frequently expensive, labor-intensive, and is not easily evaluated. Regression errors are common and difficult to detect. A major effort is necessary to develop an effective regression testing methodology.

Part of the problem with developing test methodologies that are applicable above the unit and module level is that the current emphasis in systematic testing methodologies is on testing for correctness. In embedded computer applications operational reliability, performance, conformance to user requirements, robustness, fault-tolerance, and the ability to respond to real-time inputs are frequently more important than correctness. The state-of-the-art in this area should be advanced. THE MILITARY SERVICES SHOULD ENCOURAGE AND SUPPORT THE DEVELOPMENT OF TESTING TECHNIQUES THAT TAKE INTO ACCOUNT QUALITY MEASUREMENTS OTHER THAN CORRECTNESS.

Virtually all systematic software test techniques require considerable computational resources. Furthermore, automated data manipulation is required for archiving and managing test results and documentation. The goal of rigorous software test and evaluation is only attainable with an array of testing tools appropriate to the test methodology and test reporting set forth in the test plan. The most obvious barrier to applying state-of-the-art technology in development testing is the lack of usable testing tools and environments. Lack of critical tools is also a contributing factor to many of the inadequacies in integration testing, quality assurance, data gathering and operational testing. Of the tools surveyed in Volume 2 of this report, only a small fraction are marketed, supported package software tools. Testing tools are typically developed in university or research settings or as support tools for specific software projects. Research tools generally enter the public domain and are in principle available to all interested users. However, these tools are not supported, are poorly documented, present poor user interfaces and are not easily transportable. Tools developed in support of other software development efforts usually do not become available to the general public. Some organizations consider testing tools developed in this way to be proprietary software for use in support of internal testing standards and practices, but not to be marketed. In other organizations, the tools are under the control of a programming group (usually, an individual). When the project disappears, the incentive for generalizing, documenting, and distributing the tools also disappears. In gathering the data for Volume 2, the most common situation was that the person responsible for a tool had either been reassigned or had left the organization; in either case it was rare for a tool to be supported by new personnel.

There is also a widespread opinion that existing tools (even when they are available) are not likely to be suitable to a given development testing effort. Unsuitability can arise in a variety of ways.

1. The candidate tools may not match the development environment (e.g., the source language supported by the tools is not the source language of the intended project).

2. The tool may implement a testing technique or strategy which matches neither the internal test procedures of the developer nor the test procedures set forth in the test plan for the project.

3. The tool may be inconvenient to use. That is, the extra overhead involved in invoking and using the tool may be greater than the perceived loss of effectiveness in using no tool at all.

4. The tool may be so uncomfortable that test teams and programmers will not use it under any circumstances.

5. The tool may force accountability and visibility to the development testing process that the developer wants to avoid.

6. The use of tools may not be justifiable to project and contract managers. Tool usage involves an investment of time and personnel on the part of the developer, and it is usually not possible to justify the investment on economic grounds.

7. The development organization may delegate development level testing to the development teams. If development teams consider testing to be part and parcel of program development and debugging, they will find any tool unsuitable since there is no systematic testing taking place at all.

(5) - (7) address deeper issues than the availability of appropriate tools. (1) - (4) can either be addressed by the Services through their development labs or by tri-service cooperation through STARS or other umbrella programs. IMPLEMENTATION OF EFFECTIVE PRACTICES FOR SOFTWARE T&E REQUIRE THE MILITARY SERVICES TO INITIATE ON-GOING PROGRAMS TO DEVELOP, PACKAGE, EVALUATE AND MAINTAIN TESTING TOOLS. INCLUDED IN THIS EFFORT SHOULD BE A PROGRAM TO IDENTIFY AND QUALIFY TOOLS FOR EARLY USE IN THE DEVELOPMENT CYCLE.

The qualification problem for testing tools is especially severe. Development groups will need guidance in selecting tools which are suited to the testing task at hand. Furthermore, for efficient implementation of TEMP requirements (e.g., for tools referred to in the test articles portion of the TEMP), qualified tools should be warehoused in an effective manner. There are essentially two routes that a test team can take in selecting a tool to meet test plan requirements. The first is to select a warehoused tool that has already been qualified for use; the justification for tool usage then reduces to justifying the appropriateness of the tool.

The second route is to use a tool not in the warehouse. Not only must the appropriateness of the tool be established, but a separate justification for tool selection must take place which establishes the technical characteristics of the tool and presents data to support all technical claims. The second route is clearly the more expensive one. The only reason a developer would choose it is that he has a proprietary testing tool. In such instances, the developer has commercial disincentives for not offering the tool for inclusion in the warehouse. Since he chooses this route for commercial reasons, he should carry the financial burden of qualifying the tool. In most foreseeable instances, the developer will not need to recreate his qualification data for each inclusion of the tool, so the incremental cost of choosing the second route when averaged over a number of contracts is likely to be small. THE QUALIFICATION REQUIREMENTS SHOULD SPECIFY USAGE CONTEXTS FOR SPECIFIC TOOLS AND COMPARATIVE ANALYSES OF COSTS AND EFFECTIVENESS OF INDIVIDUAL TOOLS SHOULD BE PROVIDED.

Implementation of such a program could have the adverse effect of freezing the technology unless mechanisms are introduced for inserting promising new technologies into the warehouse and phasing out obsolete technology and tools that are superceded by more advanced ones. There are likely to be two main sources for this new technology. Research organizations will contribute prototype designs for qualification and packaging. Development organizations can also be expected to produce tools that genuinely advance the state-of-the-art. As noted above, however, these are usually tools tailored to a parent project. To be useful in this setting, those tools must also be generalized to apply to a variety of applications, environments, and test groups. PROVISIONS SHOULD BE MADE FOR GENERALIZING AND IMPROVING TOOLS WHICH IMPLEMENT STATE-OF-THE-ART TEST TECHNIQUES AND STRATEGIES.

To address the problem of generalizing and distributing tools, there needs to be a reevaluation of tool production which recognizes the high cost of tool development and marketing. THE MILITARY SERVICES THROUGH THEIR PROJECT OFFICES SHOULD CONSIDER REEVALUATING CONTRACT FUNDING PATTERNS TO ALLOW SPECIAL PURPOSE TOOLS DEVELOPED IN SUPPORT OF THE CONTRACT DELIVERABLES TO BECOME DELIVERABLE ITEMS UNDER THE SAME CONTRACT. This will initially require additional government investments in software contracts, although the incremental cost of tool development should decrease in time as the available library of testing tools becomes rich enough to support the range of software technology projected for the next decade. Additional consideration should also be given to integrating testing tools into general support environments. Proliferation at both the hardware and software levels have become severe problems for all of the Military Services. In addition to cost increases, difficulties in training personnel, and the obvious transportability problems, proliferation impacts such critical areas as logistic support and system availability.

Many of these problems are shared by other key aspects of software development for mission-critical applications. The issues of technology insertion and integration with standard support environments are key aspects of the STARS program, and an effective long range solution to the problem of identifying, packaging, qualifying, distributing and supporting test tools may well be handled through STARS. DDT&E IN COORDINATION WITH THE MANAGEMENT OF STARS AND THE MILITARY SERVICES SHOULD INVESTIGATE THE POSSIBILITY OF INCLUDING THE SOFTWARE T&E TOOLS WAREHOUSE IN ONE OR MORE STARS TASK AREAS. IN PARTICULAR, THE PROCESS OF IDENTIFYING, PACKAGING, QUALIFYING AND DISTRIBUTING TEST TOOLS FOR USE IN SUPPORT OF TEST PLAN REQUIREMENTS SHOULD BE A KEY ROLE FOR THE STARS SOFTWARE ENGINEERING INSTITUTE OR ITS FUNCTIONAL EQUIVALENT.

6.3.5. TEST EVALUATION

Test and evaluation in the Department of Defense is used to support the acquisition of systems designated for operational use, identifying and reducing risk, and assessing the operational potential and reliability of those systems. DoD decision makers and system developers use the results of tests at various levels to formulate estimates of risk and opinions about progress and probable operational characteristics as programs proceed through the acquisition milestones. Thus, the problem of evaluating test results is an especially critical one for DDT&E and the acquisition communities at large. The results of this study indicate that major improvements are needed to allow decision making for software-intensive acquisitions to bal ance hardware and software issues. The following findings and recommendations address the areas of the evaluation process which are most significant for software T&E and in which the most improvement can be gained.

In practice, the decision to proceed from one stage of testing to the next is frequently based on externally imposed schedule milestones rather than test status, even though current policy states that transitions should be based on accountable completion criteria. The primary guidance given to the Military Services with respect to software test and evaluation resides in DoDD 5000.3. For example, DoDD 5000.3 states that, "Decisions to proceed from one phase of software development to the next will be based on quantitative demonstration of adequate software performance through appropriate T&E". The level of software testing conducted during a previous phase should form the basis of the accountable completion criteria. The overall effect of current practice is to make the critical programmatic decisions more subjective than is desirable, supporting neither the quantitative measures of testing progress nor the quantitative demonstrations required by DoDD 5000.3.

An area in which significant improvement in current practice can be gained is the assessment of risk. The risks involved in the development and deployment of mission-critical and embedded computer systems can be associated with system production, the success of the mission, and the consequences of system failure. When testing is used as a risk reducing activity for software, the purpose of a test is to uncover software errors or to build the testers' confidence that no errors of a given type remain to be uncovered. As we have remarked elsewhere, there currently exists a wide variety of testing strategies and techniques from which test planners can choose. However, there is little evidence available as to which test approaches are most effective in a given situation. In addition, the decision as to how much testing is enough is of necessity a subjective one.

One approach to test evaluation is economic. If the cost of testing to find residual software errors exceeds the cost incurred if the errors occur, then further testing is clearly not cost effective. The real problem with justification on narrow economic grounds is that the concept of error cost takes into account undesirable events which cannot be easily associated with dollar costs.

Another approach is to quantify the elements of risk in the software system. It is widely recognized that not all software errors are of equal significance. The occurence of certain errors (e.g., in a function controlling the release mechanism for a nuclear weapon) have more serious implications than others (e.g., misspelled words on a user display). Test requirements should take these observations into account and attempt to achieve a balance between the extent and cost of a test and the criticality of failure. Testing requirements for software, as well as other software development requirements are passed to developers by military standards and specifications. Currently, MIL-STD-1679 provides the most detailed requirements relative to software development and testing. In the view of many developers, however, MIL-STD-1679 levies extreme test requirements without regard to the impact of potential software failures. If testing is to be proportional to the cost of failure, a risk analysis should be performed on potential software errors.

THE MILITARY SERVICES AND DDT&E SHOULD DEVELOP QUANTITATIVE INDICES OF SOFTWARE TESTING PROGRESS DURING DEVELOPMENT. QUANTIFICATION SHOULD TREAT BOTH COSTS AND RISK: (1) RELIABLE COST/BENEFIT MEASURES FOR TESTING SOFTWARE SHOULD BE DEVELOPED AND THE COST/EFFECTIVENESS OF TESTING TOOLS SHOULD BE ESTABLISHED. (2) QUANTITATIVE RISK ANALYSIS TECHNIQUES FOR SOFTWARE ERRORS SHOULD BE DEVELOPED. (3) COST AND RISK SHOULD BE USED AS ESSENTIAL FACTORS IN DETERMINING QUANTITATIVE INDICES.

Ultimately, the most usable source of quantitative information about software quality is error data collected from current projects. The observation, classification, and analysis of error data can be of considerable help in planning the cost and effectiveness of tests. In addition, expected improvements in the state-of-the-art in software T&E will certainly require more exact classification of error data than is currently available. It will be necessary to improve the data gathering procedures considerably to support error and defect discovery rates. Furthermore, the Services should develop a mechanism for the careful review of significant (i.e., mission-threatening) software failures, determining error distributions for errors discovered in requirements documents, and categorizing design errors.

Software quality assessments during critical early program phases require the measurement of software characteristics and processes that affect quality. Measurements and data gathering procedures that support reliable quality assessments are rarely applied and expensive to develop. For near-term support of quantitative techniques, the calculation of metrics based on currently identified software quality factors is needed. However, existing metrics have not been throughly validated as predictors of software quality. They should be applied only when supporting error data and documentation are used for cross-validation and redundancy.

MILITARY LABS SHOULD EXPAND THEIR EFFORTS TO PROVIDE AN IMPROVED DATA GATHERING, REDUCTION, AND MEASUREMENT CAPABILITY TO PROJECT OFFICES AND DEVELOPERS. AUTOMATED DATA LOGGING AND DATA BASE SYSTEMS SHOULD BE DEVELOPED TO TRACK AND RECORD ERRORS ON SOFTWARE-INTENSIVE SYSTEMS. THE RELATIONSHIP BETWEEN MEASURABLE CHARACTERISTICS OF SOFTWARE PRODUCTS AND THE PROCESSES USED TO PRODUCE THEM SHOULD BE VALIDATED. MEASURABLE CHARACTERISTICS WHICH ARE RELIABLE PREDICTORS OF SOFTWARE QUALITY SHOULD BE APPLIED TO ENHANCE THE EVALUATION PROCESS.

Quality assurance (QA) is concerned with evaluating the process of software development. Experience has suggested that independent QA organizations are helpful, but they are by no means commonplace. In order to be effective, QA must have scheduled, budgeted and planned involvement throughout the development process. Written audit reports must be provided. As with all transitions between phases of the testing effort, the lack of effective flow of materials and information is a major roadblock to effective QA. Plans, scripts, and results of previous testing phases should be made available to QA organizations. MIL-S-52779A describes the general concepts for software quality assurance but gives few specific requirements. AN EFFECTIVE SOFTWARE QUALITY ASSURANCE STANDARD SHOULD BE DEVELOPED.

Several organizations have found that independent verification and validation (IV&V) teams can supply significant support to project offices. Air Force policy requires that IV&V be considered for use on all software-intensive systems. However, opinions on the value of IV&V are divided and many consider the technique to be needlessly expensive. The best counterbalance to costly IV&V activities is likely to be a sharp focus on validation of user needs and verification of the design and implementation. In particular, there is no need for IV&V tasks focus on the development process. When IV&V is indicated, special efforts should be undertaken to ensure that the IV&V effort is an independent V&V effort. IV&V cannot be effective unless there is an overt atmosphere of objectivity in the IV&V organization. THE MILITARY SERVICES SHOULD DETERMINE THE COST/BENEFIT ASPECTS OF IV&V AND RECOMMEND THE CONDITIONS UNDER WHICH IV&V SHOULD BE REQUIRED.

A problem common to all of the above areas - and a problem noted in several independent studies of program assessments at production decision points - is the inadequacy of information flow during the development cycle. Ineffective or untimely communication between development organizations, test organizations, project offices and other DoD management structures and the relative absence of essential T&E information in advance of major program decision points results in schedule restrictions, resource allocations not matched to the scheduled tasks, and higher decision risks. When quantifiable T&E characteristics are not made available well in advance of decisions and used as a basis for reviews and resource adjustments, expensive corrective actions are the only alternative. DDT&E AND THE SERVICE PROGRAM OFFICES SHOULD BEGIN NOW TO DEVELOP AN INTEGRATED DECISION SUPPORT SYSTEM FOR SOFTWARE T&E THAT COMBINES FUNCTIONALLY ORGANIZED TEST INFORMATION AND EVALUATIONS WITH DATA THAT IS REQUIRED FOR MAJOR PROGRAMMATIC DECISION POINTS.

6.4. TRI-SERVICE RECOMMENDATIONS

Many of the previous findings and recommendations encourage Service participation in formulating policy, guidelines, and standards for use by development, management, and test organizations. Response to DoD and Service actions in implementing these recommendations will certainly require additional investments on the part of contractors. It is important that further investments in software T&E be channeled into those areas in which improvements are likely to have the most effect. In the current T&E environment, military standards and specifications which are applied on contracts vary from Service to Service. Contractors who do business with two or more Services cannot, in general, develop a unified approach to software T&E.

In the recommended environment in which early software testing is planned and monitored, tools are qualified for project use, test evaluation is made more objective, uncoordinated requirements by the Services could result in undesirable splintering of resources by the development communities. Furthermore, uncoordinated responses by the Services could irritate an already noticeable problem: the proliferation of development and support software environments. There is a clear danger that the existing technology for implementing improved software T&E will be interpreted differently by each of the Services resulting in policies, standards, and test environments that are tailored to each Service. It is likely that this would lead to the same problems that have arisen in the development of HOL's and hardware, e.g., increased development costs, difficult logistic support, problems in transportability, and overall decreased performance and reliability.

A more desirable approach is to allow development organizations to attack the software quality problem in a unified way. This can be accomplished only by extensive cooperation on the part of the Services in developing a common approach to software T&E. THE MILITARY SERVICES SHOULD DEVELOP TRI-SERVICE STANDARDS TO MAKE UNIFIED APPROACHES TO SOFTWARE DEVELOPMENT, TESTING AND EVALUATION POSSIBLE.

CHAPTER 7

CONCLUDING REMARKS

Revision of DoDD 5000.3 and the attendant modifications to more specific regulations and standards will have a significant impact on the quality of Defense system software. However, the usefulness of new guidance in software T&E will be mediated by how rapidly the research, development and acquisition communities move toward state-of-the-art application of existing technology. One of the most significant needs is support for tool development. This may involve modifying contract funding patterns, and may initially increase project costs. However, there seems to be a consensus that testing cannot be justified on narrow economic grounds. Total lifecycle costs must be taken into account. Along the same lines, incentives must be provided for improved testing throughout the development/integration portion of the lifecycle. This may require revisions of the development process.

New guidance and regulations must also be realistic. If developers and testers find themselves too constrained by regulations, they will not have the desired effect. It has been noted, for example, that not all software components are created equal: some implement critical functions and others do not. To require the same level of testing and, therefore, the same resources for all components is probably not realistic and may actually serve to reduce the effectiveness of tests in critical components.

Software developers and requirements writers must eventually strike an accord. On one hand, development groups should recognize that neither requirements nor specifications are likely to remain static - they must learn to cope with change. On the other hand, those who formulate requirements cannot assume that software is arbitrarily malleable: software changes may be as expenseive and far-reaching as changes to any other system component. System retests and budget/schedule shortages are currently victims of the tension between requirements and development groups.

Finally, basic research is needed. There is no quantitative risk model for software. Software measurement techniques are still at an early stage of development so that objective data is still only a goal. Testing techniques, methodologies and tools need further development. The cost-quality tradeoffs for various techniques must be quantified if developers and testers are to make a choice from among the available techniques.

It is certainly feasible to formulate new DoD guidance for software T&E. New guidance must address the most pressing problems, either directly or indirectly, by encouraging new technology and acquisition procedures. With such encouragement, the technological "window" will move to provide more effective techniques for software T&E. New guidance should be general; development testers and operational test groups should not feel bound by mandated test procedures that fit neither their application nor their environment. The exact form that such guidance will take and its ultimate effect on the reliability of future military systems awaits further study.

APPENDIX

PROJECT ORGANIZATION AND MILESTONES

The prime contractor for the support contract was the Georgia Institute of Technology. Tasked with overall responsibility for the information gathering and analysis phases of STEP, a study team from the School of Information and Computer Science concentrated its efforts on the development of an overview of the state-of-the-art and projected technology in software T&E. The Georgia Tech team also provided management and project support for several other activities. Dr. Richard A. DeMillo served as director of the Georgia Tech team as well as Principal Investigator for the prime contract. The task of compiling an overview of the state of military and industrial practice in software T&E was executed by a study team from Control Data's Atlanta Research Facility under a subcontract from Georgia Tech. The Control Data study team was under the direction of Project Manager, R.J. Martin. Control Data's team was also tasked with overall responsibility for retaining consultants and managing report preparation and distribution. A third subcontract was let to Clemson University. This project, under the supervision of Dr. James F. Leathrum, was designed to provide a study of a particular tactical computer system for the U.S. Army. The results of this subcontract demonstrated the value of early testing and modelling procedures for a typical military acquisition effort.

In carrying out the tasks for Phases I and II, a key consideration has been the breadth of input from academic, military and industrial sectors. Wide participation was important for gathering information which would simultaneously: (1) give a balanced overview of current capabilities and practices, (2) ensure that expert opinions from Defense circles most involved with software development were included in the assessments, and (3) solicit suggestions for improvements in current T&E practices and policies. A secondary benefit of wide participation was its validating effect: the study teams found widespread agreement on the nature of the fundamental problems to be resolved along the way to an adequate policy for software T&E. The various mechanisms used to solicit this participation are described below.

PHASE I MILESTONES

The major milestone tasks for Phase I of the STEP support contract were the following:

1. Orientation Workshop: In March of 1982, a workshop for selected military, DoD, and industrial participants was held at the Defense Systems Management College in Fort Belvoir, Virginia. The purpose of this workshop was to brief the organizations which would be most directly involved with information gathering efforts on the aims and status of STEP. At the same time, workshop participants were given the opportunity to present summaries of individual and organizational views on STEP. These sessions were useful for directing the attention of the study teams to an initial set of concern areas, setting the stage for more intensive individual discussions and defining a community of interest for the duration of STEP. The talks and discussions which took place at the workshop were recorded, transcribed, and edited. The edited workshop transcript and copies of all available presentation materials appear in Volume 4.

2. Overview Compilation: Overviews of current and planned technology and practices were compiled with attention to the following areas:

   a. assessing the scope and effectiveness of systematic test methodologies and techniques,

   b. assessing the cost, effectiveness, and availability of automatic test tools,

   c. identifying existing and planned policy, standards and regulations which guide software T&E activities,

   d. assessing the nature and effectiveness of government procedures which implement the items identified in (c),

   e. assessing the nature and effectiveness of contractors' responses to the procedures identified in (d).

   The results of these overview compilations form the bulk of the data used to support the findings, conclusions and recommendations reported herein. The overviews themselves appear as Volumes 2 and 3. Volume 2 is devoted to an assessment of the state-of-the-art, while Volume 3 reports the state of current practices. An integral part of Volume 2 is a section dealing with currently available tools for software T&E. This section contains detailed descriptions of tool characteristics, operational strategies and evaluations of performance. Volume 2 also contains a comprehensive bibliography to the literature in software T&E.

PHASE II MILESTONES

The major milestones for Phase II of the STEP support contract were the following:

1. Reports of Consultants and Experts: A number of consultants and other experts in the field of software T&E were selected from academic, industrial and DoD organizations. This group constituted an expert panel tasked with providing input to the support contractors in 14 targeted areas:

   - assessment of the state-of-the-art
   - assessment of the state of tool development
   - applications of reliability theory to software T&E
   - applications of metrics to software T&E
   - software error studies
   - large system test experience
   - the economics of software T&E
   - the impact of new software technology on testing
   - the impact of new hardware technology on testing
   - standardization issues
   - quality assurance and acquisition policy
   - test procedures and project management
   - the relationship between development and operational testing
   - the impact of improved software quality on future military systems.

   The panel met twice. The first meeting, held in September of 1982, served to introduce the panel to the goals and progress of STEP up to that point and to expose the central issues which would eventually form the basis for the panelists' reports. The panelists reported their findings and recommendations at a national symposium held in February, 1983 (see 2 below). The written reports of the panelists appear in Volume 5. In addition, many of the recommendations of the panelists have been incorporated into the recommendations and conclusions appearing in this volume.

2. National Conference on Software Test and Evaluation: The National Security Industrial Association (NSIA) in cooperation with the Office of the Secretary of Defense sponsored a national symposium on software T&E during the first week in February, 1983. The principal goal of the symposium was to provide a national forum for the reports of the expert panelists (see 1 above) and the preliminary findings of the STEP

support contractors. During this meeting, extensive question-and-answer sessions were conducted, taped, and transcribed. Edited versions of these transcripts appear in Volume 5. In addition to the presentations of the panelists and support contractors, presentations were made by representatives of each of the services, the Office of the Director Defense Test and Evaluation, and the Office of the Deputy Undersecretary of Defense for Research and Advanced Technology. Besides offering a forum for the STEP preliminary recommendations, this conference - along with the orientation workshop described in Phase I - helped to further expand the base of information from which the final set of recommendations was eventually distilled.

3. Applicability Study: A subcontract to Clemson University under the direction of Dr. James F. Leathrum was let in early 1982. The goal of this effort was to conduct an applicability study for a specific tactical computer system. A major finding of this subcontract related to a modelling strategy which predicted an over-designed software component of the system. This system component subsequently failed during an operational test. This study is reproduced as Volume 6: Tactical Computer System Applicability Study.

4. Recommendations: The final stage of assessment in Phase II has been the development of a set of recommendations pointing toward improved policy guidance for software T&E. The results of the information gathering efforts, the recommendations of the panelists, the recommendations and concerns of the attendees of the orientation workshop and NSIA/OSD conference, and the conclusions of the contractors and subcontractors were organized, analyzed and cast into the form of specific recommendations. Each recommendation is supported by results obtained during Phase I data gathering or Phase II evaluation. The complete list of specific recommendations appears in Chapter 6 of this volume. Based on a preliminary prioritization of these recommendations, a list of general recommendations concerning improved software T&E guidance was prepared and submitted to DDT&E. These recommendations also appear in Chapter 6 of this volume. It is implicit in making these recommendations that improvements in DoD policy for software T&E are technically feasible. The recommendations themselves address only the goals of policy improvement and do not address mechanisms or implementation strategies (although possible strategies can be inferred from the supporting arguments in Chapter 6).

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

VOLUME 2
SOFTWARE TEST AND EVALUATION: State-of-the-Art Overview

SUBMITTED TO
THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION
AND
THE OFFICE OF NAVAL RESEARCH
ONR CONTRACT NUMBER N00014-79-C-0231

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

Volume 2
Software Test and Evaluation:
State-of-the-Art Overview

SUBMITTED BY
GEORGIA INSTITUTE OF TECHNOLOGY

TO

THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION

AND

THE OFFICE OF NAVAL RESEARCH

FOR

ONR CONTRACT NO. N00014-79-C-0231
Subcontract 2G36661

June, 1983

FOREWORD

This volume is one of a set of reports on Software Test and Evaluation prepared by the Georgia Institute of Technology for The Office of the Secretary of Defense/Director Defense Test and Evaluation under Office of Naval Research Contract N00014-79-C-0231.

Comments should be directed to: Director Defense Test and Evaluation (Strategic, Naval, and C^3I Systems), OSD/OUSDRE, The Pentagon, Washington, D.C. 20301.

Volumes in this set include:

Volume 1: Final Report and Recommendations
Volume 2: Software Test and Evaluation:
State-of-the-Art Overview
Volume 3: Software Test and Evaluation:
Current Defense Practices Overview
Volume 4: Transcript of STEP Workshop, March 1982
Volume 5: Report of Expert Panel on Software Test and Evaluation
Volume 6: Tactical Computer System Applicability Study

Volume 2

Software Test and Evaluation:

State-of-the-Art Overview

Table of Contents

Page

1. Definitions and Theory of Testing 1

1.1. Program Specification and Correctness 1
1.2. Reliability and Validity 5
1.3. Deductive Approaches -- Proofs of Correctness 8
1.4. Mathematical Terminology 13
1.5. Statistical Reliability Models 15

2. Software Testing 21

2.1. Testing Strategies 21

2.2. Testing Techniques 31
2.2.1. Static Analysis Techniques 32
2.2.2. Symbolic Testing 36
2.2.3. Program Instrumentation 45
2.2.4. Program Mutation Testing 50
2.2.5. Input Space Partitioning 58
2.2.6. Functional Program Testing 63
2.2.7. Algebraic Program Testing 67
2.2.8. Random Testing 70
2.2.9. Grammar-based Testing 72
2.2.10. Data-flow Guided Testing 75
2.2.11. Compiler Testing 78
2.2.12. Real-time Software and Testing 81

2.3. Other Strategies for Constructing Reliable Software 89

2.4. Comparative Evaluation of Testing Techniques 92

3. Testing and Evaluation Tools 95

3.1. Introduction 95
3.1.1. General Views on Testing Tools 95
3.1.2. Classification 97

3.2. Static Analysis Tools 99
3.2.1. Static Analysis Tool Classification 99
3.2.2. Static Analyzers 101

Page

3.3. Dynamic Analysis Tools 115
3.3.1. Dynamic Tool Classification 115
3.3.2. Symbolic Evaluators 117
3.3.3. Test Data Generators 128
3.3.4. Program Instrumenters 141
3.3.5. Mutation Testing Tools 159

3.4. Test Supporting Tools 169
3.4.1. Automatic Test Drivers 169
3.4.2. Comparators 175

4. Comprehensive Bibliography 179

Appendix A: Information Sources for Testing Tools 274

Appendix B: Alphabetical Listing of Cataloged Tools and Testing Tool Data Sheets 276

CHAPTER 1

DEFINITIONS AND THEORY OF TESTING

1.1. PROGRAM SPECIFICATION AND CORRECTNESS

When asked to give a reason for testing a computer program, typical programmers respond, "To see if it works." In practice, the notion of a "working" program is a complex one which takes into account not only the technical requirements of the programming task but also economics, maintainability, ease of interface to other systems and many other less easily quantifiable program characteristics. As these characteristics become more complex, testing to see if a particular piece of software has those characteristics becomes more difficult. The technical literature on program testing tends to deal with "working" in one simplified disguise: correctness.

For most of this overview, we will consider the following (simplified) model of the program development cycle (see Figure 1).

At the start of the programming task, the programmer is supplied with a specification of the program. The specification may be as formalized as a document which details the intended behavior of the program in all possible circumstances or it may be as informal as a few instances of what the program is intended to do. In practice, the programmer has available to him several sources of information which comprise the specification. These may include a formal specification document, a working prototype, instances of program behavior, and *a priori* knowledge about similar software. All of these sources contribute to the programmer's understanding of the task.

Working from his specification, the programmer develops the software. The test -- or, more generally, validation -- of the software lies in the comparison of the software product with the specification of intended behavior.

For most of this overview, we will not be concerned with the exact nature of specifications. The examples we give will be small and understandable. For instance, the specification of a sorting program might be the following:

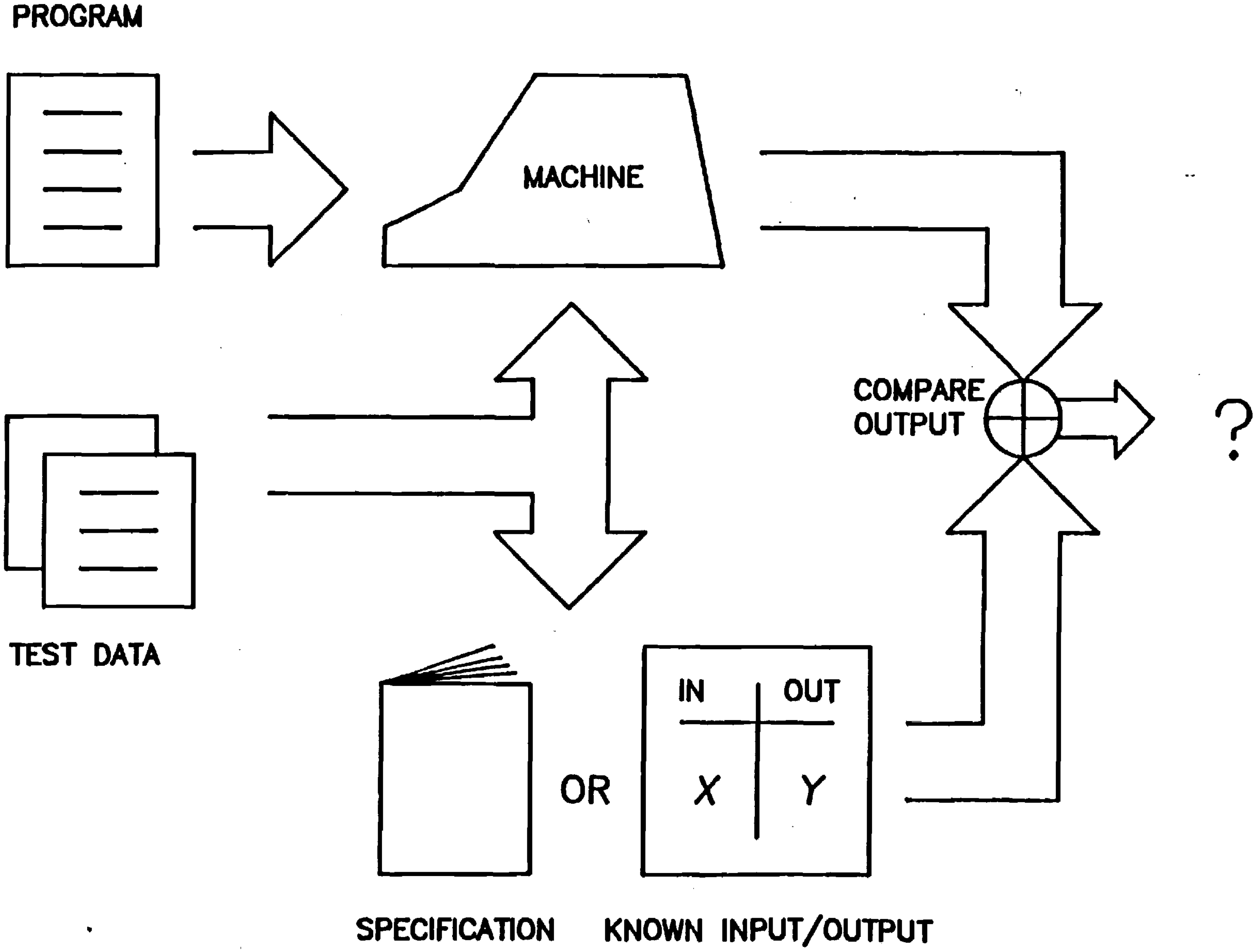


FIGURE 1
TESTING FOR CORRECTNESS

INPUT: up to 10,000 input records in the format (KEY1, KEY2, VALUE).

OUTPUT: a re-ordering of the input records with the following properties:

(1) the primary (KEY1) keys should appear in ascending order,

(2) if two records R1 and R2 have equal primary keys and if R1 precedes R2 in the input sequence, then R1 precedes R2 in the output.

Additional information could have been added to this specification. It could be required, for example, that the sorting program satisfy some performance criteria or that some standard interface conventions be followed.

Practical situations hardly ever give rise to such "clean" specifications. Much research has been devoted to the problem of specifying large and complex software systems [1,2,6,8,12,15]. For a discussion of software testing research, we will not need to be more precise with the nature of program specification.

A specification provides a description of the input data for the program. This input data is called the domain of the program and is usually represented by D. The specification also provides a description of the intended behavior of the program on D. We will represent the intended behavior on input data d by f(d). In practice, f(d) may be quite complex.

Similarly, a program P represents some computational actions which will take place when the program is supplied with input data. Even though these actions may be quite complex, we will simplify things considerably by representing the behavior of program P by P^*. Thus, $P^*(d)$ is a mathematical idealization of the behavior of program P on data item d.

The shorthand notation f(D) and $P^*(D)$ is used to represent the intended behavior on all input data and the behavior of P on all input data, respectively.

The program P is said to be correct with respect to a specification f if

$$f(D) = P^*(D),$$

that is, if P's behavior matches the intended behavior on all input data.

A problem arises in practical applications of the mathematical theory. How is it possible to determine whether or not f(d) = $P^*(d)$ for some particular datum d in the domain? If the specification document is completely formal, then it should also answer this question. However, specification documents are hardly ever completely formalized (even when they are, determining f(d) may be infeasible). When the specification is not formalized, f(d) may be obtained by hand calculation, textbook requirements or by application of estimates obtained from simulations.

Usually, these problems are not dealt with in program testing theory. Rather, we assume that an oracle exists. This oracle can judge for any specific d, whether or not $f(d)=P^*(d)$. The idealization of an oracle is essential for software testing. Various testing strategies have handled the oracle problem in different ways. For example, in some strategies, the specification is required to be uniform. That is, the specification document must provide a method for computing f(d). In some cases, this is ensured by requiring that the specification itself be executable. Other strategies, on the other hand, make no assumptions at all about how the oracle is to operate. These strategies simply present the tester with $P^*(d)$. The determination of whether $P^*(d)=f(d)$ is left to the oracle.

1.2. RELIABILITY AND VALIDITY

The correctness problem is to determine whether or not $P^*(D)=f(D)$. In program testing, this determination is made on the basis of a finite number of program executions on test data $d_0, d_1, \ldots d_n$. If

$$\begin{aligned} f(d_0) &= P^*(d_0) \\ f(d_1) &= P^*(d_1) \\ &\vdots \\ f(d_n) &= p^*(d_n) \end{aligned}$$

then we would like to be able to conclude, in general, that $P^*(D)=f(D)$. Clearly, this is not possible without some restriction on the test data. For example, since D may be infinite, P^* may simply be "rigged" to mimick f on the test data but deliver erroneous results elsewhere. The concept of reliable test data is due to Howden [7]. A set of test data T is said to be reliable for P if

$$P^*(T) = f(T) \text{ implies } P^*(D) = f(D).$$

That is, a set of test data is reliable if, by observing the results of executing P on the test data allows one to conclude that P is, in fact, correct.

How does one select reliable test data? This question was addressed in an influential paper [13] by Gerhart and Goodenough. Let C be a procedure that, for a program P, selects (possibly many) sets of test data. In order for C to select reliable test data, the procedure must satisfy two conditions: C must be reliable and valid. C is said to be reliable if, whenever C selects test data T_1 and T_2, P either matches its specification on T_1 and T_2 together or fails to match its specifications on T_1 and T_2 together. That is,

$$P^*(T_1) = f(T_1) \text{ if and only if } P^*(T_2) = f(T_2).$$

A selection procedure is said to be valid if, whenever P is not correct, C selects at least one test datum on which P fails to match its specification. More precisely, if $P^*(d)$ .NE. $f(d)$ for some d, then C selects test data T such that $P^*(T)$ .NE. $f(T)$[1].

[1] Throughout this document, for printing purposes, FORTRAN symbols will at times be used to represent relational and logical operations.

Reliable and valid selection procedure restate program correctness. That is, there is a valid and reliable test data procedure C for P selecting test data T such that $P^*(T) = f(T)$ if and only if $P^*(D) = f(D)$. To see this, first assume that P is correct, that is, suppose that $P^*(D) = f(D)$. Since the selection procedure that selects the empty set of test data is valid and reliable, the require procedure exists. Conversely, suppose that C with the required properties exist. Then P must be correct, for assume that it is not. If P is incorrect, then for some d, $P^*(d)$.NE. f(d). Since C is valid, it will select test data T such that $P^*(T)$ .NE. f(T). Since C is reliable it will then choose only test data on which P fails. This contradicts our choice of C, so we conclude that $P^*(D) = f(D)$.

Notice that reliable and valid selection procedures select reliable test data sets. That is, if C is reliable and valid, then any test data set selected by C is reliable. On the other hand, if T is a reliable test data set and T is selected by procedure C, then C is valid.

There are a number of conceptual problems with these definitions. The first, and most notable, is that the existence of reliable and valid selection procedures is equivalent to program correctness. Thus, if P is correct, showing that a particular C is valid and reliable is that same as showing that the program is correct (see section 1.3). By the same token, if P is already correct, then valid and reliable procedures can give rise to test sets which offer no empirical evidence at all that P is correct, since P will work properly on any test set [5]. Finally, validity and reliability are not independent concepts. It was observed by Weyuker and Ostrand [25] that, every selection procedure is either reliable or valid.

In many circumstances, one might search for conditions on test data that allow the tester to conclude correctness which are, nevertheless, not formally equivalent to correctness. The concept of adequate test data is due to DeMillo, Lipton and Sayward [11]. A test data set T is adequate for P if $P^*(T) = f(T)$ and for all Q such that $Q^*(D)$ .NE. f(D), $Q^*(T)$ .NE. f(T). In other words, T is adequate if P behaves correctly on T but all incorrect program behave incorrectly. It is a simple consequence of the definitions that if T is adequate then it is reliable. On the other hand, reliability does not imply adequacy since if P is correct, any test set is reliable.

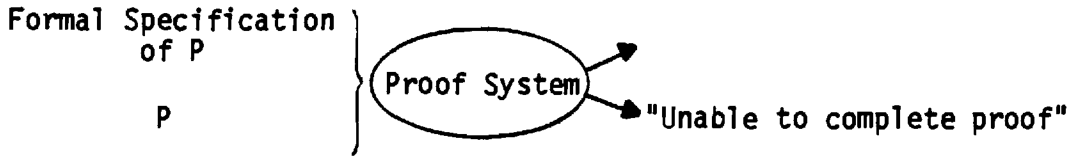
There are no general-purpose valid and reliable test selection procedures. In technical terms, no valid and reliable test selection procedure is computable [12]. The goal of testing research, then is to limit attention to specific categories of errors for which selection procedures are valuable. For example, we might construct test data that is adequate, relative to a set of programs A. We say that T is adequate for P relative to A if $P^*(T) = f(T)$ and for all program Q in A, $Q^*(D)$ .NE. f(D) implies $Q^*(T)$ .NE. f(T). For example, A might represent a certain set of errors which might be introduced into a program. Then the existence of an adequate set of test data demonstrates that P does not contain A-type errors. To see why, it is only necessary to observe that if T is adequate relative to A, then either T is reliable or P is not in A. If T is adequate for P relative to A, and T is reliable there is nothing to show. Suppose that T is not reliable. Then $P^*(D)$ .NE. f(D). But for all Q in A, if Q is not correct, then $Q^*(T)$ .NE. f(T). Since $P^*(T)$ = f(T), P cannot be in A.

1.3. DEDUCTIVE APPROACHES -- PROOFS OF CORRECTNESS

An approach to determining whether or not $P^*(D) = f(D)$ is to *prove* that the equation holds. The general strategy is as follows: if P is correct, then by a rigorous mathematical analysis of P and its specifications, it is to be proved that for all input data x in D (i.e., data meeting P's input specifications) if P operates on x, then $P^*(x)=f(x)$ (i.e., P meets its output specifications). If P, on the other hand, is incorrect, then in attempting to develop such a proof, the error will be uncovered.

There is a distinction between proving a program correct and testing to see if it works. Proving correctness is a *deductive* activity, while testing is an *inductive* activity. In proofs of correctness one argues about *all* input that satisfy a program's input specifications, and the conclusion of the argument -- that P is correct -- is *mathematically valid*. In testing, one *observes* instances of program execution, and from this observation draws the conclusion that P is correct. However, the conclusion is not mathematically valid unless the set of observations are drawn from a reliable test set. Since reliable test sets may not be practical to obtain, the tester may have to choose test data which is not as strong. One may observe program execution over an *adequate* test set. In this case, however, it can only be concluded that P is correct with high probability (either the test set is reliable or P is not in A, an event of low probability). Even though proving a program correct offers the hope of certainty that P is correct, it is seldom applied in practice (we will sketch some reasons for this at the end of this section). Nevertheless, the theoretical basis of correctness is so closely tied to program proving that it will be helpful to present a brief sketch of the theory.

The following diagram illustrates the process of proving a program correct.



The *proof system* is a set of rules that allow the formal derivation of the proof of correctness, if one exists. These rules may be simply a description of how a proof is to be developed by a human being, or may actually be implemented in a *mechanical proof system*, so that a proof is automatically constructed. Hybrid systems have also been proposed in which a human operator gives "help" to a mechanical system. The advantages of each approach have been extensively argued in the literature [3,4,7,16,20,21]. All of these systems share some common characteristics.

The first requirement is that the input and output specifications be expressed in a suitably formalized way. Such a formalization serves two ends. First, the specifications can be uniformly and exactly stated. Second, the formalization itself can be used to carry out the necessary logical inferences. The overwhelming majority of work in program proving is carried out in a language called the first order predicate calculus or FOPC. FOPC is essentially the language of elementary formal mathematics. In adapting FOPC to serve as a tool for correctness proofs, it may be enriched with some additional devices oriented toward programming. The basic language of FOPC has provisions for the following:

variables $(x,y,x_1,y_1,\ldots)$,
constants (a,0,7.6, ...),
functions (f(x),g(s,y),+,...), and
predicates (A(x,y),x=y,x.LE.0, ...).

Statements involving individuals (i.e., constants, variables, and values of functions) are constructed by combining names for individuals with predicates (e.g., f(0).LE.x+y is a statement). Statements may be combined with the boolean operations .AND., .OR., .NOT., and $\supset$ (implies) to form compound statements (e.g., x=y+z .AND. y=0 $\supset$ x=z). FOPC also allows quantification over individual variables using the symbols $\forall$ (for all) and $\exists$ (there exists). Thus, the statement that every integer has an additive inverse may be expressed in FOPC as $\forall\ x\ \exists\ y\ (x+y=0)$.

FOPC is frequently used as the basic language of logical theories. A statement of FOPC is a theorem if it is derivable from a list of axioms by a list of rules which constitute a proof system. Some axioms are essentially formalizations of logical truths (i.e., for any predicate A(x) either an individual has that property or not, so an axiom of FOPC is $\forall$x(A(x) .OR. .NOT.A(x)). Other axioms may specify specific properties of some mathematical system (e.g., the statement that every integer has an additive inverse).

In a proof of correctness of a program P with input specifications A(x) and output specifications B(x), the idea is to use the proof systems for program correctness and for FOPC to prove that whenever A(x) is true before P is execute, then B(x) is true after P is executed. This situation is sometimes illustrated with the notation A(x) [P] B(x). By defining the effects of language features on input and output predicates, a procedure which defines the proof system can be obtained. For example, if we combine two program segments P and Q into a single program P;Q, then the output specifications for P become the input specifications for Q, so if A [P] B and C [Q] D can be proved and if B -- C, then A [P;Q] D can be proved.

Looping constructs must be handled fairly carefully. Essentially a proof system must allow for the possibility of arbitrarily many interations through a WHILE loop. In the most commonly used proof systems, the method of inductive assertions is used. The basic idea is to find input and output specifications for the loop (e.g., A(x) for the input and B(x) for the output) so that A(x) is true when the loop is entered for the first time. Furthermore -- and this is the inductive part -- if A(x) is true after some n^{th} iteration through the loop (i.e., before the $n+1^{st}$ iteration), then A(x'), where x' is possibly a new value of x assigned during one iteration of the loop, is true after the $n+1^{st}$ iteration. In addition, if the loop ever terminates, it terminates with B(x) true.

The following example illustrates the main ideas in proofs of correctness. The literature in this area is extensive, and the reader should consult such sources as [11,20] to examine the possible variations. The program to be proved computes the quotient q and remainder r on dividing integer x by integer y. The axioms of FOPC are assumed along with the axioms of elementary arithmetic. The program is:

```
r:=x; q:=0;
WHILE y.LE.r DO
     BEGIN r:=r-y; q:=q+1 END.
```

The proof is broken into two parts.

```
TRUE [r:=x;q:=0] x=r+y*q

x=r+y*q [WHILE y.LE.r DO BEGIN r:=r-y;q:=q+1 END] x=r+y*q r.LT.y.
```

The input specification TRUE indicates that no additional assumptions about the input variable x and y are needed. If both of these program segments can be proved, then they can be combined to prove the program using the rule A [P] B and B [Q] C implies A [P;Q] C. The proof system is responsible for converting these A [P] B expressions into FOPC statements. The first results in

```
TRUE .AND. r=x .AND. q=0  ⊃ x=r+y*q
```

The loop generates two FOPC statements. The first corresponds to continued execution of the loop (i.e., y.LE.r) and the second corresponds to loop termination.

```
x=r+y*q .AND. y.LE.r .AND. r'=r-y .AND. q'=q+1  ⊃x=r'+y*q

x=r+y*q .AND. y.LE.r  ⊃ x=r+y*q .AND. r.LT.y.
```

These FOPC statements are sometimes called *verification conditions*. To complete the proof of the program, it is necessary to construct a FOPC proof of the verification conditions. This may be very easy (e.g., in the first verification condition above, the proof follows by simple substitution and manipulation). In general, however, the verification conditions are lengthy statements whose proofs are not apparent.

Formal proofs of correctness constitute analyses of programs. Several authors have also suggested that *synthetic* tools can be based on the deductive approach. That is, by running a proof system *backward* one can derive a program that meets its input and output specifications [7,14].

The issue of whether or not complex software can be proved correct is not yet settled. Proponents of correctness proofs argue in the following way. Properties of program are subject to mathematical proof (that is, by analyzing the program and its specifications, one obtains a mathematical statement that may be proved using the usual laws of logic and mathematics). Provided the symbolic analysis and proofs are flawless, one can then conclude that the program when started in an acceptable state, will, if it terminates, produce an acceptable output state. The statement that the program is correct has been proved for all possible test cases. The technique is applicable in principle to all programs. Its application to complex software awaits only advance automated systems to aid in the process of proof [24].

Opponents of program proving argue that proofs of correctness seriously misinterpret

(1) the concept of proof,
(2) the nature of software specifications, and
(3) the notion of reliability.

The interested reader may consult [10] for detailed discussions of (1) and (2).

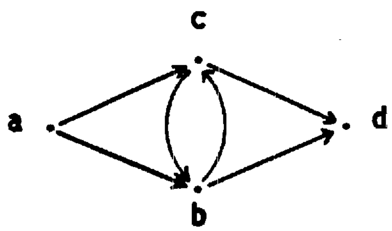
Point (3) is of special importance for program testing. Presumably, a program that is sufficiently large and complex will never be correct in the sense given above (i.e., it will always contain errors and therefore there will never be a correctness proof). That does not mean that the program is unreliable. The problem is that the deductive approach treats reliability as a two-valued function (either the program is correct or it is not). For software this assumption is unacceptable. What is needed is not a two-valued criteria, but a theory that orders uncertain events (i.e., the error-free operation of a piece of software).

Even though we will not explicitly consider deductive approaches in the remainder of this overview, the terminology and viewpoints have infused the field of program testing, and the reader may wish to consult one of the survey articles in the field or any of the more recent textbooks [20,21,24] for more details.

1.4. MATHEMATICAL TERMINOLOGY

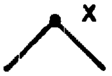
We collect here some terminology that is common to program testing research.

A graph is a collection of nodes connected by directed lines or arcs. For example, the illustration below shows one representation for a graph G.



Mathematically, a graph G is defined by a set of nodes N(G) and a set A(G) of ordered pairs of nodes which represent the arcs. For the example above N(G) = [a,b,c,d] and A(G)=[(a,b),(a,c),(b,c),(c,b), (b,d),(c,d)]. A path through a graph G is a sequence of nodes that can be traversed by following the arcs in the proper direction. For the graph given above (a,b,c,b,d) is a path while (d,b,c,b,d) is not a path. A path is a cycle if the start and end nodes along the path are the same. A cycle is simple if it contains no other cycle. A set of nodes is a cutset for the graph if the removal of that set of nodes (and their incident arcs) breaks the graph into two pieces G_1 and G_2 so that there are no paths from nodes in G_1 to nodes in G_2. A graph is strongly connected if for any two nodes x and y, there are paths leading from x to y and from y to x. Notice that the graph given above is not strongly connected.

If a graph has a unique node h (the header) so that every other node in the graph can be reached from h and a unique node f (the final node) so that there is a path from every node in the graph to f, then the graph is called a flow graph. Flow graphs are abstract representations of program flow charts. The nodes of the graph correspond to program statements and the arcs correspond to control paths. A node of the form



corresponds to a decision branch, and we assume that the arcs are labelled so that the YES/NO branches correspond to the two sides of the conditional branch.

A *program path* is any path through the associated flowgraph (i.e., from h to f). A *branch* is any path from decision point to decision point. In other words $(x_1,\ldots,x_n)$ is a branch if x_1 is either h or a decision node, x_n is either f or a decision node, and none of the nodes $x_2, \ldots, x_{n-1}$ are decision nodes.

1.5. STATISTICAL RELIABILITY MODELS

In practice a program may be judged reliable if it has been formally proven correct, if it has been run against a reliable and valid test data, or it has been developed according to a special discipline. But these approaches fail to quantify the extent to which software meets its operational objectives.

To evaluate a system's behavior in quantitative terms, when direct measurement is impossible, one needs a reliability model. A key requirement for such a model is that tun time and number of errors revealed during testing be recorded. Since, in most projects, the earliest time one can start gathering accurage software reliability data is after the start of integration test, the focus is on the integration test phase of software development.

A number of software reliability measures have been defined, and they have been used in software reliability analyses. The most important reliability measures are:

- Failure rate
- Reliability functions

In general, software reliability is measured by identifying successful runs (S) among a predetermined total number of runs (N). The index of software reliability is then the ratio of successful runs to total, or

$$R = S/N$$

The failure or unreliability rate can be expressed as

$$U = F/N$$

where F is the number of failed runs. This definition is applicable to conventional batch processing environments and real-time systems dealing with discrete operations. For real-time systems dealing with continuous streams of data, a more realistic index is mean time between failures (MTBF). It is expressed as

$$MTBF = t/F$$

where t is the predetermined total running time and F is the total number of failures in the interval [0,t]. The failure rate is then

$$u = 1/MTBF$$

The failure rate measure is used to build error models (which will be discussed later).

The reliability function is another software reliability measure. It results from extensive classical statistical reliability theory to software. Experience in the hardware area suggests that reliability must be defined as the probability of satisfactory performance of the system in the time interval [0,t], [18,23]. Another factor which must be specified is the hardware environment. For instance, if a program is written to run on a particular system will probably be modified to run on a different system. A common definition of software reliability which takes into account the factors mentioned above is as follows:

> Software reliability is the probability that a given software system operates for some time period without software error, on the machine for which it was designed given that it is used within design limits.

When this definition is mathematically expressed, we obtain the following reliability function

$$R(t) = \exp\left[-\int_0^t Z(t)\ dt \right]$$

where Z(t) is the estimated error rate. MTBF can then be expressed as

$$MTBF = \int_0^\infty R(t)dt$$

These reliability measures have been used in about 15 reliability models. Some of these models have been investigated in detail and applied to actual software projects [19,22,23]. These models differ mainly in the assumptions they make to characterize the failure rates. Two types models derived from the different measures of reliability are error models and reliability models. They can be used independently or together in assessment of reliability to a project.

Error models are mainly used to predict the remaining number of errors (Er) by assuming that the total number of errors (Et) in the program is known before it enters the integration test and errors are immediately corrected as they are found. Then, after integration testing for time t the remaining number of errors is

$$Er(t) = Et(t) - Ec(t)$$

where Ec is the number of errors corrected [18,23]. Realistic error models can be constructed by using collecred error data to get some idea about the error distribution for the type of software being tested. For examples of collected error data and the variations of the basic model described above see [18,22,23].

Reliability models are used to predict the number of errors left uncovered after integration testing has been completed. Similar to error models, realistic reliability models are constructed with reference to collected reliability data. Shooman [23] describes how to build such a reliability model from experimental data and how to estimate the parameters of the model. The underlying assumptions of Shooman's model include: errors are detected randomly and independent of each other, errors are corrected as soon as they are detected, and error rate is constant the detection of two subsequent errors and is proportional to the number of remaining faults [19]. The reliability models that employ the assumptions mentioned above are also know as exponential reliability models.

Exponential reliability models can lead to serious errors when the underlying distribution corresponds to failures that do not occur randomly or that depend on the history of the system. The Weibull reliability model circumvents the deficiencies of the exponential reliability model. In this model, the error rate is a function of time and is defined as follows:

$$Z(t) = a \cdot b \cdot t^{b-1}$$

Obviously, the error rate is decreasing or increasing depending on the value of b.

There are other types of software reliability models, i.e., truncated normal distribution model, Kelinski and Moranda models, and Shooman model. For a discussion of these models see [19,23].

Experimental evidence gained by the applications of these models mentioned above shows that these models seem to be better applied to a posteriori reliability assessment than to reliability prediction. There also do not exist data which can be analyzed reliably by these models at the same time. Therefore, it is hard to make comparisons regarding the usefulness and the accuracy of these models [19].

Even if there may be some experimental data fitting the appropriate error rate functions of these models, the underlying assumptions of their reliability distribution forces them to make questionable assumptions concerning errors in software. We mentioned some of the assumptions of exponential distribution models above. Other assumptions include: (1) the number of initial program errors can be reliably estimated and, (2) the size of the program is constant over its lifetime. These two assumptions have been tested and found to support the exponential rate. There is also considerable evidence that, for large systems, most remaining errors lie in unexecuted portions of code, which means that for these systems error rates cannot depend on either the number of remaining errors or debugging effort [22].

REFERENCES

[1] A. L. Ambler, et al.
GYPSY: A Language for Specification and Implementation of Verifiable Programs.
Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol.12(3), March 1977.

[2] D. E. Bell and L. J. LaPadula.
Secure Computer Systems.
Report ESD-TR-73-278, Mitre Corporation, Beford, MA.
November 1973.

[3] H. K. Berg, W. E. Boebert, W. R. Franta, and T. G. Moher.
Formal Methods of Program Verification and Specification.
Prentice-Hall, Inc.

[4] R. S. Boyer and J. S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

[5] T. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
The Design of a Prototype Mutation System for Program Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:623-7, 1978.

[6] R. H. Campbell and A. N. Haberman.
The Specification of Process Synchronization by Path Expressions.
Lecture Notes on Computer Science, Vol.16, 1974.

[7] Chin-Liang Chang and R. Char-Tung Lee.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, New York, 1973.

[8] R. M. Cohen.
Formal Specifications for Real-time Systems.
Proceedings of the Seventh Texas Conference on Computing Systems, October 1978.

[9] R. A. DeMillo.
Program Mutation: An Approach to Software Testing.
Report GIT/ICS-83-03, Georgia Institute of Technology,
January 1983.

[10] R. A. DeMillo, R. J. Lipton, and A. J. Perlis.
Social Processes and Proofs of Theorems and Programs.
Communications of the ACM, May 1979.

[11] R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Hints on Test Data Selection: Help for the Practicing Programmer.
Computer, Vol.11(4):34-41, April 1978.

[12] L. Flon and A. N. Habermann.
Towards the Construction of Verifiable Software Systems.
Proceedings of Conference on Data: Abstraction, Definition, and Structure, SIGPLAN Notices, Vol.8(2):141-8, 1976.

[13] J. B. Goodenough and S. L. Gerhart.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering, Vol.SE-1(2): 156-73, June 1975.

[14] D. Gries.
The Science of Programming.
Springer-Verlag, New York, 1981.

[15] J. V. Guttag.
The Specification and Application to Programming of Abstract Data Types.
Ph.D. Thesis, University of Toronto, Report CSRG-59, 1975.

[16] S. L. Hantler and J. C. King.
An Introduction to Proving the Correctness of Programs.
ACM Computing Surveys, September 1978.,

[17] W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions on Software Engineering, Vol.SE-2(3):208-14, September 1976.

[18] Infotech State of the Art Report, Software Testing, Volume 1: Analysis and History.
Infotech International, 1979.

[19] Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers.
Infotech International, 1979.

[20] Z. Manna.
Mathematical Theory of Computation.
McGraw-Hill, 1974.

[21] Z. Manna and R. Waldinger.
The Logic of Computer Programming.
IEEE Transactions on Software Engineering, Vol.SE-4:199-229, 1978.

[22] A. J. Perlis, F. G. Sayward and M. Shaw.
Software Metrics: An Analysis and Evaluation.
THE MIT Press, Cambridge, MA, 1981.

[23] M. L. Shooman.
Managing Software Testing Using Reliability Estimates.
National Conference on Software Test and Evaluation, February 1983.

[24] P. Wegner.
Research Directions in Software Technology.
The MIT Press, Cambridge, MA, 1979.

[25] E. J. Weyuker and T. J. Ostrand.
Theories of Program Testing and the Application of Revealing Subdomains.
IEEE Transactions on Software Engineering, Vol.SE-6(3): 236-46, May 1980.

CHAPTER 2

SOFTWARE TESTING

2.1. TESTING STRATEGIES

SUMMARY

Module testing is the process of testing logical units of a program individually, and integrating the individual module tests to evaluate the overall system. Main considerations in performing module testing are the design of test cases and the coordination of testing multiple modules. Test cases may be constructed from specifications or by analyzing the module code. Testing strategies corresponding to these approaches are called black-box and white-box, respectively. There are two approaches to combining module analysis: nonincremental and incremental. Top-down and bottom-up are two incremental testing strategies. Thread testing is another such strategy based on system requirements specification. A final strategy requires testing software throughout its development.

INTRODUCTION

Module testing involves the process of testing the logical units of a program (e.g., procedures or subprograms) individually, and integrating the individual module tests to test the overall system. The objective of module testing is to determine whether the module meets its specifications [4,11].

In order to perform module testing two things need to be considered: the design of test cases and the coordination of testing multiple modules. Test cases may be constructed from specifications or by analyzing the module code. The testing strategies corresponding to these two approaches are called black-box and white-box testing, respectively [8,9]. There are two approaches to combining module analysis: nonincremental and incremental. In the nonincremental approach, a program is tested by testing modules independently and then combining them to form the program without further testing. In the incremental approach, a module is tested in combination with the set of previously tested modules. The incremental approach allows earlier detection of errors. Two strategies to incremental testing are top-down and bottom-up: both strategies assume that the calling sequence of modules is a directed acyclic graph [11].

Thread testing is an incremental strategy based on a system verification diagram derived from the requirements specification [4]. Another strategy suggests that testing should start in the early stages of the development of software. Two methods proposed by Fagan and Miller [9] are introduced.

BLACK-BOX TESTING

In black-box (or functional) testing, the internal structure and behavior of the program is not considered. The objective is to find out solely when the input-output behavior of the program does not agree with its specifications. In this approach, test data for a program are constructed from its specifications [9,11].

In order to minimize the number of test cases, the input space of a program is partitioned into equivalence classes with respect to the program's input specifications, i.e., each equivalence class is a "case" covered by an input specification. Identifying the equivalence classes usually requires a heuristic approach. Myers gives a set of heuristics in [9] to identify the equivalence classes. If the specification of a program is described by a formal specification language, the specification of a program can be partitioned into equivalence classes (subspecifications). An example of such a partitioning is given in section 2.2.5.

Another methodology is to pick test data that lie on and near the boundaries of input equivalence classes. Test data can be selected similarly on a partitioning of the output space of a program. The number of combinations of input data is generally very large with these approaches. Thus, a systematic way of selecting high-yield test cases is needed. Cause-effect graphing is one such technique, in which the casual relationships between distinct input and output conditions are described by Boolean operators. The cause-effect graph is then converted into a decision table. Each column of that decision table corresponds to a test case [11].

Error guessing is another test data generation technique in which possible errors are listed and test cases based on this list are constructed. Since it is a largely intuitive and ad hoc process, a procedure for test data generation cannot be given [11].

Random testing is another black-box testing strategy in which a program is tested by randomly selecting some subset of all possible input values. As indicated by Duran [5], there has been strong disagreement about its value. Myers [11] considers the random testing as "probably the poorest" test case design methodology. On the other hand, Thayer, et al [13], recommend the use of random testing for final testing of a program by selecting test data from an "expected run-time distribution" of its inputs. The methodology of generating random test data can also be employed for real-time testing (see section 2.2.12). Experimental results confirming the viability of random testing have been obtained by Duran in [5].

Test data can also be generated automatically. Test data generation tools are the topic of section 3.3.3.

The major drawbacks of black-box testing are its dependence on the specification's correctness (which usually is not the case in practice) and the necessity of using every possible input as test case in order to be assured of module correctness [9,11].

WHITE-BOX TESTING

In this approach, the structure of the program is examined and test data are derived from the program's logic. There are criteria to determine the coverage of a program's logic. One such criterion is to require every statement in a program to be executed at least once. This criterion is necessary, but it is in no way sufficient since some errors may go undetected [11].

Another criterion requires partitioning the input space of a program into path domains and constructing test cases by picking some test data from each of these path domains to exercise every path in a program at least once. In practice, there may be an infinite number of paths in a program. Thus, a procedure is needed to select a subset of the total set of paths, but exercising every path in a program is not guaranteed to detect all possible errors [8,11].

Branch (or decision) coverage is a stronger criterion than statement coverage. It requires every possible outcome of all decisions to be exercised at least once. It includes statement coverage since every statement is executed if every branch in a program is exercised once. The problems associated with this criterion are that a program may contain no decision statements, a program may contain multiple entry points, some statements may only be executed if the program is entered at a particular entry point, and if a program contains exception handling routines, these routines may not by be executed at all. Thus, the requirement of this criteria must be extended to handle these cases [11].

Another criterion is condition coverage, which requires each condition in a decision statement to take on all possible outcomes at least once. The problems of decision criterion also apply to this criterion; therefore, the requirements of this criterion must be extended similarly. In the case of IF statements, condition coverage is sometimes better than decision coverage since it may cause every individual condition in a decision to be executed with both outcomes. Condition coverage criterion does not include decision coverage since test data exercising every condition value may not cover all decision outcomes [11].

Sometimes the decision and the condition criteria are applied together (decision/condition). But even this approach has a weakness; the errors in logical expressions may go undetected since some conditions may mask out other conditions [11]. A criterion that handles this problem is called multiple condition criteria. In addition to the requirement of a decision/condition criterion, a multiple condition criterion requires construction of test cases to exercise all combinations of condition outcomes in every decision statement [11].

Domain testing is a modified form of path analysis testing. It attempts to reveal the errors in a path domain by picking test data on and slightly off of a given closed border (see section 2.2.5).

TOP-DOWN TESTING

A top-down testing strategy starts with the top module in a program and then proceeds to test modules at lower levels progressively. In order to simulate the function of the modules subordinate to the one being tested, some dummy modules, called stub modules, are required [4,8,14].

Although there is no formal criterion for choosing an order among subordinate modules for testing, there are some guidelines to obtaining a good module sequence for testing. If there are critical modules (a critical module might be a module suspected to be error prone), these modules should be added to the sequence as early as possible, and input-output modules should be added to the sequence as early as possible [11].

Major advantages of top-down testing are that it eliminates separate system testing and integration, it allows one to see a preliminary version of the system and it serves as evidence that the overall design of the program is correct. One result may be an improvement of programmer morale [9,11].

Major disadvantages of a top-down strategy are that stub modules are required and the representation of test data in stubs may be difficult until input-output modules are added. Test data for some modules may be difficult to create if data flow among modules is not organized into a directed acyclic graph, since stub modules cannot simulate the data flow, and observation and interpretation of test output may be difficult [6].

BOTTOM-UP TESTING

Bottom-up strategies start with modules at the lowest level (modules that do not call any other modules) in a program. Driver modules are needed in order to simulate the function of a module superordinate to the one being tested [4,9,11,14]. Modules at higher levels are tested after having tested all of their subordinate modules at lower levels [11].

An advantage of bottom-up testing is that there is no difficulty of creating test data, since driver modules simulate all the calling parameters even if the data flow is not organized into a directed acyclic graph. If the critical modules are at the bottom of the calling sequence graph, a bottom-up strategy is advantageous [11].

Major disadvantages of bottom-up testing are that a preliminary version of the system does not exist until the last module is tested, and design and testing of a system cannot overlap since one cannot start testing before the lowest level modules are designed [11].

THREAD TESTING

Thread testing is based on the "system verification diagram" (SVD) derived directly from the program requirements specification. Deutsch [4] describes the testing procedure as follows:

> In this approach, software test and construction are intertwined; they do not occur separately and sequentially. The order in which the software is coded, tested, and synthesized is essentially determined by the SVD that defines the test procedure. The SVD has segmented the system into demonstrable functions called threads. The development of the threads are calendarized. The modules associated with each thread are coded and tested in an order that is commensurate with this calendarization. The threads are synthesized into higher order sections called builds; each build incrementally demonstrates a significant partial capability of the system. This culminates in a demonstration of the full system, which occurs as a natural concluding step of integrating the last build to the accumulation of previous builds ...

Thread testing is well-suited to real-time system testing. Real-time systems implement critical functions that require immediate responses in real time. The implementation of these functions can be scheduled early in the development of a such system [4] (also see section 2.2.12).

TESTING IN SOFTWARE DEVELOPMENT

In practice, software testing is usually performed after code has been produced. But, it has been observed that the later an error has been detected, the more expensive it is to correct. This observation encourages testing early in the development of software [2,9].

Two methods are proposed by Fagan and Miller in [9] for testing software early in its development lifecycle. Fagan considers software development as consisting of a statement of objectives, design, coding, testing, and shipment. The inspection of objectives: design plans, and code is performed before the code is actually tested. Design, code, and test stages can be repeated until it is believed that the software meets its requirements. Miller describes the development of software in three phases with an optional fourth phase by assuming a stable program. The first phase is manual analysis in which the requirements specification, design and implementation plans, the program itself, and all other available information is analyzed. The second stage is static analysis in which the requirements and design documents, and code are analyzed, either manually or automatically (see section 2.2.1). Dynamic analysis is the third stage in which the software is tested with a set of test data (see section 2.2.2 - 2.2.7). The optional fourth stage is that of proving the program correct. It may be reserved for critical modules.

Recently, there have been discussions about the utility of the development cycle process [1,3,6,7,10]. Blum [1] considers the life cycle model with respect to two factors: problem comprehension and ease of implementation. Problem comprehension refers to the completeness of problem understanding before implemention begins. He points out that if the project has a high technical risk, then considerable analysis is required before the design is complete. Furthermore, he indicates that "throwaway" prototypes [12] in alternate programming languages can be used as analytic tools for such projects [1].

Ease of implementation refers to the availability of tools which support the deferred binding of requirements. He points out that the implementation cycle is too slow to incorporate changes reflecting new knowledge [1].

> The first, system architecture, implies that all necessary information is available prior to the start of implementation. The second model, system sculpture, supports the implementation of applications in which knowledge of the system requirements is refined or generated during the implementation process. It requires tools which allow the binding of new requirements into the final system. As the title suggests, the designer begins with a rough functional model and, through interaction with user and model, modifies it to create the final system.

He then compares the two approaches with respect to high technical risk, high application risk, the model of life cycle (classical or dynamic), requirements, tools, prototypes, test (against requirements or subjective acceptance), maintenance, response to new requirements, and applicability to large projects [1].

Blum suggests the categorization of different types of systems which are being developed, the consideration of the alternate life cycle models which are applicable to each, and the identification and the development of the tools and methods which facilitate the implementation of these different application classes [1].

The effect of the alternate models on the development of new methodologies for testing software in its development is yet to be seen.

REFERENCES

[1] B. I. Blum.
The Life Cycle -- A Debate For Alternative Models
Software Engineering Notes, Vol.7(12), 1982.

[2] B. W. Boehm.
Software Engineering.
IEEE Transactions on Computers, Vol.C-25(12), December 1976.

[3] Command and Control Software Development and Acquisition Study, NSIA, 1983.

[4] M. S. Deutsch.
Software Verification and Validation Realistic Project Approaches.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[5] J. W. Duran and S. Ntafos.
A Report on Random Testing.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 179-83.

[6] G. R. Gladden.
Stop the Life Cycle, I Want to Get Off.
Software Engineering Notes, Vol.7(10), 1982.

[7] P. A. V. Hall.
In Defense of Life Cycles.
Software Engineering Notes, Vol.7(11), 1982.

[8] W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions of Software Engineering, Vol.SE-2(3), September 1976.

[9] Infotech State of the Art Report, Software Testing Volume 1: Analysis and Bibliography.
Infotech International, 1979.

[10] D. D. McCracken and M. A. Jackson.
Life Cycle Concept Considered Harmful.
Software Engineering Notes, Vol.7(10), 1982.

[11] G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

[12] S. L. Squires, M. Zelkowitz and M. Branstad.
Rapid Prototyping Workshop: An Overview.
Software Engineering Notes, Vol.7(11), 1982.

[13] R. A. Thayer, M. Lipow and E. C. Nelson.
Software Reliability.
North-Holland, Amsterdam, 1978.

[14] E. Yourdon and L. L. Constantine.
Structured Design Fundamentals of a Discipline of Computer Program and Systems Design.
Prentice-Hall, Inc., Englewood Cliffs, NJ.

REFERENCES NOT CITED IN TEXT

E. M. Boehm, R. K. McClean, and D. D. Urfrig.
Some Experience with Automated Aids to the Design of Large-Scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1:125-33, 1975.

E. B. Daily.
Software Development.
Proceedings of European Computing Review, Infotech International, Ltd., 1978.

R. Dunn and R. Ullman.
Quality Assurance for Computer Software.
McGraw Hill Book Company, New York, 1982, pages 166-168.

M. E. Fagan.
Design and Code Inspections to Reduce Errors in Program Development.
IBM Systems Journal, Vol.15(3), pages 182-211, 1976.

R. E. Fairley.
Tutorial: Static Analysis and Dynamic Testing of Computer Software.
Computer, pages 14-23, April 1978.

M. S. Fugi.
Independent Verification of Highly Reliable Programs.
Proceedings of COMPSAC 77, pages 38-44, IEEE, 1977.

R. L. Glass.
Software Reliability Guidebook.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979, pages 86-95.

W. E. Howden and E. Miller.
A Survey of Static Analysis Methods.
In Tutorial: Software Testing and Validation Techniques, IEEE, 1981, pages 101-15.

G. J. Myers.
A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.
Communications of the ACM, Vol.21(9):760-88, 1978.

M. P. Perriens.
An Application of Formal Inspections to Top-Down Structured Program Development.
RADC-TR-77-212, IBM Federal Systems Division, Gaithersburg, MD, 1977, (NTIS AD/A-041645).

D. Teichrow and E. A. Hershey, III.
PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Translations on Software Engineering, Vol.SE-3:41-8, 1977.

2.2. TESTING TECHNIQUES

Software quality should be a primary concern in software development efforts. The traditional methods of assessing software quality are software testing and software evaluation.

Software evaluation examines the software and the processes used during its development to see if stated requirements and goals are met. Static analysis techniques employ this method of software quality assessment. In these techniques, the requirements and design documents and the code are analyzed, either manually or automatically, without actually executing the code.

Software testing assesses software quality by exercising the software on representative test data under laboratory conditions to see if it meets stated requirements. One testing approach consists of demonstrating that all paths of the software have been traversed successfully. An analogous approach is to test a program for possible input cases to see if the correct outputs are produced. Since these approaches are clearly prohibitive, more pragmatic approaches are considered, e.g., the input space of a program is partitioned into path domains, i.e., subsets of the program input domain that cause execution of each path, and the program is executed on test cases which are constructed by picking test data from these domains. Examples of such techniques are input space partitioning, symbolic testing, random testing, algebraic program testing, grammar-based testing and data-flow guided testing. Another approach is to instrument the program by recording processes which do not affect the functional behavior, but record properties of the executing program.

Real-time software testing and functional program testing employ different approaches than the ones mentioned above.

In mutation testing, test data is applied to the program being tested and its mutants, i.e., programs that contain one or more likely errors. If a program passes a mutation test, then either the program is correct or it contains an improbable error. Strictly speaking, mutation testing is a metric device for evaluating the adequacy of test data rather than a testing technique.

2.2.1. STATIC ANALYSIS TECHNIQUES

SUMMARY

In static analysis, the requirements and design documents and the code are analyzed, either manually or automatically, without actually executing the code. Only limited analysis of programs containing array references, pointer variables, and other dynamic constructs is possible using this technique. Experimental evaluation of code inspections and code walk throughs has found these static analysis techniques to be very effective in finding from 30% to 70% of the logic design and coding errors in a typical program.

REQUIREMENTS ANALYSIS

The users of a system define its requirements in terms of their needs. Traditionally, the requirements are analyzed using a checklist of correctness conditions, including such properties of the requirements as consistency, their necessity to achieve the goals of the system, and the feasibility of their implementation with existing resources. Different properties may require different methods to check for correctness [7].

Requirements can be defined by a requirements specification language and then checked by an analyzer. Teichrow's [14] problem statement language/problem statement analyzer (PSL/PSA) is such a system. In this system, the user models the system in PSL and PSA checks for the consistency of the model [9].

DESIGN ANALYSIS

The elements of a software system design, e.g. the algorithms, the data flow diagrams, and the module interfaces, can be analyzed by using a checklist similar to the one used in requirements analysis. Each property specified in the checklist may be checked by a different method. For instance, the consistency of module interfaces can be determined by comparing the common parts of different design elements [9]. Modelling and simulation can be used to determine if the design meets the performance requirements [9,13].

An inductive assertions method, e.g. symbolic testing, can be used for formal analysis of some design elements. For instance, this method has been used prove the correctness of algorithms included in the program design. The method has also been employed by TRW's Design Analysis Consistency Checker (DACC) for checking the consistency of module interfaces [1].

CODE INSPECTIONS AND WALKTHROUGHS

Code inspections and walkthroughs involve the visual inspection of a program by a group of people, first individually, then as a group, in order to detect deviations from specifications.

A code inspection is a set of procedures to detect errors during group code reading [3,5,8,10]. Two things take place during a code inspection session: the programmer narrates the logic of the program statement by statement, and the program is analyzed with respect to a checklist for common programming errors, e.g. computation and comparison errors, and unexplored branches [4]. A checklist of historically common programming errors can be found in [10].

A walkthrough is similar to an inspection but the procedures and error detection techniques are slightly different. During the group meeting, a small set of test cases are walked through the logic of the program by the participants.

Code analysis of the program can be performed with the assistance of a static analyzer. Static analyzers analyze the control and data flow of the program, and record in a database such problems as uninitialized variables, inconsistent interfaces among modules, and statements which can never be executed. Other properties are then inferred from the data base. (See Section 3.2 on Static Analysis Tools.)

EXPERIMENTAL EVALUATION

A major practical limitation of static analysis involves array references and the evaluation of pointer variables. The elements of an array and the data items referenced by a pointer variable cannot be distinguished by static analysis of the code. Symbolic testing (see section 2.2.2) may be employed for evaluation of array references and pointer variables [6].

Experimental evaluation of code inspections and walkthroughs have shown these methods to be effective in finding from 30% to 70% of the logic design and coding errors in typical programs [10]. Myers [11] has found that walkthroughs and code inspections detected an average of 38% of the total errors in the programs studied. Uses of code inspections by IBM [5,12] have shown error-detection rates of about 80% of the total errors. Daily [2] has compared the effectiveness of code inspections and design analysis to other validation methods and estimated that 90% of the errors found by simulator testing can be found by code inspections and design analysis.

REFERENCES

[1] E. M. Boehm, R. K. McClean, and D. B. Urfrig.
Some Experience with Automated Aids to the Design of Large-scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1:125-33, 1975.

[2] E. B. Daily.
Software Development.
Proceedings of European Computing Review, Infotech International, 1978.

[3] M. S. Deutsch.
Software Verification and Validation Realistic Project Approaches.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982, pages 271-3.

[4] R. Dunn and R. Ullman.
Quality Assurance for Computer Software.
McGraw Hill Book Company, New York, 1982, pages 166-8.

[5] M. E. Fagan.
Design and Code Inspections to Reduce Errors in Program Development.
IBM Systems Journal, Vol.15(3):182-211, 1976.

[6] R. E. Fairley.
Tutorial: Static Analysis and Dynamic Testing of Computer Software.
Computer, pages 14-23, April 1978.

[7] M. S. Fuji.
Independent Verification of Highly Reliable Programs.
Proceedings of COMPSAC 77, pages 38-44, IEEE, 1977.

[8] R. L. Glass.
Software Reliability Guidebook.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979, pages 86-95.

[9] W. E. Howden.
A Survey of Static Analysis Methods.
In Tutorial: Software Testing & Validation Techniques, pages 101-15. E. Miller and W. E. Howden, Editors, IEEE, 1981.

[10] G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, New York, 1979.

[11] G. J. Myers.
A Controlled Experiement in Program Testing and Code Walkthroughs/Inspections.
Communications of the ACM, Vol.21(9):760-8, 1978.

[12] M. P. Perriens.
An Application of Formal Inspections to Top-down Structured Program Development.
RADC-TR-77-212, IBM Federal Systems Division, Gaithersburg, MD, 1977, (NTIS AD/A-041645).

[13] Static Analysis Techniques.
In Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography.
Infotech International, 1979, pages 107-23.

[14] D. Teichrow and E. A. Hershey, III.
PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering, Vol.SE-3:41-8, 1977.

2.2.2. SYMBOLIC TESTING

SUMMARY

During symbolic testing, input data and program variable values are given symbolic values. The symbolic values may be elementary symbolic values or expressions. The possible executions of a program are characterized by an execution tree. The execution state of a program consists of the PC, an accumulator of the properties which the inputs must satisfy in order to execute the associated path. The execution is performed by a system called a symbolic evaluator whose major components are a symbolic interpreter and an expression simplifier.

Symbolic execution can be used to prove the correctness of a program. A program may be thought of as a finite set of assertion-to-assertion paths. If each path is shown to be correct, then the program is correct. When a program contains loops, the execution tree may contain infinite branches. Two possible methods for analyzing loops are informal inductive assertions and recurrence relations describing the behavior of each variable affected by the loop. Experimental evaluation results of symbolic execution are given.

INTRODUCTION

Symbolic testing derives its name from the fact that during testing the values of program variables and data are symbolic. The usual approach to testing programs is to execute the program on a set of input values and to examine the output of the program by some external mechanism, most likely by the programmer, to determine the correctness of the program behavior. In contrast to the usual approach, the input values may be "symbolic" constants, and the output values may consist of "symbolic formula" or "symbolic predicates". Thus, "symbolic testing is a natural extension of normal execution, providing the normal computations as a special case" [19].

The symbolic inputs may correspond to a class of actual data values. For example, the symbolic input X might represent an actual data value within the range 1 to 500; therefore, one may think that testing a program once symbolically is equivalent to a large number of tests with actual data.

Symbolic execution of a program can be characterized by an execution tree. It consists of the nodes associated with the statements executed and directed arcs indicating program flow.

The symbolic execution of a program may make proving program correctness easier, if the source program is considered as a set of path conditions which the symbolic data values must satisfy [5,19,20]. Some program errors are easily identified by examining the symbolic output of a program if the program is supposed to compute a mathematical formula. In such a case, the output is checked against the formula to see if they match [15].

Symbolic testing has some difficulties with programs containing either loops or array variables. Also, symbolic execution trees associated with large programs may create problems [20]. These problems are discussed further below. Systems that perform symbolic testing, called symbolic evaluators, are discussed in Section 3.3.2.

Aside from program testing, symbolic execution has another potential use. Current trends in programming methodology and in programming language design have been increasing the need for good code optimization tools. The use of data abstraction and encapsulation facilities have changed the nature of the code optimization. The programmer's attention is directed to improving the clarity and the maintainability of the code rather than the efficiency. Optimizers must therefore detect inefficiencies that might in the past have been ignored by assuming that no "reasonable" programmer would create them [3]. As a result, more reasoning power is needed in optimization tools. The symbolic evaluator may well serve as a basis of such an optimizer.

SYMBOLIC EVALUATION AND TESTING

Symbolic Execution

The symbolic value may be an elementary symbolic value or an expression. An elementary symbolic value is any string that is used by the programmer as the value of a variable. An expression is any combination of numbers, arithmetic operators, and symbolic values. In Figure 1, the symbolic values of the variables of procedure SAMPLE are given.

```
Procedure SAMPLE (X,Y)
S = 2*X + 3*Y
T = S - Y
RETURN
END
```

```
The value of a program variable is denoted as v(variable name)
Assume v(X) = a, v(Y) = b, then S = 2*a + 3*b
T = 2*a + 3*b - b
T is simplified to T = 2*a + 2*b
```

Figure 1. Procedure SAMPLE and the symbolic value of its variables.

The execution of the program starts with the assignment of symbolic values to the variables. The execution of the arithmetic expressions used in the assignment statements and IF statements requires the introduction of a capability for path selection [13,20]. When an assignment statement is executed, the value of the variables on the right-hand side of the statement is substituted into the left-hand side. In order to select a path in an IF statement, the evaluation of a "path condition" (PC) is required.

A PC is a Boolean expression which computes to "true" or "false", such as (a1.GE.0 .AND. a2+a3.GE.0). As indicated in [19], the PC is an accumulator of the properties which the inputs must satisfy in order to execute the associated path. The PC is initialized to TRUE when the execution of the program starts. The execution of an IF statement starts with the evaluation of the Boolean expression. If the result of this evaluation is Q, and the current PC contains Q, the THEN part is taken, otherwise the ELSE part is taken. This type of execution of the IF statement is called "nonforking" execution. If the PC contains neither Q nor .NOT.Q, the symbolic execution forks into two executions; one is assumed to follow the THEN part, the other is assumed to follow the ELSE part, since both alternatives are possible. This type of execution of the IF statement is called a "forking" execution in [19,20]. Each forking execution of the IF statement adds a condition to the PC. The nonforking execution does not cause any change in the PC.

The "execution state" of a program consists of the PC, statement counter, and values of the variables. As we will see below, the execution state of a program is used in the construction of a symbolic execution tree.

Symbolic Execution Tree

The execution of a program is characterized by an execution tree [5,19], sometimes called an evolution tree [4]. It consists of nodes associated with the statements executed and directed arcs indicating program control flow. The current PC is stored in each node. The nodes that are associated with each forking IF statement execution have two outgoing arcs labeled T (true) and F (false) for the THEN and ELSE branches. All other statements have only one outgoing directed arc. In Figure 2, a function procedure ABSOLUTE and its execution tree are given. This example is essential that given in [8].

The execution tree of large programs cannot always be easily processed. An advantage of symbolic testing over conventional testing is that the testing of a program symbolically is equivalent to a large number of tests with actual data. But this may not be an advantage if the program considered is a large one, as the execution tree becomes so large that the entire tree cannot be examined, and the programmer may not

```
1.   ABSOLUTE:
2.   PROCEDURE (X);
3.   DECLARE X, Y INTEGER;
4.   IF X .LT. 0
5.        THEN  Y := -X;
6.        ELSE  Y := X;
7.   RETURN (Y);
8.   END;
```

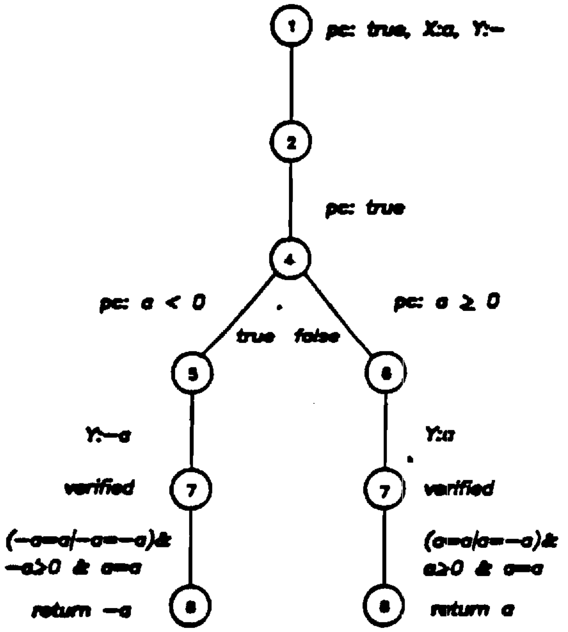


Figure 2. Function Procedure ABSOLUTE and its execution tree.

know which subtrees to examine [20]. Thus, symbolic testing may not provide substantial confidence of correctness in the case of large programs.

Structured programs may be easy to test symbolically, because they can be decomposed into modules. Then the execution tree of each module can be handled separately [20].

If a program is executed normally with a specific set of integers as inputs, one will get the same results when the program is executed symbolically and integers are assigned to the symbolic results [19]. Thus symbolic execution is an extension of the conventional execution.

Symbolic Execution and Program Correctness

Symbolic Execution is used by some researchers to prove program correctness [1,2,3,4,18,19,21]. As indicated by Darringer [20], the program is annotated with assertions at the input, output, and at every loop. Therefore, a program may be thought as composed of a finite set of assertion-to-assertion paths. Correctness of the program is shown by performing the following actions for each path:

1. Assume the assertion at the beginning of the path holds.
2. Execute the statements in the path symbolically.
3. If 1 and 2 implies the end assertion of the path, then the path is correct; otherwise, it is not.

If all the paths are shown to be correct, then the program is said to be "correct".

King [19] uses Floyd's method [7] to prove program correctness. Deutsch independently developed the notion of symbolic execution in exploiting this proof technique in his interactive verifier [6]. In his system, the user indicates the correct path interactively at each program branch; the system then checks the consistency of that choice with the current PC value, and conjoins it to the PC if it is consistent. Inconsistency indicates the presence of an error.

Difficulties associated with this technique for proving program correctness are the creation of assertions and the requirement of human interaction. The automated generation of assertions is possible in only simple cases as indicated in Chapter 1.

When a program contains a loop, another difficulty arises [14]; the number of iterations depends on the value of the loop variable, and the execution tree may contain infinite branches. Hence, since symbolic testing cannot be exhaustive, concluding program correctness by symbolic execution is not always achievable.

Informal inductive loop predicates [5,20] provide an aid in executing the infinite branches of such an execution tree. Symbolic values are assigned to the variables affected by the loop at the body beginning to analyze the loop behavior. By executing the loop once, the observed variable values are used to formulate inductive formal assertions.

Another approach in analyzing the behavior of the loops has been taken by Cheatham, et al [3], in which an iteration counter K is associated with each loop. Then, for a variable whose value may be changed in the loop a function (K), denoting the value of X at the beginning of the K-th cycle, is determined. Secondly X_L, the number of cycles taken by the loop, is determined.

The underlying idea in this approach is to describe the behavior of each variable affected by the loop as a recurrence relation. Associated with each variable whose value might change in the loop body, X_K, a "Possible Induction Value" (PIV in short) is installed at the beginning of the loop body. After executing the body symbolically, the following values are computed:

1. X(K+1): The value of X as X would be affected in the next cycle.
2. The symbolic expression p_j: The value of each exit condition P_J. (It is assumed that explicit exit conditions $P_1,\ldots,P_n$ are given in the loop body.)

In addition to these, the value of X prior to the first cycle of the loop, denoted as X(1), is determined. Then, X(K+1) and X(K) with the boundary value X(1) are treated as a recurrence relation. The solution to this recurrence is the desired value X(K).

Each X_k occurring in exit condition P_j is substituted with its solution X(K) in order to compute X_L. Thus, a solution for P_j, denoted p_j, is obtained. If the upper limit of the range of the loop is given explicitly, then its successor is called lim, otherwise it is called "infinity", a unique value. Then the following formula gives the number of cycles taken by the loop:

$$\text{least}(j,1,\text{lim},p_1(j) \text{ or}\ldots\text{or } p_n(j))$$

The symbolic values of data items are occasionally undefined as when the program contains an array reference, say X(I). The particular element of this array is identified by the value of the variable I. If the value of I is a symbolic expression, one cannot distinguish the elements of the array X. Some possible approaches for solutions to this problem are as follows [19,20]:

1. Exhaustive case analysis as in the case of an unresolved IF statement.
2. Leaving the ambiguity unresolved but preserved by storing conditional values for variables. This is called partial symbolic execution in [20].

EXPERIMENTAL EVALUATION AND RELIABILITY OF SYMBOLIC TESTING

Symbolic execution has been used in several symbolic evaluation and program proof systems such as DISSECT [13], SELECT [1], and EFFIGY [19]. These systems are introduced in Chapter 3. The main uses of this technique have been for generating test data as explained in Section 3.3.3 and for proving program correctness.

The effectiveness of symbolic execution for uncovering program errors has been studied by Howden [9,10,11,12]. He applied DISSECT to the programs in error cited in Kernighan and Plauger [16]. According to his results, 15 out of 22 errors may possibly be detected by symbolic execution.

REFERENCES

[1] R. S. Boyer, B. Elpas, and K. N. Levitt.
SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution.
SIGPLAN Notices, Vol.10(6):234-45, June 1975.

[2] R. M. Burstall.
Proving Correctness as Hand Simulation with a Little Induction.
Proceedings of the International Federation of Information Processing Societies, North Holland, Amsterdam, The Netherlands, 1974, pages 308-12.

[3] T. E. Cheatham, Jr., C. H. Holloway, and J. A. Townley.
Symbolic Evaluation and the Analysis of Programs.
IEEE Transactions on Software Engineering, Vol.SE-5(4):402-17, July 1979.

[4] L. A. Clarke.
A System to Generate Test Data and Symbolicly Execute Programs.
IEEE Transactions on Software Engineering, Vol.SE-2(3):215-22, September 1976.

[5] J. A. Darringer and J. C. King.
Application of Symbolic Execution to Program Testing.
Computer, Vol.11(4):51-60, April 1978.

[6] L. P. Deutsch.
An Interactive Program Verifier.
Ph.D. Thesis, University of California, Berkeley, May 1973.

[7] R. W. Floyd.
Assigning Meanings to Programs.
Proceedings of the Symposium on Applied Mathematics, Vol.19:19-32, American Mathematical Society, Providence, RI, 1967.

[8] S. L. Hantler and J. C. King.
An Introduction to Proving the Correctness of Programs.
ACM Computing Surveys, Vol.8(3):332-53, September 1976.

[9] W. E. Howden.
An Evaluation of the Effectiveness of Symbolic Testing.
Software Practice and Experience, Vol.8(4):381-97, July-August 1978.

[10] W. E. Howden.
Experiments with a Symbolic Evaluation System.
National Computer Conference, AFIPS Proceedings, June 1976, pages 899-908.

[11] W. E. Howden.
Reliability of Symbolic Evaluation.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 442-7.

[12] W. E. Howden.
Symbolic Testing - Design Techniques, Costs and Effectiveness.
NBS Report GCR77-89, National Bureau of Standards, Springfield, VA, 1977, (NTIS PB268517).

[13] W. E. Howden.
Symbolic Testing and the DISSECT Symbolic Evaluation System.
IEEE Transactions on Software Engineering, Vol.SE-3(4):266-78, July 1977.

[14] J. C. Huang.
An Approach to Program Testing.
ACM Computing Surveys, Vol.7(3):113-28, September 1975.

[15] Infotech State of the Art Report, Software Reliability, Volume 1: Analysis and Bibliography.
Infotech International, 1977.

[16] B. W. Kernighan and P. J. Plauger.
The Elements of Programming Style.
McGraw Hill Book Company, New York, 1974.

[17] J. C. King.
A New Approach to Program Testing.
SIGPLAN Notices, Vol.10(6):228-33, June 1975.

[18] J. C. King.
Proving Programs to be Correct.
IEEE Transactions on Computers, Vol.C-20(11):1331-6, November 1971.

[19] J. C. King.
Symbolic Execution and Program Testing.
Communications of the ACM, Vol.19(7):385-94, July 1976.

[20] Symbolic Testing.
In Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography, pages 125-137.
Infotech International, 1979.

[21] R. W. Topor and R. M. Burstall.
Verification of Programs by Symbolic Execution - Progress Report.
Unpublished Report, Department of Machine Intelligence,
University of Edinburg, Scotland, December 1972.

2.2.3. PROGRAM INSTRUMENTATION

SUMMARY

Programs can be instrumented by recording processes that do not affect the functional behavior, but record properties of the executing program. Additional output statements, assertion statements, monitors, and history-collecting subroutines may be used to instrument a program. There exist various tools to automatically insert these into a program. Experimental evaluation of dynamic assertions indicates that programmers who are familiar with the use of assertions can decrease the debugging time for complex programs.

INTRODUCTION

In program instrumentation, each time the software under test performs a significant event, the occurrence of that event is recorded. The nature of the recording process depends on the measurements desired and the type of event performed [13]. Ramamoorthy [11] describes a method to place a minimum number of such processes into a program and still adequately instrument the program. The events recorded may be the range of some particular variables, the number of times some statements are executed, or whether a condition on a statement is violated. The recording processes are sometimes called "probes", "monitors", or "software instruments" [6,7].

One type of recording process is the "history-collecting subroutine" in which the number of events recorded may vary. In a system implemented by Fairley [3], a program is automatically instrumented with history-collecting subroutines by an instrumenting compiler. Execution of an instrumented program on test cases selected produces a history of the behavior of a program which is recorded in a data base. The behavior information of a program can then be used to construct new sets of test data and to determine the compatibility of the program's behavior with its specifications. For other types of program instrumentation tools see the section on Dynamic Analysis Tools, Section 3.3.

In an interpretive system no history of the program behavior is available since instrumenting interpreters generally display only the current behavior of a program.

THE ASSERT STATEMENT

A well-known method of program instrumentation is the manual insertion of additional output statements into a program. Once a program is completely debugged, it is necessary to remove these statements. This may be tedious and time consuming in the case of large programs [6]. Thus, the idea of using special recording processes, called assertions, in the program has been developed.

The general format of the assertion statement found in some programming languages is as follows:

ASSERT boolean expression statement.

Semantically, if the boolean expression evaluates to FALSE, then the associated statement is executed and the program is terminated [9]. Some programming languages have built-in assertion statements, e.g. PLAIN and EUCLID. In other languages a pre-processor, called the "dynamic assertion processor", implements the assertions inserted by the user in the form of a comment. By recompiling the output of the pre-processor, these comments can be removed from the program [6]. Monitors are assertions that check whether the value of a variable is within the range specified by the assertion. The Program Evaluator and Tester system, PET [12,13] is an automated tool implementing MONITOR and ASSERT commands.

In conjunction with testing, program instrumentation can be used to measure the coverage of program paths. However, this usage is usually limited to decision paths (DDP's) [10]. The program Test Coverage Analyzer instruments each DDP by counting the number of times it is executed [4]. DDP analysis is available at the commercial level [2,15].

EXPERIMENTAL EVALUATION

Examples of simple errors that cannot be detected by the DDP method have been reported [5]. Performance evaluation of dynamic assertions has been performed by L.G. Stucki [14] and J.M. Adams [1]. These results indicate that programmers who are familiar with the use of assertions can significantly decrease debugging time for complex programs.

INSTRUMENTING COMPILERS AND INTERPRETERS

A program can be automatically instrumented with history-collecting subroutines by an instrumentating compiler to record statement execution counts, ranges of variables, and timing estimates such as CPU time and relative time in each routine. A computation state of a program at any point in an execution is determined by the values of currently accessible variables and the control flow information necessary to continue execution from that point. An execution history, the sequence of changes in the computational states of an instrumented program, is recorded in a database as the program executes on the selected test cases. The entire execution history of the program can be analyzed after its termination [3].

Instrumenting interpreters maintain a current computation state and update that state at each step as the execution advances. Thus, no history of execution is available [3].

The execution history information can be used to construct new sets of test cases and to determine if a program functions correctly with respect to its specification [8]. An advantage of having a history execution available in a database is that it is possible to perform analysis and history collecting seperately. This enables one to use existing batch compilers, loaders, and library routines. It is also possible to observe how a particular computation is influenced by the previous ones [3,8].

The disadvantages of analyzing the program behavior from a database are that the noise introduced by history collecting subroutines prevents one from obtaining appropriate timing information, and the execution history of a program is potentially large in size. To limit the execution history size one either selects some regions of a program and events to be recorded in these regions or carefully constructs test cases and integrates a top-down and bottom-up approach into the instrumenting compiler (see Section 2.1). Furthermore, unlike the interpretive systems, users can make a change beyond the current statement in a program by stopping execution and observing the effects of the change by continuing the execution from the halt point [3,8].

REFERENCES

[1] J. M. Adams.
Experiments on the Utility of Assertions for Debugging.
Proceedings Eleventh Hawaii International Conference on System Science, Honolulu, HI, January 1978, pages 31-9.

[2] Automated Testing Analyzer for Cobol.
Software Technology Center, Science Applications, Inc., San Francisco, CA, April 1976.

[3] R. E. Fairley.
Tutorial: Static Analysis and Dynamic Testing of Computer Software.
Computer, pages 14-23, April 1978.

[4] R. L. Glass.
Software Reliability Guidebook.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

[5] J. B. Goodenough and S. L. Gerhart.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering, Vol.SE-1(2):156-73, June 1975.

[6] J. C. Huang.
Program Instrumentation.
In Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography, pages 144-50.
Infotech International, 1979.

[7] J. C. Huang.
Program Instrumentation: A Tool for Software Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 147-59.
Infotech International, 1979.

[8] Infotech State of the Art Report, Software Testing Volume 1: Analysis and Bibliography.
Infotech International, 1979.

[9] G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

[10] M. R. Paige.
Program Graphs, an Algebra, and Their Implication for Programming.
IEEE Transactions on Software Engineering, Vol.SE-1:286-91, September 1975.

[11] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen.
Optimal Placement of Software Monitors Aiding Systematic Testing.
IEEE Transactions on Software Engineering, Vol.SE-1(4):403-11, December 1975.

[12] L. G. Stucki and G. L. Foshee.
New Assertion Concepts for Self-Metric Software Validation.
Proceeding of the 1975 International Conference on Reliable Software, April 1975.

[13] L. G. Stucki.
New Directions in Automated Tools for Improving Software Quality.
In Current Trends in Programming Methodology, Volume II: Program Validation, R. T. Yeh, Editor, pages 80-111.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

[14] L. G. Stucki.
Tutorial on Program Testing Techniques.
Slide masters, COMPSAC-77, November 8-11, 1977, Chicago, IL.

[15] Fortran Automated Verification System (FAVS), Volume 1. User's Manual.
General Research Corporation, Santa Barbara, CA, January 1979.

2.2.4. PROGRAM MUTATION TESTING

SUMMARY

Program mutation is a technique for the measurement of test data adequacy. Test adequacy refers to the ability of the data to insure that certain errors are not present in the program under test. In mutation testing, test data is applied to the program being tested and its "mutants" (i.e., programs that contain one or more likely errors). If a program passes a mutation test, then either the program is correct or it contains an improbable error.

Weak mutation testing and trace mutation testing are variations of mutation testing. Experimental evaluation of mutation testing indicates that the results of mutation testing are good predictors of operational reliability.

INTRODUCTION

Program mutation is a technique for measuring of test data adequacy. Recall from Chapter 1 that a test data set is adequate if the program runs successfully on the data set and if all incorrect programs run incorrectly. Furthermore, a test data set is adequate relative to a set of programs A if the program under test runs correctly and all incorrect programs in A fail on at least one point in the test set.

In mutation testing, the set A in the definition of relative adequacy is taken to be a set of programs which are "close" to the program being tested. In this context, "close" refers to the potential errors which could have occurred in the program being tested: either the program under test is error-free or it contains one or more of the most likely errors. The test data should be strong enough to distinguish a correct program P from those versions of P that contain the most likely errors.

More precisely, suppose that a correct program P is to be tested using test data D. A set of mutants of P consists of a set of programs which differ from P in containing a single error chosen from a given list of error types. Call this set M(P). Some of the mutant programs in M(P) will turn out to be (functionally) equivalent to P -- that is, they will be indistinguishable from P under all test data. For example, the "error" that replaces the GO TO statement in the Fortran subroutine shown below by a RETURN has no effect on the logic of the subroutine, and therefore, gives rise to an equivalent mutant.

```
      SUBROUTINE F(X,Y,Z)                          SUBROUTINE F(X,Y,Z)
      IF (X.EQ.0) GO TO 10                         IF (X.EQ.0) RETURN
      Z=(X+Y)/X                    ====>           Z=(X+Y)/X
 10   RETURN                                 10    RETURN
      END                                          END
```

Let E(P) represent the set of equivalent mutants of P. Finally, when the programs in M(P) are executed on D, some will return results which differ from the results which P delivers on D. Call this set of mutants DM(P,D).

A mutation score is defined to the fraction of the nonequivalent mutants of P which are distinguished by the test set D. That is, if m, e, and dm represent the number of elements in M(P), E(P), and DM(P), respectively, then the mutation score of D and P is defined:

$$ms(P,D) = dm/(m-e).$$

A mutation score is a number in the interval [0,1]. A high score indicates that D is very close to being adequate for P relative to the set of mutants of P. A low score indicates a weakness in the test data of the following kind: the test data does not distinguish P from the program P', which contains an error. Mutation scores can be calculated automatically once a method for determining the mutants has been defined.

Mutation testing is an error-based testing technique. In error-based testing, the goal is to construct test cases that reveal the presence or absence of specific errors. Error based testing is present in nearly all heuristic approaches to testing. For example, informal debugging sessions frequently include checks on extreme values of variables. In addition, the folklore of many applications areas consist of heuristic rules (e.g., in testing compilers, one of the first test cases tried is usually the "null" program). More formalized attempts at error-based testing include Howden's study of errors in algebraic programs [19], the revealing subdomain strategy [25], the strategies for uncovering domain errors [26], and the Budd-Miller study of typographical errors in numerical software [8].

FORMS OF MUTATION TESTING

In many respects, mutation testing is to software what fault analysis is to digital circuits. In digital circuit testing, the errors to be eliminated are drawn largely from experience and physical theory of how circuits are most likely to fail. In mutation testing the errors result from logical failures in a program rather than failures of physical components.

EVALUATION OF MUTATION TESTING

Experimental and theoretical evaluation are available in [1,2,7,9]. Principle results for mutation testing have centered around the complexity of mutation testing and the coupling effect. The results for weak mutation testing have deal with complexity and example of useful test cases. For trace mutation testing [5], the key results have been theoretical.

The coupling effect has been investigated from a number of points of view. Just as in the case of digital circuit faults, there are restricted programming languages for which error coupling can be proved mathematically. In addition, there is a body of experimental evidence supporting some sort of error coupling. In [7], single subject experiments were performed to determine whether errors can be detected. In [1,2], statistical results support with high confidence the coupling of simple and complex errors.

The complexity of mutation testing has been examined experimentally. In [1], a number of observations of mutation complexity are derived. Furthermore, the weak mutation metric introduced by Howden [20] provides still more information on the expected cost of mutation. The expected cost of a mutation score calculation is determined by the number of mutants executed and the expected running time of a mutant program. This is not the same as the running time of a program under test, since the running time of a program with n errors is $E(t) = c/n$. The expected number of mutants depends on the set of error operators, but for a typical Fortran system, this number is approximately the number of distinct variable names times the number of variable references.

Practical experience with mutation systems has led some investigators to consider heuristics for speeding up the calculation of mutation scores (such as limiting the number of mutation operators [24] and random sampling of mutants [1,2]).

REFERENCES

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward.
Mutation Analysis.
Report GIT/ICS-79-08, Georgia Institute of Technology, 1979.

[2] A. T. Acree.
On Mutation.
Ph.D. Thesis, Georgia Institute of Technology, 1980.

[3] V. K. Agarwall and G. M. Masson.
Recursive Coverage Projection of Test Sets.
IEEE Transactions on Computers, Vol.C-28(11): 865-70, November 1979.

[4] D. Baldwin and F. Sayward.
Heuristics for Determining Equivalence of Program Mutations.
Technical Report 161, Yale University, 1979.

[5] M. Brooks.
Testing, Tracing, and Debugging Recursive Programs Having Simple Errors.
Ph.D. Thesis, Stanford University, 1980.

[6] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
The Design of a Prototype Mutation System for Program Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:623-27, 1978. Also reprinted in Tutorial: Automated Tools for Software Engineering, E. F. Miller, Editor. IEEE Computer Society, 1979.

[7] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Program.
7th ACM Symposium on Principles of Programming Languages, January 1980.

[8] T. A. Budd and W. C. Miller.
Detecting Typographical Errors in Numerical Programs.
University of Arizona, Tuscon, AZ, 1982.

[9] T. A. Budd.
Mutation Analysis of Program Test Data
Ph.D. Thesis, Yale University, 1980.

[10] T. A. Budd.
Mutational Analysis: Ideas, Examples, Problems and Prospects.
In Computer Program Testing, B. Chandrasekaran and S. Radicchi, Editors. North-Holland, 1981.

[11] J. Burns.
Stability of Test Data from Program Mutation.
Digest for the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, FL, 1978, pages 324-334.

[12] R. A. Demillo.
Mutation Analysis as a Tool For Software Quality Assurance.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Hints on Test Data Selection: Help for the Practicing Programmer.
Computer, Vol.11(4):34-41, April 1978.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Program Mutation: A New Approach to Program Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 107-26.
Infotech International, 1979.

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Program Mutation as a Tool for Managing Large-Scale Software Development.
1978 ASQC Technical Conference Transactions, American Society for Quality Control Engineers, Chicago, 1978.

[16] R. A. DeMillo, D. Hocking, and M. J. Merritt.
A Comparison of Some Reliable Test Data Generation Procedures.
Report GIT/ICS-81-08, Georgia Institute of Technology, 1981.

[17] J. Gourlay.
Theory of Testing Computer Programs.
Ph.D. Thesis, University of Michigan, 1981.

[18] J. M. Hanks.
Testing Cobol Programs by Mutation: Volume I - Introduction to the CMS.1 System, Volume II - CMS.1 System Documentation.
Report No. GIT/ICS-80-04, Georgia Institute of Technology, 1980.

[19] W. E. Howden.
Algebraic Program Testing.
Acta Informatica, Vol.10, 1978.

[20] W. E. Howden.
Weak Mutation Testing and Completeness of Test Sets.
IEEE Transactions on Software Engineering, Vol.SE-8(4):371-79, July 1982.

[21] R. J. Lipton and F. G. Sayward.
The Status of Research on Program Mutation.
Digest for the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, FL, 1978, pages 355-73.

[22] S. C. Ntafos.
On Required Element Testing.
Proceedings of COMPSAC 81.

[23] D. L. Ostapko and S.-J. Hong.
Fault Analysis and Test Generation for Programmable Logic Arrays (PLA's).
IEEE Transactions on Computers, Vol.C-28(9):617-27, September 1979.

[24] I. J. Riddle, J. A. Hennel, M. R. Woodward, and D. Hedley.
Practical Aspects of Program Mutation.
University of Nottingham, Nottingham, UK.

[25] E. J. Weyuker and T. J. Ostrand.
Theories of Program Testing and the Application of Revealing Subdomains.
IEEE Transactions on Software Engineering, Vol.SE-6(3):236-46, May 1980.

[26] L. White and E. A. Cohen.
A Domain Strategy for Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-6, 1980.

[27] M. R. Woodward, M. A. Hennell, and D. Hedley.
A Limited Mutation Approach to Program Testing.
University of Nottingham, Nottingham, UK, 1980.

2.2.5. INPUT SPACE PARTITIONING

SUMMARY

A path in a program consists of some possible flow of control. In path analysis testing techniques the input space of a program is partitioned into path domains: those subsets of the program input domain that cause execution of each path. The program is then executed on test cases that are constructed by selecting test data from these domains.

Path analysis is meant to detect computation, path, and missing path errors. Domain testing detects many path selection errors by selecting test data on and near the boundary of a path domain. In partition analysis, the specification of a program written in a formal specification language is partitioned into subspecifications. This partition is then intersected with that on the input space to obtain a new partition, whose elements are called "procedure subdomains", specifying a set of input data for which a subspecification and a path are applicable. Then the test cases are constructed by selecting some test data from each procedure subdomain. One major problem of these techniques is that current technology limits their use to programs which have a small number of input variables since the number of required test cases is exponential in the number of input variables.

INTRODUCTION

A path in a program consists of some possible flow of control. The paths in a program partition the input space of the program into path domains: those subsets of the input domain of the program that cause execution of each path. Each path corresponds to a path domain and a path computation function. The path computation function is the function computed on the input domain by the execution of the statements along the path. Conditional branches on a path determine the boundary of the path domain. Symbolic evaluation of branch predicates can be used to construct path domains and paths in terms of input variable values [2,3,4,8,10].

The underlying idea of the path analysis testing techniques is to partition the program input space into path domains. The program is then executed on test cases that are constructed by choosing test data from these domains. Partition analysis utilizes the formal specification of the program to produce a similar partition that can be compared to that determined by the program [5].

PATH ANALYSIS AND TESTING

Path analysis testing involves the selection of test data to execute chosen paths. A test case is constructed by choosing one test point from each path domain so that each path through a program is executed at least once. In practice, a program may contain an infinite number of paths. Thus, a practical path analysis testing strategy has to use a procedure to select a subset of the total set of paths [4]. Woodward suggests a hierarchy of structural test metrics to guide the choice and monitor the coverage of test paths [9].

Howden [4] classifies the types of errors that path analysis is meant to detect: computation, path selection, and missing path errors. A computation error occurs when a computation statement along a path is computed incorrectly. Path selection errors occur when the branching predicates are incorrect, and missing path errors are those in which the required branch predicate does not exist in a program.

The general problem of determining the paths of a program and the selection of test data to execute the chosen paths is intractable. However, by restricting the features of the language in which the programs are written, it is possible to select finite subsets of test data for the chosen paths to detect certain types of errors [8,10].

DOMAIN ANALYSIS

Domain testing detects many path selection errors by selecting test data on and near the boundary of a path domain. One underlying assumption of domain testing is that a test oracle is available which determines if the execution of a program on the selected test data produces correct output. Other assumptions are such that coincidental correctness cannot occur, the input space is continuous, and the predicate interpretations are simple linear inequalities [2,8].

Two domain testing strategies have been proposed by White and Cohen [8], and by Hassell, et al [2]. In testing the programs in two dimensions, that is with two input variables, White and Cohen propose to select two test points ON and one test point slightly OFF of the given closed border. Thus, each OFF point is in the adjacent domain. The two ON points are chosen close to the ends of the given border. The OFF point is chosen so that its projection lies in between the two ON points. This 2 by 1 strategy can be extended to N dimensions to give an N by 1 strategy [8].

Alternatives to the 2 by 1 and N by 1 domain testing strategies are two ON - two OFF (2 by 2), and its generalizations N by N and E by E, where E is the number of edges of the given border. In the 2 by 2 strategy, two ON points are chosen as in [8], two OFF points are chosen so that one is at each end of the border being tested. The 2 by 2 strategy does not detect all path selection errors, but it may detect more shifts than the 2 by 1 strategy. Of these strategies, the E by E strategy provides guidance for choosing the best set of test data and is completely sensitive to changes in the path domain shape, but requires the largest number of test cases [2].

The major drawbacks of domain testing are its limitation to simple linear predicates, and the difficulty of selecting test cases for a program which has large number of input variables. Moreover, domain testing concentrates on path selection errors; therefore, other testing methods must be employed to thoroughly test a program. The domain testing strategies can be modified to handle equality and nonequality predicates, but in order to handle a wider range of applications, the linearity assumption must be dropped [2].

PARTITION ANALYSIS

In partition analysis, the specification of a program is assumed to be correct and to be described in a formal specification language. This specification is at such a low level that it is almost itself a program. The input domain and the specification of a program are then partitioned into path domains and subspecifications respectively by using a symbolic evaluation technique. These two partitions are then intersected to get a new partition specifying a set of input data for which a subspecification and a path are applicable. The elements of this new partition are called "procedure subdomains" [5].

The process of examining these partitions to determine how closely an implementation and specification agree is called partition analysis verification. In partition analysis verification, the implementation and the specification are checked first for input-output compatibility to see if they have the same number and type of inputs and outputs, and that the inputs come from the same domain. Once input-output compatibility is established, symbolic representations of the subspecification and path computation, for each procedure subdomain, are compared.

Partition analysis employs symbolic testing to establish the computational equivalence of a subspecification and a subdomain and their corresponding computations. Equivalence of symbolic representations of all subspecifications and path computations over their associated procedure subdomains implies the correctness of implementation with respect to the specification. Inequality of a subspecification and a subdomain, or of their corresponding computations, probably indicates the presence of an error in the implementation. Furthermore, when consistency cannot be determined (as the problem is undecidable [4]), testing is necessary to attempt to show either the equivalence of the computations or the presence of errors. This is carried out in partition analysis testing, in which test data are constructed by choosing one or more test points from each procedure subdomain [5].

Partition analysis testing may employ the techniques of domain analysis and extremal and special values testing to show the existence of missing path and the computation errors, respectively (for extremal and special values see Functional Program Testing, Section 2.2.6). A missing path error which occurs when a subspecification is forgotten in the implementation will likely be detected by partition analysis testing, because each subspecification is contained in some procedure subdomain in the partition analysis [5].

In related work [6], a similar comparison of the implementation and specification partitions is discussed, but the problem of obtaining the specification partition is not addressed. In this approach, after taking the intersection of these two partitions, the user refines each subset of this intersection by considering the classes of errors which intuitively are likely to occur in that subset, and chooses data from each domain in the refined partition [6].

EXPERIMENTAL EVALUATION

In an experiment performed by Howden, path analysis testing revealed the existence of nine out of twelve computation errors, one out of three path selection errors, but it could not detect a missing path error [4]. Domain analysis has been examined for programs with two input variables [8] and has been evaluated for a program with three input variables [1]. Partition analysis was applied to a sample program in [5] and it showed the existence of a fairly subtle error in this program. It has been examined for three programs in [6] and it enabled the detection of some errors in these programs.

The testing techniques mentioned in this section are based on path analysis. They try to overcome the problems of path analysis by considering information other than the program itself to construct test cases. But, they have their own drawbacks. Major problems associated with path analysis are that in practice a program may contain an infinite number of paths, determining the domains may not be possible, and it does not detect all of the path selection, computation, or missing path errors. Domain analysis imposes certain restrictions on a language in which the program is written. For all restricted classes of programs, domain testing detects many path selection errors. Partition analysis considers the specification of a program, and the specification is assumed to be correct. In practice, a specification is incomplete and contains errors; both faults undermine the reliability of partition analysis.

REFERENCES

[1] R. A. DeMillo, E. D. Hocking, and M. J. Merritt.
A Comparison of Some Reliable Test Data Generation Procedures.
Report GIT/ICS-81-08, Georgia Institute of Technology,
April 1981.

[2] J. Hassell, L. A. Clarke, and D. J. Richardson.
A Close Look at Domain Testing.
IEEE Transactions on Software Engineering, Vol.SE-8(4):380-90,
July 1982.

[3] W. E. Howden.
Methodology for the Generation of Program Test Data.
IEEE Transactions on Computers, Vol.C-24(5):208-14, May 1975.

[4] W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions on Software Engineering, Vol.SE-2(3):208-14,
September 1976.

[5] D. J. Richardson and L. A. Clarke.
A Partition Analysis Method to Increase Program Reliability.
Proceedings of the 5th International Conference on Software
Engineering, pages 244-53. IEEE, 1981.

[6] E. J. Weyuker and T. J. Ostrand.
Theories of Program Testing and the Application of Revealing
Subdomains.
IEEE Transactions on Software Engineering, Vol.SE-6(3):236-46,
May 1980.

[7] L. J. White, F. C. Teng, H. Kuo, and D. Coleman.
An Error Analysis of the Domain Testing Strategy.
Technical Report 78-2, Computer Information Science Research
Center, Ohio State University, Columbus, September 1978.

[8] L. J. White and E. K. Cohen.
A Domain Strategy for Computer Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-6(3):247-57,
May 1980.

[9] M. R. Woodward, D. Hedley, and M. A. Hennell.
Experience with Path Analysis and Testing of Programs.
IEEE Transactions on Software Engineering, Vol.SE-6(3):278-85,
May 1980.

[10] S. J. Zeil and L. J. White.
Sufficient Test Sets for Path Analysis Testing Strategies.
Proceedings of the 5th International Conference on Software
Engineering, March 9-12, 1981, San Diego, CA, pages 184-91.

2.2.6. FUNCTIONAL PROGRAM TESTING

SUMMARY

In functional program testing, the design of a program is viewed as an abstract description of its design and requirements specifications. Function and data abstractions are used as guides to identify the abstract functions of a program and to generate the functional test data respectively.

Functional testing requires the specification of domains of each input and output variable for a program. Extremal and special values are the most important values in the domain of a variable. 38 out of a collection of 42 errors known in advance were discovered during the study of the errors that occurred in a release of a major software package.

FUNCTION, DATA ABSTRACTIONS AND TEST DATA SELECTION

Functional program testing is a design-based approach to program testing in which the design of a program is viewed as an abstract description of its design and requirements specifications [5].

There are two steps in functional testing. The first step involves the decomposition of the program into functional units, guided by functional abstraction methods used to design the program. The second step involves the generation of test data used to test the functional units independently, using data abstraction as a guide.

Function abstraction is a program design strategy in which programs are viewed as a heirarchy of abstract functions. This hierarchy is used to identify the functions to be tested. A function at one level of this hierarchy is defined from the functions at a lower level. Figure 1, taken from [5], describes the function abstraction of the text-string processor in [2]. The input to the program is a string of text consisting of aphabetic characters (a's), blanks (b's), new-line indicators (n's), and a string termination character (t). The output is the text printed on separate lines. The maximum length of the output lines is specified in advance. The n characters in the input string do not necessarily indicate the end of the output line and are treated as blanks. The maximum length of a word is the length of a line and no word should be broken up between lines. Each line may contain as many words as possible.

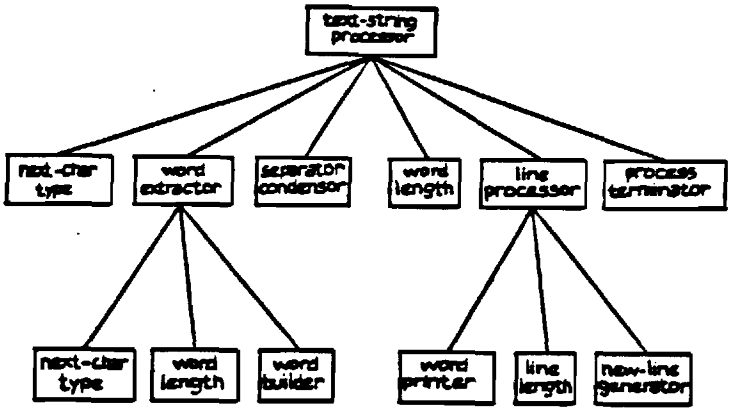


Figure 1. Function abstraction design for text processor.

There are two kinds of functions which may be implemented: requirements functions and design functions. Requirements functions describe the expected behavior of a program and can be identified by examining the requirements specifications of the program, e.g., the text-string processor in Figure 1. A requirements function for a program may require the invention of some other functions called design functions. They are called "general design functions" or "detailed functions" according to the generality of the behavior they implement. Word Extractor and Word Length are general and detailed design functions respectively in Figure 1 [5].

In data abstraction, the structure of data may be modeled by a diagram as a hierarchy of abstract data types, each of which is a "set of values". The set of allowable values for each input and output variable for a program is specified by using data abstraction as a guide. The set of all possible values of a variable is called the domain of the variable [3].

Extremal and special values are the most important values in the domain of a variable. Extremal values are the ones that lie on the edges of a variable domain interval. Special values have special mathematical properties, e.g., zero, one, a very small value, a very large value.

The domain of a numeric variable is usually either a finite set of discrete points or an interval of the form [a,b]. In the former case, numeric values at discrete points are considered as the extremal values. In the latter, a and b are the extremal values. Some examples of extreme value test cases for the text-string processor of Figure 1 are $\underline{n}$, $\underline{b}^k$, and $\underline{b}^k\underline{n}$, for 2.LT.k.

The combinations of values in the domain of a variable create a combinatorial explosion problem; if a program has m input/output variables, each of which can take on k values, then there are k^m possible combinations of values [3]. In order to avoid this problem, less exhaustive "test set" combination rules are used. Howden [5] describes an informal procedure to construct a sequence of test sets which identifies the important classes and combinations of test data, using less exhaustive test set combination rules.

EXPERIMENTAL EVALUATION

Functional testing based on state-of-the-art design analysis techniques was developed by Howden during a study of the errors that occurred in a release of a major software package [4]. 38 out of a collection of 42 errors known in advance were discovered by functional testing. Twenty of the errors were associated with requirements functions, nine with general design functions, and nine with detailed design functions. The use of data abstractions was critical to the discovery of four errors for which functional testing was effective.

Function and data abstraction are generic methods; they may be employed with other software engineering methodologies. For example, structured design [7] and the Jackson design methodology [6] are function and data abstraction approaches to design. The functional testing approach which uses both data and function abstraction as guides may also be used with the SREM requirements analysis system for real-time systems [1], as described in [5].

REFERENCES

[1] M. W. Alford.
A Requirements Engineering Methodology for Real-time Processing Requirements.
IEEE Transactions on Software Engineering, Vol.SE-3:60-8, 1977.

[2] J. Goodenough and S. L. Gerhart.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering, Vol.SE-1(2):156-73, June 1975.

[3] W. E. Howden.
Functional Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-6(2):162-9, March 1980.

[4] W. E. Howden.
An Analysis of Software Validation Techniques for Scientific Programs.
Report No. DM-171-IR, Department of Mathematics, University of Victoria, March 1979.

[5] W. E. Howden.
Functional Testing and Design Abstractions.
The Journal of Systems and Software, Vol.1(4):307-13, January 1980.

[6] M. Jackson.
Principles of Program Design.
Academic, London, 1975.

[7] E. Yourdan and L. L. Constantine.
Structured Design.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

2.2.7. ALGEBRAIC PROGRAM TESTING

SUMMARY

In algebraic program testing, program correctness may be thought of as a program equivalence problem. Since the equivalence of two programs written in a powerful enough programming language is undecidable, in this approach programs are restricted to lie in some restricted class for which testing on a small set of test data is sufficient to prove program equivalence. Applications of algebraic program testing to a restricted class of array manipulation programs, classes of multinomials, and a class of recursive mathematical subprograms are given. Therefore, algebraic program testing provides a theoretically sound way of determining program correctness for restricted classes of programs.

PROGRAM EQUIVALENCE TESTING

In algebraic program testing, testing is used to show the correctness of a program P by showing its equivalence to a "hypothetical correct version Q" of P. Since the equivalence problem for powerful programming languages is undecidable, the power of the language must be restricted, e.g., removing the branch statements and allowing only integer type variables. Thus, in this approach, programs are restricted to lie in some class for which testing on a small set of test data is sufficient to prove the program equivalence.

When given a program P, in the approach taken by [4], P's correctness is shown by testing as follows: A class of programs P^* and associated sets of test cases T^* are defined by a set of properties. The definition of P^* requires that the programmer know certain computational properties of the "correct program" in P^* which we denote by Q. The definition of T^* requires that the programmer have a good understanding of the execution properties of Q. If P and Q generate identical symbolic outputs for any T in T^* then they are computationally equivalent.

There are two problems in this approach. The first is that it is difficult to define sufficiently general classes of programs P^* and related classes of test cases T^*. The second problem is related to the assumptions about the properties of Q that the programmer should know. Those assumptions about Q must be made before using the method to prove the correctness of P. There are two kinds of assumptions that must be made about Q. The first is that there is a correct Q with the properties of the class P^*. The second is that it is possible to generate sets of test cases in T which satisfy the computational properties of Q [4].

EXPERIMENTAL EVALUATION

Algebraic program testing has been applied in [4] to a program which carries out polynomial division. This program contains arrays and variables for indexing these arrays. There are two types of operations carried out in this program: operations on array elements and operations on the array indices. Programs involving these types of operations are referred to as array manipulation programs.

A class of array manipulation programs is defined in which all loops are FOR-loops, programs do not contain branch statements and the only variable types are computational and index types. The class is further restricted to create P^* by limiting the computational assignments, array index expressions and loop bounds, e.g., all loop bounds in FOR-loops are linear functions of the input index variables and of the loop indices of containing loops.

Some restrictions on T^* are then the number of times a loop is executed by the data and the algebraic structure of a set of test cases T, e.g., the union of the values of each test case in T form a cascade set [1] of degree 2.

Computational equivalence of the programs P_* and Q in the restricted class of array manipulation programs P^*, with identical output for any T in T^*, is proven by a "Computational Equivalence Theorem". It is assumed in the proof of the theorem that the programs P and Q generate identical symbolic outputs.

As a further indication of the limitations of the restricted class of array programs, the outputs of the programs in this class contain structural information about the programs.

Some of Howden's ideas have been investigated independently by Geller [3], and some other algebraic results which are not used in "Computational Equivalence Theorem" have been proven by [5].

DeMillo and Lipton [2] have extended Howden's work to a restricted class P^* in which programs compute multinomials with certain limitations on their degree and number of variables. They have shown that a given program P can be shown equivalent to Q in P^* by testing on small sets of test data if some (small) probability of error is allowed.

Rowland and Davis have applied algebraic program testing to classes of polynomials, multinomials, and rational functions [6] and to a class of recursive mathematical subroutines [7]. Applications of these results assume that both functions are in one of the classes mentioned above. Test data are then chosen that uniquely identify members of that class [6,7].

REFERENCES

[1] B. F. Caviness.
On Canonical Forms and Simplification.
Ph.D. Thesis, Carnegie-Mellon University, 1968.

[2] R. A. DeMillo and R. J. Lipton.
A Probabilistic Remark on Algebraic Program Testing.
Information Processing Letters (Netherlands), Vol.7(4):193-5, June 1978.

[3] M. Geller.
Test Data as an Aid in Proving Program Correctness.
Proceedings of Second Symposium on Principles of Programming Languages, pages 209-218. ACM Publications, New York, 1976.

[4] W. E. Howden.
Algebraic Program Testing.
Acta Informatica (Germany), Vol.10(1):53-66, 1978.

[5] W. E. Howden.
Elementary Algebraic Program Testing Techniques.
Computer Science Technical Report 12, Applied Physics and Information Sciences, University of California, San Diego, CA, 1976.

[6] J. H. Rowland and P. J. Davis.
On the Use of Transcendentals for Program Testing.
Journal of the Association for Computing Machinery, Vol.28(1):181-90, January 1981.

[7] J. H. Rowland and P. J. Davis.
On the Selection of Test Data for Recursive Mathematical Subroutines.
SIAM Journal Computers, Vol.10(1):59-72, February 1981.

2.2.8. RANDOM TESTING

SUMMARY

Random Testing is essentially a black-box testing strategy in which a program is tested by randomly choosing a subset of all possible input values. The distribution may be arbitrary, or may attempt to accurately reflect the distribution of inputs in the application environment.

SAMPLING STRATEGY

Random testing is essentially a black-box testing strategy in which a program is tested by randomly selecting some subsets of all possible input values. Since the results compiled by Duran and Ntafos [1] indicate that random testing can be cost-effective for many programs, it may be employed to generate test data for real-time software.

Program testing may be viewed as sampling for errors, i.e., a program is executed for a subset of input data and errors, if any, are detected by observed failures of the expected behavior of a program. Most testing techniques aim at increasing the probability that given sampling instances reveal (existing) errors. Currently there does not exist a technique which can assure that a tested program will perform correctly. A measure of program correctness is "the proportion of elements in the program's input domain for which it fails to execute correctly". This measure is directly related to the test results, since a test case either succeeds or fails. The number of failures in a set of test cases is related to the measure mentioned via some probability distribution function P. P depends on the way one chooses test data.

Test data may be chosen randomly or by a sampling procedure reflecting "the actual probability distribution on the input sequences". This allows one to estimate the "operational reliability" [2]. Experiments performed by Duran and Ntafos [1] to study the effectiveness of random testing have shown that the sets of randomly generated test data for their sample programs have provided near total branch coverage. The branches uncovered have tended to be the ones that handle exceptional cases. This suggests that random testing may be used in conjunction with extremal/special values testing.

REFERENCES

[1] J. W. Duran and S. Ntafos.
A Report on Random Testing.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 179-83.

[2] J. W. Duran and J. J. Wiorkowski.
Quantifying Software Validity by Sampling.
IEEE Transactions on Reliability, Vol.R-29:141-4, June 1980.

2.2.9. GRAMMAR-BASED TESTING

SUMMARY

A formal specification of some systems, e.g., airline reservation systems and the call processing component of telephone switching systems can be modeled by a finite-state automaton (FSA). A regular grammar for the language accepted by the FSA can be constructed. A testing strategy can be based on the grammar to generate inputs and outputs of a system under consideration. The strategy is extended to test a wider class of programs by using attribute context-free grammars. The description of an automated test system implementing this strategy is discussed.

GRAMMAR-BASED TESTING STRATEGIES

The components of a grammar-based testing system are the Requirements Language Processor (RLP), the Test Plan Generator (TPG), and the Automatic Test Executer (ATE). Input to the RLP is a formal specification of the system under test. The output of the RLP is a state-transition matrix (STM) representation of a FSA. Since the RLP may assure that there are no inconsistencies in the requirements, it is assumed that the STM represents a deterministic FSA. The reachability of each state is assured by computing the transitive closure of the STM [2,3]. Using a result from automata theory a regular grammar for the language accepted by a FSA can be constructed [5]. The regular grammar is manually augmented to take into account the relevant system information for each state transition and to indicate the observable outputs of the FSA, e.g., "the observable outputs from the finite-state machine must be terminal symbols" in the grammar [1].

Input to the TPG is an augmented FSA. The TPG then outputs a set of test scripts. Each script is a sequence of executable inputs and corresponding outputs expected from the system under consideration. The ATE executes each test script and reports whether the system responds in a desired manner or not [1].

Another grammar-based strategy is described in [4]. This strategy is based on attribute context-free grammars and it addresses a more general class of programs. Theoretically, using the results obtained in [6], this strategy can be applied to any program.

EVALUATION

An important consideration associated with these strategies is that it is necessary to employ a criterion for choosing the productions in the grammar to prevent loops. One such criterion is to limit the number of times a production is used.

Grammar-based testing strategies have been applied for testing nested conditional statements in Ada, testing a sort program, and a testing text reformatter [4].

Another application of grammar-based test data generation strategies involves generating test programs for input to a compiler under test [7].

REFERENCES

[1] J. A. Bauer and A. B. Finger.
Test Plan Generation Using Formal Grammars.
Proceedings of the 4th International Conference on Software Engineering, September 1979.

[2] A. M. Davis and W. J. Rataj.
Requirements Language Processing for the Effective Testing of Real-Time Systems.
ACM Software Engineering Notes, Vol.3(5), November 1978.

[3] A. M. Davis and T. G. Rauscher.
Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications.
Proceedings of the Specifications of Reliable Software Conference, April 3-5, 1979, Cambridge, MA.

[4] A. G. Duncan and J. S. Hutchison.
Using Attribute Grammars to Test Designs and Implementations.
Proceedings of the 5th International Conference on Software Engineering, March 1981.

[5] J. E. Hopcroft and J. D. Ullman.
Introduction to Automata Theory, Languages, and Computation.
Addison-Wesley Publishing Company, 1979.

[6] D. R. Milton and D. N. Fischer.
LL(k) Parsing for Attribute Grammars.
Proceedings of the 6th International Colloquium on Automata, Languages and Programming, July 1979.

[7] A. J. Payne.
A Formalized Technique for Expressing Compiler Exercisers.
SIGPLAN Notices, Vol. 13(1), January 1978.

2.2.10. DATA-FLOW GUIDED TESTING

SUMMARY

Data-flow analysis is a technique used in optimizing compilers to analyze certain structural properties of programs. In data-flow guided testing, data-flow analysis is used to extract program variable relationships from a flow graph. Three data-flow guided techniques are block testing, which treats single-entry single-exit blocks as the basic unit of data transformation, the definition-tree strategy, and data-space testing. The effectiveness of these techniques has only been established for data transformation errors.

INTRODUCTION

Data-flow analysis is a technique for obtaining structural information about programs. In this approach, a program is considered as establishing meaningful relationships among program variables. Then a testing strategy may be defined in terms of data transformation paths for some or all program variables. Control flow information about the program is then used to define a set of paths to be exercised [1].

A variable in an instruction is said to be used (defined) when it is referenced (assigned a new value). A definition of a variable is live at an instruction I, if there exists a control path from where the variable is defined to I along which the variable is not redefined. The arguments of an instruction are those variables whose values are used by the instruction to perform the computation specified. An elementary data context of an instruction is a "tuple of definitions of all arguments of that instruction that can possibly be used" by the instruction if a particular control path is exercised. Data context of an instruction is "the set of all its elementary data contexts". A block in a program is a sequence of instructions executed together. For testing purposes, it is more convenient to deal with blocks rather than instructions. Therefore, the notion of liveness is extended to block level. The counterparts of the arguments and results of an instruction are the input and output variables of a block, respectively. A variable is an input (output) variable of a block, if it is used before it is defined (if it is defined) within the block [1].

A path in a program is said to be error-sensitive (error-revealing) if an error might be (is always) detected when it is exercised. A testing strategy is viable if it guarantees at least one error-sensitive path will be exercised.

TESTING STRATEGIES

One strategy is called block testing, and it is described by Laski [1] as follows:

> Each tuple of definitions from the data context of every block in the program is to be tested at least once.

In order to perform a complete block test, a set of paths that activate all elementary data contexts of every instruction needs to be exercised. A data flow analyzer can be used to find a complete test. A major drawback of this strategy is that it exercises blocks of a program independently, therefore failing "to force the control flow to activate more complex use-definition chains".

A definition-tree strategy has been proposed for meeting this weakness of block testing. In this approach, a programmer can specify a set of variables whose final values are of interest to him rather than the instructions which define them. These variables are generally output variables appearing in the specification of a program. Laski [1] describes the strategy as follows:

> The next step is to determine the data context of the exit instruction, i.e., the set of all tuples of simultaneously live definitions of the output variables. Each tuple becomes then the root of a definition tree (d-tree). The immediate sons of a root are the elementary contexts of the root ...

Next, the definitions of output variables are traced backwards from the exit instruction until "either an input, first block context is reached or a cyclic use of a context appears". As before, test data are generated to exercise each tuple in the tree at least once.

Another strategy similar to the definition-tree strategy is data-space testing. In contrast to the definition-tree strategy, this approach traces the definitions of all data items, e.g., variables and constants [2].

EVALUATION

Laski indicates that these strategies are more difficult to apply in practice than control oriented strategies, since finding data contexts of a block for a program of considerable size is almost impossible without software assistance (some algorithms which can be used for this purpose are referenced in [1]).

These strategies are conjectured to be viable for "data transformation errors" i.e., misspelled variables and incorrect values but they do not detect missing paths.

REFERENCES

[1] J. Laski.
On Data Flow Guided Program Testing.
SIGPLAN Notices, Vol.17(9), September 1982.

[2] M. Paige.
Data Space Testing.
Performance Evaluation Review, Vol.10(1):117-27, Spring 1981.

2.2.11. COMPILER TESTING

SUMMARY

Compiler testing has been studied extensively since the requirements of compilers are stable and languages such as FORTRAN and COBOL have been around for a long time. Grammar-based approaches and the approach used by the Federal Compiler Testing Center (FCTC) for testing compilers are introduced. An experimental evaluation of strategies is given.

COMPILER TESTING STRATEGIES

Grammar-based testing strategies (see Section 2.2.9) can be employed to test compilers. In these strategies, a compiler is exercised by a set of compilable programs, automatically generated by a test generator. The generator is driven by a description of the source language. This description is in a formalism which extends context-free grammars in a context-sensitive direction. The output of the test generator is a set of programs covering all syntactical units of the source language [1,9]. In addition to the automatic generation mode, the system in [1] provides the users with a set of directives to generate incorrect programs in a controlled way. This allows compiler implementers to hypothesize the type of errors and verify that the compiler accepts programs if and only if they are correct [1].

Another strategy to determine the degree to which the compiler under consideration conforms to a standard language definition (as defined by the National Bureau of Standards for example) is employed by the Federal Compiler Testing Center (Office of Software Development General Services Administration). A Compiler Validation System for a particular language consists of audit routines containing features of the language, their related data, and a preprocessor routine which prepares the audit routines for compilation. The testing of a compiler is performed as follows [2,3,4,5,6,7,8].

> The testing of a compiler in a particular hardware/operating system environment is accomplished by compiling and executing each audit routine. The report produced by each routine tells whether the compiler passed or failed the tests in the routine. If the compiler rejects some language elements by terminating compilation, giving fatal diagnostic messages, or terminating execution abnormally, then the test containing the code the compiler was unable to process is deleted and the audit routine compilation and execution repeated.

The compilation listing and the output reports of the audit routines are then analyzed to produce a validation summary report.

A different approach is being adopted by the implementers of the Ada (Registered Trademark of the Ada Joint Program Office, U.S. Government) programming language. Ada compilers must be "validated" against a test suite of 350 Ada programs and constructs [10]. At present, there are no Ada compilers which have been validated, so no experimental evidence is available on the effectiveness of this approach.

EXPERIMENTAL EVALUATION

The system described in [1] has been tested intensively on a Pascal subset, three PLZ compilers, and on some of the other languages. The system has been successful for discovering lexical, syntactical, and semantical errors. Some design errors of a PLZ compiler have been revealed.

The system of the Federal Compiler Testing Center has been used successfully to test different implementations of FORTRAN and COBOL.

REFERENCES

[1] F. Bazzichi and A. Spadafora.
An Automatic Generator for Compiler Testing.
IEEE Transactions on Software Engineering, Vol.SE-8(4):343-53, July 1982.

[2] Federal Compiler Testing Center.
Report FCTC-81-40, Falls Church, VA.

[3] Federal Compiler Testing Center.
Report FCTC-81-106, Falls Church, VA.

[4] Federal Compiler Testing Center.
Report FCTC-81-112, Falls Church, VA.

[5] Federal Compiler Testing Center.
Report FCTC-81-115, Falls Church, VA.

[6] Federal Compiler Testing Center.
Report FCTC-81-118, Falls Church, VA.

[7] Federal Compiler Testing Center.
Report FCTC-81-135, Falls Church, VA.

[8] Federal Compiler Testing Center.
Report FCTC-81-146, Falls Church, VA.

[9] A. J. Payne.
A Formalized Technique for Expressing Compiler Exercisers.
SIGPLAN Notices, Vol.13(1), 1978.

[1]] J. B. Goodenough.
Ada Compiler Validation Implementer's Guide.
SofTech, Inc., Waltham, MA, October 1980.

2.2.12. REAL-TIME SOFTWARE AND TESTING

SUMMARY

The characteristic phases of typical real-time software testing are host and target testing. Most of the testing techniques that are used for host computer testing are the same as for non-real time applications. The testing of an integrated system on a host requires running an environment simulator and controlling ongoing processes appropriately. In target testing, first module testing, then system integration and full system testing is performed.

Techniques for testing a ballistic missile defense system, an embedded-microprocessor command and control system, nuclear protection system, the NASA Space Shuttle Program, and material acquisition systems are presented.

INTRODUCTION

Real-time software is defined by Glass [5] as follows:

> Real-time software is software that drives a computer which interacts with functioning external devices or objects. It is called real-time because the software actions control activities that are occurring in an ongoing process. For example, real-time software may drive an acceleration/ deceleration controller in a rapid transit system vehicle; or it may capture data from other physical devices in a nuclear physics experiment; or it may interpret radar data onboard an antisubmarine aircraft and translate it into displays for a military operator at console.

An embedded computer system is defined as a computer and its software embedded in some larger system. "Real-time" and "embedded" are sometimes used interchangeably except that the embedded computer is physically included in the system that it serves [5].

The typical attributes of real-time software that makes the testing more difficult are the large number of modules that have to be integrated and tested, the same sequence of test cases, when input at slightly different times, may produce different outputs, the inherent logical complexity, e.g., large number of decision statements, many modules sharing the same computer at the same time, and many modules accessing the memory randomly which makes the isolation of problems difficult [6]. Glass [5] observes that real-time testing is still a "lost-world" compared to "civilization" developed in other areas of software, reflecting the little work done in the area.

The computers involved in real-time software testing are host and target computers. A target computer is the real-time computer which controls the activities of an ongoing process. A host computer is used to construct programs for the target and is usually a commercially available computer. It usually contains a cross compiler or a cross assembler or both, a linkloader for the target, and an instruction level simulator. An instruction level simulator is a program that allows a host to simulate the behavior of the target [5].

The characteristic phases of typical real-time software testing can be characterized as host and target computer testing. The goal of host computer testing is to reveal errors in the modules of software. Most of the testing techniques that are used for testing on a host computer are the same as for non-real-time applications. The testing of the full system is rarely done on the host. If it is done it requires running an environment simulator on the host and controlling ongoing processes appropriately. An environment simulator is an "automated replication of external system world" built for testing. Some target dependent errors and errors in the support software e.g., the compiler target code generator or assembler may be detected on the host by using an instruction level simulator [5].

In target testing, module testing is performed first. Software system integration testing may be performed by hooking the environment simulator to the target. Finally, full system testing is performed in the real world by removing the environment simulator. In all these phases of the target testing virtually no tools are available to support the real-time tester [5].

Glass proposes a methodology and presents a list of solutions for real-time software testing in [5].

Real-time software typically requires the generation of a large number of test cases. Since random generation of test data (see Section 2.2.8) is cost effective, techniques for random generation of test data may be employed.

In the following, techniques for testing a ballistic missile defense system, an embedded-microprocessor command and control software, nuclear protection systems, the NASA Space Shuttle Program, and material acquisition systems are discussed.

ADAPTIVE TESTING

Adaptive testing is an automated technique for testing using an adaptive tester in an interactive testbed consisting of the process under test and an environment simulator. In such an adaptive tester, test data selection is based on a performance criterion and automated techniques are used for test data generation, data gathering and reduction, and "intelligent" test case perturbation [2].

Performance criteria are used to select test data by determining the acceptable performance boundary of the system under test. For this purpose, the input space of the system is partitioned into acceptable and unacceptable domains of performance. This approach also allows stress sensitivity analysis of the system e.g., for obtaining stress parameters [2].

Automated aids for the generation of test cases, and data reduction and analysis involve the use of an algorithm called "Parameter Perturbation Algorithm" (PPA). PPA searches a performance surface to provide an intelligent perturbation of the input parameters. PPA performs the search in such a way that testing must proceed toward the stress boundary of the system. In order to decide which parameters to perturb, PPA uses the knowledge about previous test cases and a set of heuristics. Davis describes the heuristics that have been used to test a Ballistic Missile Defense (BMD) system as follows:

> The heuristics range from the known relationships that exist between an increase in the number of threatening objects and the expected number of objects to penetrate the defense to more detailed relations between software parameters at a lower level ...

Adaptive testing continues until a point on the stress boundary is reached [2].

AN EMBEDDED-MICROPROCESSOR COMMAND AND CONTROL SOFTWARE TEST TECHNIQUE

This technique takes into account the fact that embedded-microprocessor real-time software which monitors natural processes typically compute continuous functions. The inherent continuity of the functions computed by these monitor programs require them to have continuous inputs, outputs, or both. Watkins [7] describes the technique as follows:

> The software testing method presented here identifies potential errors by incrementally varying one (or more) input parameter's values over its (their) domain, usually by indexing on its (their) least significant value. Simultaneously, one (or more) output parameter's values are measured against an automatically calculated curve (surface). Potential errors are just those input-output combinations that result in an output which does not lie "close" to this curve. The curve is calculated from the immediately preceding output's value(s), the change in value(s), and the rate of change in value(s). Thus, this technique predicts the program's future behavior from its past behavior and flags as potential errors exactly those outputs which are not predictable.

For an implementation and an application for the technique to a piecewide linear function with 12-bit binary fraction input parameter, see [7].

TESTING TECHNIQUES FOR NUCLEAR PROTECTION SYSTEMS

After software of the reactor protection system has been coded, the source code is analyzed statically. Then, the source code is instrumented. Later, the code is executed on test data. Test data are selected either by analyzing the structure of the code and its specifications or from the estimated statistical distribution of data. All of the steps can be performed by a testing tool, e.g., SADAT and RXVP [4].

After this phase of software testing, the complete reactor system is tested under real-time conditions by performing specification-related tests followed by application-related tests. In the specification-related test, the unmodified software is tested and all possible input cases of the specifications are attempted to be covered. To specify the test data, the specifications are first analyzed to identify all elementary conditions e.g., the conditions in which the temperature is greater than the limit temperature. Then a subset of the possible combinations of these elementary conditions is selected by analyzing the results of the static analysis on "segmentation and the internal structure of algorithms". This subset is converted into decision table-like tables called combination tables whose columns correspond to distinct test cases [4].

During application-related testing, test data representing "frequently occurring situations and critical cases of the application" are generated by a test computer. Test computer then sends test data to the test object, and reads in the test results. It also documents the test results after checking them. In order to check the test results ...

> ... a functional model of the test object is derived from the specifications. The test data are also processed by the model. Then, the test results of the real system and the model are compared. To avoid identical errors in the real system and the model, the model is implemented by an independent team and in a high-level language.

A pilot experiment of protection system software and related work in the nuclear field are also included in [4].

A TESTING TECHNIQUE DEVELOPED FOR THE NASA SPACE SHUTTLE PROGRAM

The following technique has been used by IBM during the verification of the NASA Space Shuttle Project. The object of the Flight Software Verification Project was "an independent test, analysis and evaluation of the Space Shuttle flight program to ensure conformity to specifications and satisfaction of user needs" [1].

During the development cycle the modules were tested individually. At the completion of the development cycle, the software was released to a verification organization which was formed independently from the software development organization. Members of the verification organization participated in the design and code reviews conducted by the software development organization. During this phase, testing checklists were developed to design test cases [1].

The development and verification of the flight software were performed in an IBM-developed simulation testbed which provided "a sophisticated array of user capabilities" e.g., real-time flight software patching. The verification project involved two distinct and complementary phases: detailed and performance testings [1].

For detailed testing, the requirements specifications were divided into functional units. An individual analyst was responsible for developing a test plan to exercise each defined requirements path and a "sufficient" number of path combinations for each such unit. Additionally, " a level of requirements validation and user need testing" which was based on "the analyst's observations of the flight software response and knowledge of the intent of the requirements" was planned [1].

Detailed testing analysis which followed the acceptance of the test plan involves explicit testing of all requirements defined in the requirements specifications, explicit testing of I/O compatibility of modules, and each detailed test case was executed with the entire operating software. Each test case which revealed the existence of problems was executed again on the corrected software [1].

Performance testing involves testing flight software under "normal operation, 'reasonable' failures, and severely degraded conditions". Test cases were constructed in such a way that the components of software were stressed selectively. The criteria for performance success were "to operate successfully across the range of the customer's flight performance envelope" and "to satisfy a set of software design constraints" e.g., cpu utilization [1].

This approach to testing software for the NASA Space Shuttle Project resulted in 550 discrepancy reports and also revealed some specification discrepancies [1].

A METHODOLOGY FOR TESTING IN THE MATERIAL ACQUISITION PROCESS

The testing and evaluation methodology introduced in this section can be applied to software for complex material systems, e.g., tactical and weapon systems. The fundamental features of the methodology are "early involvement and activity participation by the tester-evaluator" during software development, "integration of test requirements", consideration of both functional and processing requirements throughout the development of software, and the provision for adequate data collection" by instrumenting software under consideration [3].

The methodology employs the methods of documentation analysis, software test-live environment, software test simulated environment, software/computer system simulation, and co-development participation. These methods are categorized as static and dynamic test methods. Each category is further subcategorized with respect to functional and process control, e.g., resource management and task management aspects of software under consideration.

Documentation analysis is a static analysis technique (see Section 2.2.1) which has applications to both functional and process control aspects of software. It does not require the availability of target computer hardware but requires the target software to perform code analysis.

Software test simulated environment and software test-live environment (also known as field testing) are both dynamic test methods having applications to functional and process control aspects of software. Simulated environment testing is performed throughout the development of software starting with module testing and proceeding through module integration testing and functional ("hardware-software") integration. Functional integration involves testing in both simulated and live environments, and possibly in a partially simulated environment. Live environment testing is necessary to demonstrate system capabilities for acceptance.

Data collection from simulated and live environments is performed by instrumenting the target software/computer system. Software monitors and hardware monitors are two basic methods used for instrumentation. A hardware monitor is a "physical device attached to the computer itself or to a data link to the computer (e.g., the computer-radar interface)." Data collection can also be done by a dedicated microprocessor rather than by instrumenmtation.

Software/Computer system is a static test method having application to test and evaluation of the process control aspects of software. Ellis [3] describes the simulation as follows:

> A simulation of the performance of the software/computer system defined by the devices and jobs specified is carried out under given workload or scenario conditions, and statistics relative to both devices are collected.

Package simulators or simulation languages, e.g., Extendable Computer System Simulator (ECSS) can be used for simulating software/computer systems.

This approach involves the participation of a tester during the software/system development process. The aim of this co-development participation is to reduce overall testing and development costs, and to have an improved final product.

REFERENCES

[1] J. F. Clemons.
Verification of the Onboard Flight Software Developed for the NASA Space Shuttle Program.
Proceedings of the Eighth Texas Conference on Computing Systems, 1979.

[2] C. G. Davis.
The Testing of Large, Real Time Software Systems.
Proceedings of the Seventh Texas Conference on Computing Systems, October 30 - November 1, 1978, Houston, TX.

[3] J. O. Ellis.
A Methodology for Software Testing in the Material Acquisition Process.
Proceedings of the Sixth Texas Conference on Computing Systems, November 14-15, 1977, University of Texas, Austin, TX.

[4] W. Geiger, L. Gmeiner, H. Trauboth, and U. Voges.
Program Testing Techniques for Nuclear Protection Systems.
Computer, Vol.12(8):10-8, August 1979.

[5] R. L. Glass.
Real-Time: The "Lost World" of Software Debugging and Testing.
Communications of the ACM, Vol.23(5):264-71, May 1980.

[6] R. V. Head.
Testing Real-time Systems. Part 1: Development and Management.
Datamation, page 42, July 1964.

[7] M. L. Watkins.
A Technique for Testing Command and Control Software.
Communications of the ACM, Vol.25(4):228-32, April 1982.

2.3. OTHER STRATEGIES FOR CONSTRUCTING RELIABLE SOFTWARE

SUMMARY

A number of strategies which can be used to augment formalized testing efforts have been proposed and successfully applied. These strategies usually take the form of policy and guidelines used to manage development efforts. These include the use of independent test organizations, team development approaches, and constructive methods for software development. Independent test organizations are separate from development organizations and may often be third party contractors. Team development approaches are based on organizational theories designed to establish effective communication patterns within software development organizations. Constructive methods include the application of advanced design methodologies and the use of support tools.

INTRODUCTION

In addition to formalized methodologies for testing, there are heuristics and guidelines used by software development management to improve the reliability of software under development. These include having an independent software organization test software, organizing teams to achieve better utilization of human resources and employing a hierarchical-decomposition design approach.

INDEPENDENT TEST ORGANIZATIONS

One approach to improving software reliability is the utilization of an independent test organization. Separate from the development organization, such a test organization performs analysis of software requirements, develops system test plans and scenarios, and evaluates software performance against the performance requirements of the system. The independence of the testing organization from the development organization is important to ensure an unbiased and independent evaluation. Independent test organizations are often contractors outside the project organization [3,4].

Indqpendent testing has been standard for highly critical realtime software contracts of the U.S. Air Force and other Federal agencies. For instance, independent testing of the Viking Lander Flight Software has been done by TRW Systems [3].

TEAM DEVELOPMENT APPROACHES

These approaches are intended to establish "simpler communication patterns within the software development organization" and to permit "the concentration of collective mental resources on design/programming problems" [2].

IBM's chief programmer team was the earliest of team concepts. A team consists of chief and back-up programmers. The chief is responsible for "designing, coding, and integrating the top-level control structure as well as the key components of the team's product". A back-up programmer assists the chief [1,2].

Variations of the chief programmer team approach are dual-member design team and thread integration team (for a discussion see Section 2.1). These differ in the way the teams are organized. The overall features of team approaches include continuous verification and validation, simpler communication paths, and better integration of efforts of individual team members [2].

CONSTRUCTIVE METHODS

Constructive methods of design and programming involve the application of structured techniques. These techniques include a structured (hierarchical-decomposition) design approach, "guided by formalized sets of principles" (e.g., modularity, abstraction, and uniformity), "processes" (e.g., purpose and mechanism), and "goals" (e.g., modularity, efficiency, and reliability) [1].

These methods "improve the testability of the system and furnish verification and validation assistance in parallel to the basic design and construction activities" [2].

Some automated tools are developed to aid in the structured design process. They assist the designer "in recognizing erroneous and weak areas in the design that are in need of revision". Examples of such tools include a structure chart graphics system and a design quality metrics system developed at Hughes Aircraft Company [2].

REFERENCES

[1] B. W. Boehm.
Software Engineering.
IEEE Transactions on Computers, Vol.C-25(12), December 1976.

[2] M. S. Deutsch.
Software Verification and Validation Realistic Project Approaches.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[3] R. L. Glass.
Modern Programming Practices, A Report from Industry.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

[4] G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

2.4. COMPARATIVE EVALUATION OF TESTING TECHNIQUES

SUMMARY

Problems associated with the selection of an appropriate testing methodology from the existing ones are explained. Comparative evaluation results of some of the testing techniques on sample programs are given.

COMPARATIVE STUDIES

The selection of an appropriate testing methodology from the existing ones is not an easy task. This was stated in [6] as follows:

> In choosing testing methods suitable for his own application, the software tester must weigh the particular advantages of each method against the resources it consumes and its shortcomings. The balance must meet the overall standard of testing required and still fall within the capabilities of the resources available - or else these constraints must be amended.

One problem that a tester must keep in mind is that testing is not conclusive; that is, it is never known how many errors in the software remain undetected.

Another problem associated with the selection of appropriate testing techniques is that there are no means of quantitatively assessing the effectiveness of a testing technique (and a tool). But, two approaches to measuring the effectiveness of testing techniques have been studied by Howden [4]: theoretical and empirical. He describes the two approaches as follows:

> In the theoretical approach, situations are characterized in which it is possible to use testing to formally prove the correctness of programs or the correctness of properties of programs. In the empirical approach, statistics are collected which record the frequency with which different testing strategies reveal the errors in a collection of programs.

The results obtained by theoretical measurements only provide insight into the reliability of software testing but have limited practical application. The difficulty associated with the empirical approach is that it is difficult to consolidate the information gained from empirical measurements since reliable error data is rare, even in small organizations [4,6,9].

Since there are no formal test effectiveness measures, a tester usually relies on the results obtained from practical experiences of different testing techniques. Several researchers have produced quantitative reports on the effectiveness of various testing techniques based on their experiments [1,2,3,4,5,8]. The results obtained in [4] indicate that path testing was reliable for 18 of the 28 errors. Branch analysis was reliable for 6 of the 28 errors. Structured testing was reliable for 12 of the 28 errors. Symbolic testing was found to be 10-20 percent more reliable than structured testing. The combined use of the structured and extremal values was reliable for 25 of the 28 errors. The combined use of all the techniques mentioned above was reliable for 26 of the 28 errors. Howden [3] found that path testing was "almost reliable" for about 65 percent of the program errors in the small survey of 11 programs in Kernighan and Plauger [7]. Howden [5] also found that symbolic testing and static analysis can be used to raise the error detection rate from 11 out of 22 for conventional testing to 17 out of 22 for the programs in [7]. The results obtained by Gannon [2] showed that path testing was more effective than static analysis at detecting logic, computational, and data base errors. Path testing alone detected 25 percent of the seeded errors. DeMillo [1], for a single program studied, found that domain testing was more effective than statement analysis, specification analysis, and branch analysis. Most of the results are mentioned in the experimental evaluation sections of each testing technique explained in this report. There is no published data available on comparative evaluation of test methodologies for large-scale software development efforts.

REFERENCES

[1] R. A. DeMillo, D. E. Hocking, and M. J. Merritt.
A Comparison of Some Reliable Test Data Generation Procedures.
Report GIT/ICS-81-08, Georgia Institute of Technology, 1981.

[2] C. Gannon.
Error Detection Using Path Testing and Static Analysis.
Computer, Vol.12(8):26-32, August 1979.

[3] W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions on Software Engineering, Vol.SE-2(3):208-14, September 1976.

[4] W. E. Howden.
Theoretical and Empirical Studies of Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-4(4):293-7, July 1978.

[5] W. E. Howden.
Symbolic Testing and the DISSECT Symbolic Evaluation System.
IEEE Transactions on Software Engineering, Vol.SE-3(4):276-8, July 1977.

[6] Infotech State of the Art Report, Software Testing Volume 1: Analysis and Bibliography.
Infotech International, 1979.

[7] B. W. Kernighan and P. J. Plauger.
The Elements of Programming Style.
McGraw Hill Book Company, New York, 1974.

[8] K.-C. Tai.
Program Testing Complexity and Test Criteria.
IEEE Transactions on Software Engineering, Vol.SE-6(6):531-8, November 1980.

[9] R. Thibodeu.
GRC Report to AIRMICS, 1981.

CHAPTER 3

TESTING AND EVALUATION TOOLS

3.1 INTRODUCTION

With the possible exception of certain static analysis techniques, testing techniques require controlled execution of programs. During program development, such controlled execution may be as uncomplicated as standard debugging runs against test data in which the programmer initiates, observes, and logs all testing activities. Even during early stages of development, however, automated aids are frequently required. For example, to test a free-standing subroutine, a driver must be used which simulates the actions of the calling sequence which will invoke the subroutine in its eventual run time environment. As logical units are assembled into larger modules, the possible interactions become more numerous and complex. Programs which will control these interactions and allow programmers and testers to simulate the intended actions of modules are common.

3.1.1. GENERAL VIEWS ON TESTING TOOLS

The application of systematic testing techniques also frequently requires automated help. Many techniques call for massive clerical operations such as developing verification conditions from an annotated program or determining variable liveness by data flow analysis. These operations are tedious and error-prone when carried out by hand and are best left to special purpose programs. Similarly, several testing techniques have considerable overhead in the number and size of test cases. Execution from a development environment of dozens -- perhaps hundreds -- of test cases under direct programmer control is so labor-intensive that in most cases the natural approach is to develop special software that retrieves test cases, initiates program execution, and logs test results. The development of the test data itself may involve processes which are combinatorially explosive when expressed as functions of the number of program components so that hand calculation is out of the question. For many applications, checking calculated results against expected results is not feasible manually. Output files may be too large for hand inspection, expected output may be derived by independent execution of an executable specification, correctness of output may be determined by performance constraints, or the number of executions required may be excessively large. In such cases, special programs are required to examine the results of execution and determine automatically the correctness of execution on test cases. The maintenance of test files, logs, and documentation may also be automated for similar reasons. Finally, execution of an integrated system in an environment which is subject to real time inputs and constraints or which controls physical events and processes may require software which simulates physical systems.

A testing tool is a piece of software which implements one or more of these functions. Rather than write special purpose programs for each software module under test, a software developer may find it more cost effective to obtain a generalized program which can perform its function in a variety of test situations. The borderline between a true tool and special purpose (throw-away) testing software is necessarily vague, but the following appear to be characteristics of testing tools:

- Generalized Interfaces
  A true tool should allow the user flexibility in specifying both the testing requirements and the program under test.

- Sharability
  A tool should be general enough in its formulation that it can be shared among a community of users.

- Reusability
  A tool should be able to successfully survive the first use. That is, the lifecycle of a tool should span the lifecycles of several applications.

The development of effective testing tools seems to be a prerequisite for successful application of systematic testing methodologies. The state-of-the-art in tools for software testing is surveyed in this chapter. The functions which these tools carry out have been classified to correspond to testing techniques whenever possible.

Information concerning existing tools has been extracted from the sources listed in Appendix A and by direct contacts with the developers. Readers should be aware that many of the published sources of tool-related information are woefully out of date. Tool development is frequently not carried out in an organized fashion and is frequently the responsibility of an individual. In addition, many tools are designed in response to a specific Government contract. Thus, access to an announced tool may be restricted because of poor documentation, unavailability of the individual responsible for the tool, and abandonment of the tool after contractual requirements are met.

Another pitfall in surveying testing tools is the proliferation of attributes describing the tools. Such questions as to what information is essential for the selection of a tool, how portable is a tool, what background is needed to successfully apply a tool, and how is the performance of the tool judged are frequently answered inadequately. For example, claims that a tool uncovered 20 errors in 20 programs or that the number of errors uncovered was significant are common. These evaluations amount to little more than testimonials and are not helpful in tool selection. Cost considerations are almost never announced, and cost along with performance appears to be the most important comparative attribute in determining tool selection.

In an attempt to back up published studies with more current data concerning tools, tool classification data sheets were sent to tool developers. A small fraction of these were returned with useful data. These data sheets are included in Appendix B and are summarized in tables following each major subsection below.

3.1.2. CLASSIFICATION

Testing tools can be classified into two groups according to the analysis they perform: static analysis tools and dynamic analysis tools. In addition, there is a family of related tools that neither perform direct tests nor use any specific testing technique. Such tools are called test support tools. This classification allows the grouping of tools using similar operations and components. A representative member of each grouping can then be used as a basis for explaining the underlying operational principles of all tools in the group.

Within each grouping, the tools may be further classified according to specific functions performed. The National Bureau of Standards has started to catalog tool features and classify software development tools in NBS Special Publications 500-75 and 500-88. Since many of the NBS terms are not yet well accepted, other sources have also been consulted in order to define the scope of a tool classification.

Comprehensive testing systems are combinations of analysis techniques and do not fall directly under any one category. These tools are listed in several classifications in the sequel. Some of them are summarized in the testing tool data sheets and at the end of the sections which are most applicable to their main functions. The information concerning other tools which are not summarized can be found in NBS Publication 500-88 and the other sources referred to above (see Appendix A).

Thus, each major subsection below contains an extended catalog of tools pertaining to that subsection. This catalog is organized as follows:

(1) Catalog Listing of Tools: an alphabetical listing of tools cited in the references of Appendix A which pertain to the current subsection.

(2) Listing According to Function: for each major function implemented by the tools cataloged in (1), an alphabetical listing of tools having that function.

(3) Data Sheet Summary Tables: an alphabetized set of summary descriptions for a subset of the cataloged tools. Information in these tables is taken from the testing tool data sheets in Appendix B. The data sheets were obtained by direct contact with the developers and sources obtained from Appendix A (of (1) above). The absence of a data sheet for a particular tool does not in itself imply that the tool is unavailable. However, in following up many of the cataloged tools, it was determined that the published availiability of a tool is no guarantee that it is actually available. A number of the data sheets were not returned because (a) the tool is proprietary, (b) the contact person has left the developing organization, (c) the organization does not market the tool, (d) the tool is not available for distribution and (e) the tool no longer exists.

3.2. STATIC ANALYSIS TOOLS

Static analysis techniques provide limited analysis of programs. The focus of static analysis is on requirements and design documents and on structural aspects of programs, i.e., on those characteristics of a program that are discernable without actually executing it. The tools that implement static analysis techniques are varied in scope and functionality. They range from systems which simply enforce coding standards to systems which carry out sophisticated structural error analyses.

3.2.1. STATIC ANALYSIS TOOL CLASSIFICATION

Static Analysis testing tools analyze characteristics obtainable from program structure without regard to the executability of the program under test.

Static analysis of programs may include any combination of the functions listed below. The tools that perform these functions are usually classified by function.

Code Auditing: Code auditing refers to the examination of source code to determine whether or not specified programming practices and rules have been followed. Typical rules and practices include adherence to structured design and coding, use of portable language subsets, or use of a standard coding format. Tools that implement such functions are called code auditors or standards enforcers.

Consistency Checking: A consistency check determines whether or not units of program text are internally consistent in the sense that they implement a uniform notation or terminology and are consistent with a specification. Such tools are usually used to check adherence to design specifications and are called consistency checkers.

Cross Referencing: Cross references are dictionaries relating entities by logical name. Cross referencing tools are frequently features of high level language compilers, although they also appear in debugging tools.

Interface Analysis: Interface analysis checks the interfaces between program elements for consistency and adherence to predefined rules or axioms. Typical interface checks may include checks on parameters passed to subroutines and the completeness of common blocks. These tools are called interface checkers.

I/O Specification Analysis: The goal of I/O specification analysis is the generation of input data by analysis of I/O specifications.

Data Flow Analysis: Data flow analysis originated in compiler optimization studies. It consists of the graphical analysis of collections of (sequential) data definition and reference patterns to determine constraints which can be placed on data values at various points of execution of the source program. Tools that perform such functions are called data flow analyzers.

Error checking: Error checkers determine discrepancies, their importance and causes.

Type Analysis: Type analysis involves the determination of correct use of named data items and operations. Usually, type analysis is used to determine whether or not the domain of values (functions, etc.) attributed to an entity are done so in a correct and consistent manner.

Units Analysis: Units analysis determines whether or not the units or physical dimensions attributed to an entity are correctly defined and consistently used.

3.2.2. STATIC ANALYZERS

SUMMARY

Static analyzers are programs that analyze source code to reveal global aspects of program logic, structural errors, syntactic errors, coding styles, and interface consistency. They consist of a front end language processor, a data base, an error analyser, and a report generator. The basic operation includes data collection, which creates necessary tables and graphs, error analysis, and error report generation. The existing static analyzers differ in terms of their scope of error analysis, the flexibility of user command languages, and the nature of error descriptions. Static analyzers have been used in many software development efforts.

GENERAL DESCRIPTION

Static analyzers are programs that analyze source code to reveal errors or dangerous constructs without actually executing the code [7].

E. F. Miller views a static analyser as, "a program analysis system that 'proves' static allegations about the programs it is asked to process [11]." An allegation here refers to a statement of a program property known to be desirable such as:

- all variables are type-declared in an explicit type declaration,
- each variable used in some statement is set before being referred to,
- no explicit type conversions are made in expressions evaluation, etc.

These allegations involve features of the programs that are actually legal in the programming language but are not good programming practice [12].

Static analyzers as described include many different types of tools using various techniques. In this section, we are concerned with those tools which use dataflow analysis as the main technique in extracting information from a source program.

Static analyzers are mainly used to check certain global aspects of program logic, syntactic errors, coding styles, and interface consistency. The information revealed by static analyzers include:

1) syntactic error messages;

2) number of occurences of source statements by type;

3) cross-reference maps of identifier usages;

4) analysis of how the identifiers are used in each statement (data source, data sink, calling parameter, dummy parameter, subscript, etc.);

5) subroutines and functions called by each routine;

6) uninitialized variables;

7) variables set but not used;

8) isolated code segments that cannot be executed under any set of input data;

9) departures from coding standards (both languages standards and local practice standards);

10) misuses of global variables, common variables, and parameter lists (incorrect number of parameters, mismatched types, uninitialized input parameters, output parameters not assigned, output parameters assigned but never used, parameters never used for either input or output, etc.) [7].

Control flow graphs and call graphs are created and analyzed to derive this information. In addition, variable usages in each statement must be investigated. For detailed information concerning dataflow analysis, see Section 2.2.10 and [1,2,3,7,8].

BASIC OPERATION

Static analyzers generally consist of four main components: a front end language processor, a data base, an error analyzer, and a report generator.

The front end language processor incorporates a lexical analyzer and a parser. In the DAVE system, there is a statement recognizer to classify different types of statements. A source program is subdivided into program units (e.g., main program, subroutines). Each program unit is further broken into statements, then tokens [15]. In this phase, a number of tables containing information such as variable usages, types, labels, and control flow are created and stored in a data base.

The data base used in most tools is specifically designed to store a large amount of information recorded during the lexical analysis phase. The query language is, therefore, not very flexible and is usually in the form of a command language. The FAST system uses a commercially available data management system as its data handler and data correlator along with the FACES source program parser and the BOBW parser generator [5].

The error analyzer performs error checking under the direction of a user who uses a provided command language or a query language to communicate with the system. The level of flexibility of the command language varies from system to system. The FACES system has the Automated Interrogation Routine (AIR) to interpret queries and automatically search the data base. The user may query the entire program, or an individual routine, by variable names or by lists of attributes. In the FAST system, the command/query language allows a user to request displays of statements or variables which satisfy specified attributes or a logical expression of attributes. The range of the analysis can be limited to within specified program lines or intra-module or the entire program.

The effectiveness of a static analyzer relies on the clarity of the error report that the system provides to a user. Most of the older static analyzers operate in batch mode and the report generation is done at the end of the analysis. The output may contain cross reference tables, calling sequence tables, common block versus subroutine cross reference table or program graphs. The DAVE system prints a description of each anomaly located; the description is designed to simplify the difficult task of identifying the cause. However, it does not attempt to positively identify the exact nature of every error in a program. Instead, the program is probed for suspicious and elusive constructs. The programmer must then use the messages and warnings produced by DAVE to improve the program [6].

The FACES system provides a trace routine to trace local variables within a module. The JOYCE system provides flowlists in the form of microfilm Fortran listings. Newer static analyzers such as FAST, the improved version of FACES, operate in the interactive mode, allowing a user to modify the direction of his testing and to see the results on a terminal right away.

The differences among static analyzers are the scope of error analysis, the power of the command languages provided, the nature of the error descriptions and the output tables produced. Some of the existing static analyzers are described individually at the end of this section.

Most of the static analyzers are designed mainly for Fortran and its dialects, as Fortran is a widely used language and there are many pitfalls associated with the language and its compiler. Normally, a Fortran compiler is structured to process one module at a time and does not check the compatibility of parameter interfaces and common blocks [18]. Another problem is the lack of type checking. Newer languages require explicit type declaration and their compilers enforce type checking for consistent usage of variables. Ada allows separate compilation and also requires type checking of the parameters of subprograms [4].

Although a compiler and a static analyzer have many common functions, there is a basic difference. The primary goal of a compiler is to produce object code efficiently. It is designed for maximum compilation speed and is constructed to forget source code details as quickly as possible. In contrast, the main function of a static analyzer is to locate structural errors and suspicious code practices. As the result, it has to record as many source code features as possible to facilitate anomaly discovery [18].

TOOL EVALUATIONS

Although there is no serious effort to evaluate static analysis tools, some experience has been gained in using many of the systems. Two static analyzers, DAVE and JOYCE, were evaluated by Leon G. Stucki. On the scale of 1 = low to 5 = high, the operating costs of DAVE and JOYCE were rated at 5 and 2, respectively. In term of ease of uses (1 = easy to 5 = difficult), both systems were rated at 2 [17].

The FACES system was used to analyze itself and three instances of misspelling errors and two subtle keypunch errors were found [16]. In an initial application at NASA, FACES found approximately 1 error per 200 statements in a large Fortran program. In an analysis of software associated with NASA's space shuttle, FACES found problems in 6.5% of the statements [13,14].

Brown and Johnson state that DAVE represents one of the best Fortran validation tools available. However, it does not provide a full range of analyses [6].

LIMITATIONS

All static analysis tools are limited by the problem of decidability. Static analyzers also face the problem of array element identification when the subscript is a variable. Since the exact value of the variable may not be known until the execution time, one cannot generally know which array element is being referenced or defined in a statement. The DAVE system treats all elements of the same array as a single variable [15].

REFERENCES

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

[2] F. E. Allen.
Interprocedural Data Flow Analysis.
Proceedings of the IFIP Congress 1974, pages 398-402. North Holland Publishers, Amsterdam, 1974.

[3] F. E. Allen and J. Cocke.
A Program Data Flow Analysis Procedure.
Communications of the ACM, Vol.19(3):137-47, March 1976.

[4] J. G. P. Barnes.
Programming in Ada.
Addisson-Wesley Publishing Company, 1982.

[5] J. C. Brown and D. B. Johnson.
FAST: A Second Generation Program Analysis System.
Proceedings of the 3rd International Conference of Software Engineering, May 10-12, 1978, Atlanta, GA, pages 142-8.

[6] J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY, February 1980.

[7] R. E. Fairley.
Tutorial - Static Analysis and Dynamic Testing of Computer Software.
Computer, Vol.11(4):14-23, April 1978.

[8] R. E. Fairley.
Proceedings, Infotech State of the Art Conference on Program Testing, September 1978, London.

[9] R. C. Houghton, Jr.
Software Development Tools.
NBS Special Publication 500-88, National Bureau of Standards, 1982.

[10] W. E. Howden.
A Survey of Static Analysis Methods.
In Tutorial: Software Testing & Validation Techniques, E. Miller and W. E. Howden, Editors, pages 101-15. IEEE, 1981.

[11] Infotech State of the Art Report, Software Reliability, Volume 1: Analysis and Bibliography.
Infotech International, 1977.

[12] Infotech State of the Art Report, Software Reliability, Volume 2: Invited Papers.
Infotech International, 1977.

[13] G. J. Myers.
The Art of Software Testing, pages 149-51.
John Wiley & Sons, Inc., New York, 1979.

[14] NASA Software Specification and Evaluation System, Final Report.
Science Applications, Huntsville, AL, 1977 (NTIS N77-26828).

[15] L. J. Osterweil and L. D. Fosdick.
DAVE: A Validation Error Detection and Documentation System for Fortran Programs.
Software Practice and Experience, Vol.6(4):473-86, October-December 1976.

[16] C. V. Ramamoorthy and S-B. F. Ho.
Testing Large Software with Automated Software Evaluation Systems.
IEEE Transactions on Software Engineering, Vol.SE-1(1):46-58, March 1975.

[17] L. G. Stucki.
Software Development Tools - Acquisition Considerations - A Position Paper.
National Computer Conference, AFIPS Proceedings, Vol.46:267-8, 1977.

[18] I. K. Wendel and R. L. Kleir.
Fortran Error Detection through Static Analysis.
Software Engineering Notes, Vol.2(3):22-8, April 1977.

CATALOG LISTING OF STATIC ANALYSER TOOLS

The following tools have been listed as static analysis tools by one or more of the sources in Appendices A or B.

	STATIC TOOLS	SOURCE OF INFORMATION
1.	ADF	[1,10]
2.	ADS/CERL	[1,10]
3.	AFFIRM	[1,10,11]
4.	ARTS	[1]
5.	ASSET	[1,2,10]
6.	ATDG	SEE APPENDIX B
7.	AUDIT	[1,10,11]
8.	AUDITOR	[1,10,11]
9.	AUTO-DBO	[1]
10.	AUTOFLOW	[1,10]
11.	CADSAT	[1]
12.	CALLREF	[1,10]
13.	CARA	[1,10]
14.	CAVS	[1,10,11]
15.	CCA	[1]
16.	CCREF	[1,10]
17.	CICS COMP ANALY	[1]
18.	COBOL/DV	SEE APPENDIX B
19.	COBOL/QDM	[1]
20.	COBOL STRUCT	[1]
21.	COBOL/SP	[1]
22.	COMGEN	[1]
23.	COMGEN/TRW	[1,10]
24.	COMLIST	[1]
25.	COMLIST/TRW	[1,10]
26.	COMMAP	SEE APPENDIX B
27.	COMSCAN	[1]
28.	COMSORT	[1,10]
29.	COMSORT	[1,19]
30.	CONFIG	[1,10,11]
31.	CONFIGURATOR	[1]
32.	CORE	[1]
33.	CPA-ADR	[1,10]
34.	CRO-REF	[1,10]
35.	DA	[1,10]
36.	DATAMACS	SEE APPENDIX B
37.	DAS	[1,10]
38.	DARTS	[1]
39.	DAVE	SEE APPENDIX B
40.	DCD	[1,10]
41.	DDPM	[1]
42.	DPECHT	[1,10]

STATIC TOOLS	SOURCE OF INFORMATION
43. DICTANL/LOCATE	[1]
44. DOCUTOOL	SEE APPENDIX B
45. DPNDCY	[1,10]
46. ECA Automation	[1]
47. ENFORCE	[1]
48. FACES	SEE APPENDIX B
49. FADEBUG-I	SEE APPENDIX B
50. FASP	[1,11]
51. FAST	SEE APPENDIX B
52. FAVS	SEE APPENDIX B
53. FCA	SEE APPENDIX B
54. FLOBOL	[1]
55. FORAN	SEE APPENDIX B
56. FORREF	[1,10]
57. FORTRAN AUDITOR	SEE APPENDIX B
58. FORTREF	[1,10]
59. FTNXREF	[1,10]
60. GAVS	[1]
61. GENTESTS	[1]
62. GIRAFF	[1,10]
63. GOTO-ANALYZER	[1]
64. HAWKEYE	[1,10]
65. INFORM/REFORM	[1,11]
66. INTERFACE DOCUM	SEE APPENDIX B
67. ISUS	[1,10,11]
68. JAVS	SEE APPENDIX B
69. JIGSAW	[1,10,11]
70. JOVIAL/VS	[1]
71. JOYCE	SEE APPENDIX B
72. LEXICON	[1,10]
73. LIBREF	[1]
74. LOGICFLOW	[1,11]
75. LOGOS	[1,10]
76. MED-SYS	[1,10]
77. MEDL-R	[1]
78. PBASIC	[1]
79. PDL	[1,10]
80. PDL/PSA	
81. PET	SEE APPENDIX B
82. PFORT	SEE APPENDIX B
83. PREF HDR GEN	[1,11]
84. PSL	[1]
85. QUICK-DRAW	[1,10]
86. RA	[1]
87. RADC/FCA	[1,10]
88. REFER	[1,10]
89. REFTRAN	[1,10]

STATIC TOOLS		SOURCE OF INFORMATION
90.	RISOS TOOLS	[1,10]
91.	RTT	[1]
92.	RXVP80	SEE APPENDIX B
93.	SADAT	SEE APPENDIX B
94.	SARA	[1,10]
95.	SCAN/370	[1,11]
96.	SCG/DQM	[1]
97.	SDL	[1]
98.	SDP/MAYDA	[1]
99.	SEF	[1,2,10,11]
100.	SIGS	[1]
101.	SNOOP	[1,10,11]
102.	SOFTOOL80	[1,10,11]
103.	SPECLE/DARS	[1]
104.	SPELL	[1,10]
105.	SREM	[1,10]
106.	SREP	[1,10]
107.	SRIMP	[1]
108.	SSA	[1]
109.	STAG/TWMS	[1,10]
110.	STRUCT	[1,10]
111.	STRUCTURE(S)	[1,10]
112.	SUBCRS	[1,10]
113.	SURVAYOR	[1,10,11]
114.	SYDIM	[1,10]
115.	SYDOC	[1]
116.	SYMCRS	[1,10]
117.	SYSXREF	[1]
118.	TAPS/AM	[1,10]
119.	TOOLPAK	[1,10]
120.	TPT	[1]
121.	UCA	[1,10]
122.	VIRTUAL	[1]

STATIC ANALYSIS TOOLS LISTED ACCORDING TO FUNCTIONS

The following list classifies the static analysis tools cataloged above by function.

DATA FLOW ANALYSIS

ADF
ATDG
AUDIT
CAVS
DARTS
DAVE
DCD
DDPM
FACES
FAST
FAVS
ISUS
PREF HDR GEN
RXVP80 (TM)
SADAT
SARA
SNOOP
SOFTOOL 80 (TM)
SRIMP
SURVAYOR
TOOLPACK
TPT

INTERFACE ANALYSIS

AUTO-DBO
DAVE
FAST
FORAN
INFORM/REFORM
JAVS
PREF HDR GEN
RXVP80 (TM)
SEF
SOFTOOL 80 (TM)
SYDIM

CROSS REFERENCE

ADS/CERL
AUTOFLOW (TM)
CALLREF
GAVS
CCREF
CICS DUMP ANALY
COBOL/SP
COMGEN/TRW
COMGEN
COMLIST/TRW
COMLIST
COMMAP
COMSORT
CONFIG
CORE
CPA-ADR
CRO-REF
DA
DAVE
DCD
DDPM
DEPCHT
DICTANL/LOCATE
DPNDCY
FASP
FAVS
FLOBOL
FORAN
FORREF
FORTREF
FTNXREF
GIRAFF
INTERFACE DOCUM
JOYCE
LEXICON
LIBREF
LOGOS
PBASIC
PDL
PSL/PSA
QUICK-DRAW
REFER
REFTRAN (TM)
RISOS TOOLS
RTT
RXVP80 (TM)
SARA
SCAN/370
SCG/DQM
SDL
SDP/MAYDA
SNOOP
STAG/TEMS
STRUCTURE(S)
SUBCRS
SYDOC
SYMCRS
SYSXREF
TAPS/AM
TOOLPACK
VIRTUAL OS

COMPLETENESS CHECKING

AUTO-DBO
CADSAT
CONFIGURATOR
MEDL-R
PSL/PSA
PWB FOR VAX/VMS
RA
RXVP80 (TM)
SARA
SIGS
SOFTOOL 80 (TM)
SPECLE/DARS
SREM

CONSISTENCY CHECKING

AFFIRM
ARTS
ASSET
AUTO-DBO
CARA
CONFIGURATOR
DAS
FAST
FORAN
MED-SYS
MEDL-R
MEDL-D
PSL/PSA
RA
RXVP80 (TM)
SARA
SCG/DQM
SREM
SREP
SRIMP

UNIT ANALYSIS

RSVP80 (TM)
UCA

TYPE ANALYSIS

AFFIRM
FAVS
RXVP80 (TM)

AUDITING

ADS/CERL
AUDITOR
AUDIT
CA
CCA
COBOL/QDM
COBOL STRUCT
COMSCAN
CPA-ADR
DAS
ECA AUTOMATION
ENFORCE
FACES
FCA
GOTO-ANALYZER
HAWKEYE (TM)
JIGSAW
JOVIAL/VS
LOGICFLOW
PBASIC
PET
PFORT
PSL
RACD/FCA
SADAT
SCG/DQM
SOFTOOL 80 (TM)
SPELL
SSA
STRUCT

ERROR CHECKING

ATDG
AUDITOR
COMMAP

I/O SPECIFICATION ANALYSIS

COBOL/DV
GENTESTS
DATAMACS
PREF HDR GEN
FADEBUG-I

STATIC ANALYZERS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 1 of 2)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
COMMON BLOCK MAP/ COMMAP	Static Analyzer	Cross Reference Error Checking	A/L/S	CDC	FORTRAN 77	FORTRAN 66 FORTRAN 77	YES	N/A	Boeing Computer Services
/DAVE	Static Analyzer	Data Flow Analysis Interface Analysis Cross Reference	A/P/N	CDC 6400 IBM UNIVAC DEC/VAX	FORTRAN 66	FORTRAN	YES	$250	University of Colorado
/DOCUTOOL	Static Analyzer	Automatic Code Documentor	A/L/S	CDC	Pascal	FORTRAN 66 FORTRAN 77	YES	N/A	Boeing Computer Services
FORTRAN Analyzer Program/FORAN	Static Analyzer	Consistency Checker Interface Analysis Cross Reference	A/P/S	CDC 6000 CDC 7000	FORTRAN	FORTRAN	YES	N/A	U.S. Army Adv. Res. Ctr.
FORTRAN AUDITOR/	Static Analyzer	Auditing Error Checking	A/L/S	DEC Data Gen. IBM Gould-SEL	FORTRAN	FORTRAN	YES	$16,000	Softool Corp.
FORTRAN Automated Code Evaluation System/FACES	Static Analyzer	Data Flow Analysis Auditing	A/-/N	UNIVAC CDC 6400 IBM 360	FORTRAN	FORTRAN	YES	N/A	University of Georgia (COSMIC)
FORTRAN Code Auditor/FCA	Static Analyzer	Auditing	A/P/N	Honeywell	FORTRAN IV	FORTRAN Y	YES	N/A	TRW

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

STATIC ANALYZERS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 2 of 2)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
FORTRAN Analysis System/FAST	Static Analyzer	Consistency Checking Data Flow Analysis Interface Analysis	-/L/-	System 2000 (MRI System Corp)	FORTRAN	FORTRAN	N/A	N/A	Information Research Associates (IRA)
INTERFACE DOCUMENTER/	Static Analyzer	Interface Analysis	A/L/S	DEC, DG, IBM Gould-SEL	FORTRAN	FORTRAN, COBOL, any object code	YES	$7,000	Softool Corp.
/JOYCE	Static Analyzer	Cross Reference	A/-/S	CDC 6X00/ 7X00	FORTRAN	FORTRAN	N/A	N/A	McDonnell
PFORT Verifier/ PFORT	Standard Enforcer	Auditing	A/P/-	N/A	FORTRAN	FORTRAN	N/A	N/A	Jet Propulsion Laboratory
/RXFP-80	Static Analyzer Test Driver Assertion Processor Instrumenter Coverage Analyzer	Data Flow Analysis Interface Analysis Cross Reference Completeness Checking Consistency Checking Type Analysis	A/L/S	CDC, IBM UNIVAC	N/A	FORTRAN, IFTRAN (TM), or V-IFTRAN (TM)	YES	$26,000	General Research Corporation

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

3.3 DYNAMIC ANALYSIS TOOLS

Dynamic analysis tools provide support for testing by direct execution of the program being tested. The range of functions supported by dynamic tools is broad. Systems which generate and evaluate test data using any one (or a combination) of the testing techniques listed in Chapter 2 have been implemented and used in a variety of settings. In addition, tools which provide run-time statistics through program instrumentation are in fairly widespread use. The tools that are discussed below fall roughly into one of four categories: symbolic evaluators, test data generators, program instrumenters, and program mutation analyzers.

3.3.1 DYNAMIC TOOL CLASSIFICATION

Dynamic analysis tools are the tools that collect information from the execution of the tested program. According to the report NBS 500-88, this group includes symbolic execution tools which could be considered either static or dynamic tools.

Dynamic analysis includes the following operations:

Coverage Analysis - determining and assessing measures associated with the invocation of program structural elements to determine the adequacy of a test run. A tool for this function is called a coverage analyzer.

Tracing - tracing the historical record of execution of a program. Tracing can be further divided to path flow tracing, breakpoint control, logic flow tracing, and data flow tracing. A tool for this function is called a tracer.

Tuning - determining what parts of a program are being executed the most.

Simulation - representing certain features of the behavior of a physical or abstract system by means of operations performed by a computer.

Timing - reporting actual CPU times associated with a program or its parts.

Resource Utilization - analysis of resource utilization associated with system hardware or software.

Symbolic Execution - reconstructing logic and computations along a program path by executing the path with symbolic, rather than actual values of data. A tool for this function is called a symbolic evaluator.

Assertion Checking - checking of user-embedded statements that assert relationships between elements of a program. An assertion is a logical expression that specifies a condition or relation among the program variables. Checking may be performed with symbolic or run-time data. A tool that performs this function is called a dynamic assertion processor or an assertion checker.

Constraint Evaluation - generating and/or solving path input or output constraints for determining test input or for proving programs correct. This function is generally a part of the symbolic evaluator and test data generators.

3.3.2. SYMBOLIC EVALUATORS

SUMMARY

Symbolic Evaluators are programs that accept symbolic values and execute them according to the expression in which they appear in a program. They are used to support test data generation, assertion checking, path analysis, and detection of data flow anomalies. The basic operation consists of symtax analysis, path selection, evaluation of path constraints, constraint simplification, and inequality solving. Problems concerning loop iteration and array reference are the main limitations of symbolic evaluators. Some of the well-known systems include SELECT, EFFIGY, ATTEST, DISSECT, and SMOTL.

GENERAL DESCRIPTION

Symbolic evaluators or symbolic executors are programs that accept symbolic values for some of the inputs and algebraically manipulate these symbols according to the expressions in which they appear. These tools perform operations symbolically as if the program were executing and derive output values as symbolic expressions involving the input variables [4].

The primary use of symbolic evaluators has been to support test data generation. Secondary applications include symbolic debugging, assertion checking, path analysis, detection of unreachable code, array boundary errors, and potential overflow or underflow.

As described in Chapter 2, the basic idea in symbolic evaluation is to allow numeric variables to take on symbolic as well as numeric values. A program is interpreted on symbolic values without compilation.

BASIC OPERATION

The basic operation of symbolic evaluators consists of syntax analysis, path selection, evaluation of paths symbolically, constraint simplification, and inequality solving. The typical system flow of symbolic evaluation is shown in Figure 1. A syntax analyzer converts the source program into an internal representation. This representation is combined with values saved in a file and executed symbolically on designated paths. In path selection, a user may have to make the decision as to which path is to be analyzed. For instance, if a program is executed on actual data, predicates in branch statements such as the if-statement can be evaluated to either true or false, and the system can select the appropriate control-path. On the other hand, if a predicate contains symbolic values, it may not be evaluated. In this case, a user decides which path he wants to select or the system selects all possible paths. To give his decision to the system, the user may designate paths to be followed interactively, or by supplying a list of paths in a batch approach.

After the evaluation on a path, variables and constraints (a set of predicates or inequalities) might be simplified automatically by a simplifier. Clarke used Atlan, a language designed for algebraic manipulations by Bell Labs, to transform and simplify nonlinear constraints into linear constraints [2].

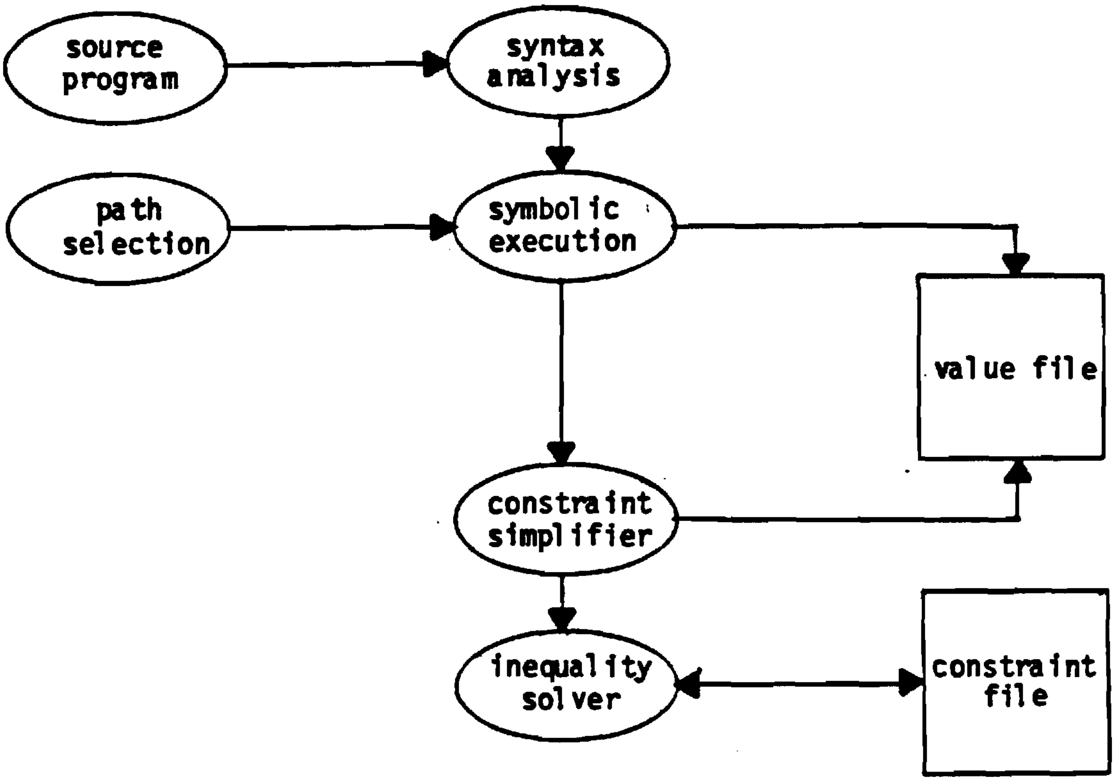


Figure 1. System Flow of a Symbolic Evaluator

Each constraint generated by the system and simplified by the simplifier is passed to an inequality solver to check its consistency with the existing constraints saved in a constraint file. If the constraint is inconsistent, the path is infeasible. The system will inform the user of this fact. If the constraint is consistent, the symbolic execution of the path continues. Techniques to obtain a solution employed by the inequality solver are linear-programming, trial-and-error, and a fast segment algorithm used in SMOTL [1]. Techniques employed by existing tools are summarized in Table I and in Appendix B.

There are some differences among symbolic evaluation systems. ATTEST, SMOTL, and CASEGEN function mainly as test data generators. EFFIGY is mainly used as an interactive debugging tool. It provides features such as tracing, breakpoints, state saving, and assertion checking. DISSECT analyzes ANSI Fortran program to determine the computations along selected paths, the set of symbolic values which cause the path to be executed, and the symbolic values of the output variables. However, DISSECT does not provide automated test data generation. The SELECT system provides static analysis, path structure analysis, assertion checking, and test data generation. Except SMOTL, which is oriented towards data-processing application, all systems mentioned above are research tools.

Table 1: Summarized Features of Current Symbolic Evaluators and Test Data Generators

TOOL NAME (First Publication)	SOURCE LANGUAGE	MODE OF OPERATION	FORMULA OUTPUT	TEST DATA GENERATION	ASSERTION VALIDATION	PATH SELECTION	ARRAY REFERENCE	LOOP ITERATION.	INEQUALITY SOLVER
AMPIC	FORTRAN ASSEMBLY		YES			AUTOMATIC (all segments)			
ATTEST (1975)	ANSI FORTRAN	INTERACTIVE or BATCH	YES	MAIN FUNCTION	NO	USER DEFINED	AMBIGUITY is UNDEFINED	USER DEFINED	LINEAR PROG.
CASEGEN (1976)	FORTRAN	BATCH	NO	YES	NO	AUTOMATIC (All branches)		FIXED	TRIAL & ERROR
DISSECT (1976)	FORTRAN	BATCH	YES	NO	YES	USER DEFINED or selected automatically if max. number of loop executions specified	COLON EXPRESSION	USER DEFINED	NON-LINEAR
EFFIGY (1975)	PL/I subset (integer variables & one dimensional array)	INTERACTIVE	YES	NO	YES	USER DEFINED interactively		USER DEFINED	NO SOLUTION
SELECT (1975)	QLISP (LISP SUBSET)	INTERACTIVE or BATCH	YES	YES	YES	USER DEFINED or selected automatically if max. path lengths specified	ALL CASES	FIXED	LINEAR PROG.
SMOTL (1977)	SMOD (COBOL subset)	BATCH	NO	YES	NO	AUTOMATIC			

TOOL EVALUATIONS

The main limitations of symbolic evaluation systems are problems concerning loop iteration and array reference as mentioned in Chapter 2. There have been many evaluations on symbolic systems. Some of the evaluations of well-known systems are summarized below.

EFFIGY is limited in practical use. It only applies to programs written in a simple PL/I style language that is restricted to integer variables and one dimensional arrays and the array reference problem is left unresolved [3,6].

SELECT is an effective tool for rapidly revealing program errors but needs additional manipulative powers beyond inequalities and algebraic simplification. It does not have some useful features such as the detection of potential overflow and underflow, division by zero, and reference of unitialized variables. The system is better suited to analysis of moderate sized data-processing programs than to complex algorithms [3].

The ATTEST system has difficulties in handling Fortran arrays and does not have file implementation. Its inequality solver is limited to systems of linear predicates [2,3].

Howden evaluated the effectiveness of symbolic execution using his tool DISSECT [5]. In his experience, six programs which contain a total of 28 errors were selected. One of two heuristic strategies utilizing DISSECT was functional testing with symbolic values, which examines each functional module decomposed from a program as a separate program. The other approach used was symbolic integrated testing in which functional modules are examined within the context of the entire program. Of 28 errors, the former approach, functional testing, detected 14 errors and the latter approach, integrated testing, detected 15 errors. With the six sample programs, he found the use of symbolic testing resulted in an increase in reliability 10-20 percent over the conventional testing. In addition, he also indicated that symbolic evaluation was very useful in four of the six sample programs for eliminating infeasible paths. Stucki [7] rated DISSECT's operation cost at 4 in a scale of 1 = low to 5 = high, and its ease of use was rated at 3 in a scale of 1 = easy to 5 = difficult.

REFERENCES

[1] J. Bicevskis, J. Borozovs, U. Straujums, A. Zarins, and E. F. Miller, Jr.
SMOTL - A System to Construct Samples for Data Processing Programming Debugging.
IEEE Transactions on Software Engineering, Vol.SE-5(1):60-6, January 1979.

[2] L. A. Clarke.
A System to Generate Test Data and Symbolically Execute Programs.
IEEE Transactions on Software Engineering, Vol.SE-2(3):215-22, September 1976.

[3] J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY, Feburary 1980.

[4] H. R. Downs.
Automated Tools for the Verification of Computer Programs.
Transactions of the American Nuclear Society and the European Nuclear Society, 1980 International Conference, November 16-21, 1980, Washington, DC, pages 253-4.

[5] W. E. Howden.
An Evaluation of the Effectiveness of Symbolic Testing.
Software Practice and Experience, Vol.8(4):381-97, July - August 1978.

[6] J. C. King.
Symbolic Execution and Program Testing.
Communication of the ACM, Vol.19(7):385-94, July 1976.

[7] L. G. Stucki.
Software Development Tools - Acquisition Considerations - A Position Paper.
National Computer Conference, AFIPS Proceedings, Vol.46:267-8, 1977.

REFERENCES NOT CITED IN THE TEXT

T. E. Cheatham, Jr., G. H. Holloway, and J. A. Townley.
Symbolic Evaluation and the Analysis of Programs.
IEEE Transactions on Software Engineering, Vol.SE-5(4):402-17, July 1979.

L. A. Clarke.
Testing - Achievements and Frustrations.
Proceeding of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 310-4.

Infotech State of the Art Report, Software Reliability, Volume 2: Invited Papers, pages 184-215.
Infotech International, 1977.

Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography, pages 125-37.
Infotech International, 1979.

W. E. Howden.
Symbolic Testing and the DISSECT Symbolic Evaluation System.
IEEE Transactions on Software Engineering, Vol.SE-3(4):266-78, July 1977.

I. Myamoto.
Automated Testing-Aid Tools Survey.
Information Processing Society of Japan (Joho Shori) (Japan), Vol.20-(8):688-93, August 1979.

CATALOG LISTING OF SYMBOLIC EVALUATORS

The following tools have been listed as symbolic evaluators by one or more of the sources in Appendices A or B.

SYMBOLIC EVALUATORS	SOURCE OF INFORMATION
1. ATDG	SEE APPENDIX B
2. ATTEST	SEE APPENDIX B
3. CASEGEN	SEE APPENDIX B
4. COBOL/DV	SEE APPENDIX B
5. DISSECT	SEE APPENDIX B
6. EFFIGY	SEE APPENDIX B
7. GENTEXTS	SEE APPENDIX B
8. SELECT	SEE APPENDIX B
9. SMOTL	SEE APPENDIX B
10. TEVERE-1	SEE APPENDIX B

SYMBOLIC EVALUATORS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 1 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
Automated Test Data /ATDG	Test Data Generation Symbolic Evaluator	Test Data Generation Path Structure Analysis Anomaly Detection Variable Analysis	-/-/-	UNIVAC	FORTRAN	N/A	N/A	N/A	TRW for NASA in Houston
Automatic Test Enhancement System /ATTEST	Test Data Generation Symbolic Evaluator	Test Data Generation Data Flow Analysis Automatic Path Selection Constraint Simplification	-/P/N	VAX	FORTRAN 77	FORTRAN 66	YES	N/A	Software Development Laboratory University of MA.
/CASEGEN	Test Data Generation Symbolic Evaluator	Path Generation Automatic Test Data Generation Path Constraint Generation	-/-/	N/A	FORTRAN	FORTRAN	N/A	N/A	N/A

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

SYMBOLIC EVALUATORS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 2 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
/COBOL/DV	Test Data Generation Symbolic Evaluator	Documentation Aid Test Data Generation Run-Time Debugging Aid I/O Specification Analysis	A/L/S	N/A	COBOL	COBOL	N/A	N/A	Applied Data Research
/DISSECT	Symbolic Evaluator	Path Structure Analysis Documentation Assertion Checking Static Analysis	A/P/-	PDP-10 LISP System	LISP	ANSI FORTRAN	YES	N/A	N/A
/EFFIGY	Symbolic Evaluator	Assertion Checking Interaction Debug Tools	-/-/-	IBM/370 Model 168	PL/1	PL/1 restricted to integer valued variables and one dimensional arrays	N/A	N/A	IBM

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

SYMBOLIC EVALUATORS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 3 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
/GENTEXTS	Test Data Generator for Compilers	Test Data Generator for Compilers	A/L/S	CIT-HB	PL/1 Pascal SIMULA 67	N/A	YES	$2,000	IRISA, University of Rennes, France
Symbolic Evaluation Language to Enable Comprehensive Testing/SELECT	Symbolic Evaluator Test Data Generator Assertion Processor	Static Analysis Path Structure Analysis Assertion Checking Test Data Generation	-/-/-	PDP-11	LISP	LISP Subset	N/A	N/A	SRI International Advanced Computer Systems Dept.
/SMOTL	Test Data Generator Symbolic Evaluator	Test Data Generator Regression Testing Run-Time Error Detecting Coverage Analysis	-/-/-	MINSK-32	SMOD	N/A	N/A	N/A	N/A
A Software System for Programs Testing and Evaluation /TEVERE-1	Test Data Generator Symbolic Evaluator	Symbolic Evaluation	A/P/N	PDP-11	LISP 1.4	IFTRAN	YES	N/A	S. Bologna, J. R. Taylor, ENEA CRE-CASACCIA

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

3.3.3. TEST DATA GENERATORS

SUMMARY

A test data generator is a tool which assists a user in the generation of test data. Three types of test data generators are pathwise test data generators, data specification systems and random test data generators. Pathwise test data generators have four basic operations: program digraph construction, path selection, symbolic execution, and test data generation. Difficulties with the use of the test data generator are the computational efforts wasted in computing infeasible paths and in array reference problems. A data specification system provides a user with a special language to specify his data files. The system then generates test data from a specification program written in the provided language. Random test data generators simply pick random values from input domains. Some of the existing test data generators are the ATTEST, SETAR, SMOTL, CASEGEN, and ADG systems.

GENERAL DESCRIPTION

A test data generator is a tool which assists a user in the generation of test data for a program or module. The purpose is to relieve the effort required in generating a large volume of test data, and in the case of automatic test data generation, to avoid programmer's bias in preparing his own test data.

A test data generator only assists a user in generating test cases, since a test case consists of both test data and expected output [16]. The expected output is usually determined by hand calculation, simulation, or with the aid of a test specification system such as REVS to be a test oracle responding to the output [1].

CLASSIFICATION AND BASIC OPERATION

Test data generators can be classified into three types:

1) pathwise test data generators,
2) data specification systems, and
3) random test data generators.

1. Pathwise Test Data Generators

One approach to generate input data that is a comprehensive representation of the input space is to select input data from the input domains associated with program paths. The inputs are selected to exercise a specified set of program paths. Systems that generate test data in this manner are known as pathwise test data generators. This is the most common type of test data generator. Its basic operation consists of four main steps: program digraph construction, path selection, symbolic execution, and test data generation. The primary differences among these types of systems are in the techniques of test path selection and in early detection of infeasible test paths. Other differences include the breadth of their symbolic execution capability and capacity for symbolic simplification of algebraic expressions [14].

Program Digraph Construction

The source program is preprocessed to create a digraph representation of control flow in the program. Other relevant information is collected for later analysis.

Path Selection

Path selection is concerned with selecting program paths that satisfy testing criteria. The test criteria may be a level of test coverage as mentioned in [9,19]. Frequently implemented examples are total path coverage, statement coverage, and branch coverage. In these criteria, either every feasible path, or every statement, or every branch statement must be executed at least once. The ATTEST system [7] provides a choice among these three coverage criteria as well as a loop boundary condition in which the system creates path descriptions that will execute a program's loops a minimum and maximum number of times [8]. The SMOTL system [13] utilizes a coverage criterion which requires that every program segment be executed at least once; the path selection is completely automatic. In the SETAR system, a new path is selected by altering one or more of the contraints in the path conditions gained from executing previous data [14].

The path selection process can be manual or automatic, static or dynamic. Manual dynamic path selection requires a user to select the next statement whenever a decision point is encountered. This is very tedious, inefficient, and difficult. Manual static path selection requires a user to completely specify program paths before analysis is initiated. However, users tend to inadvertently select a large portion of non-executable paths. In automatic static selection, paths are automatically selected prior to symbolic execution. This method is usually based on the graph structure of the program; without additional semantic information, it has the same drawbacks as the previous two methods. Automatic total selection requires all the feasible paths, and therefore, has the disadvantage of inundating the user with paths [13].

The number of program paths is very large and the path length is usually unbounded. Most systems must select the paths by one of the following techniques:

(1) a user specifies all the paths to be analyzed in advance;

(2) a user specifies in advance the maximum path length to be traced to the maximum number of loop executions;

(3) a user interactively selects the path to be analyzed and executes it statement by statement;

(4) automatic selection by the system to satisfy a level of test coverage [4,19].

Table I in Section 3.3.2 summarizes selection methods used by some well-known systems.

Symbolic Execution

Once a path is selected, symbolic execution is used to generate path constraints. Path constraints consist of equalities and inequalities describing program input variables; input data satisfying these constraints will result in the execution of that path.

Test Data Generation

This step involves selecting data that will cause the execution of the selected paths. Most systems use linear programming algorithms to find numerical solutions to the inequalities of path constraints. SELECT initially used two linear programming algorithms, GOMORY [10] and BENDERS [2], and later switched to a gradient algorithm which solves a wider class of inequality systems but must be run interactively [5]. Clarke [7] uses F. Glover's linear programming algorithm [4].

There are two other approaches. CASEGEN uses a trial and error method in conjunction with a random number generator [17]. SMOTL uses a fast segment algorithm [3].

The weaknesses associated with pathwise test data generators are the significant computational effort wasted in analyzing infeasible paths, loop and array reference problems in symbolic execution [4,17].

2. Data Specification Systems

A data specification system assists a user in the generation of a test case by providing a data specification language to describe the input data. The system then uses the description to generate the desired input data [16].

An example of such a system is the automatic data generating program (ADG). It is a compiler which translates the ADG code, an English-like language, describing the characteristics of a data file into a PL/1 program which will generate the specified data file [15].

GENTEXTS is another system with a similar basic operation. It is designed to prepare test programs for compiler testing. Its data specification language is in the form of command grammars describing the desired test programs. The system processes the grammar to generate SIMULA programs which are then compiled and executed to generate the actual test programs [11].

File generators can be considered as data specification systems in the sense that they generate test files by using special command languages to describe the data structure of the files. E. F. Miller, however, points out that while test data generators are concerned with the values of data to cause the execution of program segments, file generators are more concerned with the form of the data structure and in some cases also generate typical values for the content of chosen fields within the generated files [12].

DATAMACS is a flexible file generator for COBOL programs. It generates all types of files, creates hierarchical record structures, and changes field values automatically. The data specification language is in the form of special control cards interspersed in the environment and data divisions. Data is created using both the control commands and information from the file definition [11,18].

3. Random Test Generators

Test data is generated by simply selecting a random point from the domain of each input variable of a program. For the randomness to be meaningful, it must be applied to both the selection of data within a path domain and the selection of different path domains. If a uniform distribution is chosen, the method is equivalent to a black-box approach as mentioned in Section 2.1. Moranda comments that usage of random test data is more stressing to a program than those constructed by analysts as test cases. The main advantage of random testing is its simplicity; it is also the easiest way to introduce some program independence into the testing process [13]. However, the value of the test is yet to be established as discussed in Section 2.2.8.

TOOL EVALUATIONS

Most of the existing test data generators are still in the development stage. Detailed information concerning their performance and effectiveness is not available in the literature. However, the remainder of this section discussed details of some existing tools. The year in parentheses indicates how current this information is.

ATTEST (1979)

The ATTEST system is a pathwise test data generator. It has difficulties with arrays and file manipulation. I/O specification is partially implemented. The test data generation component is restricted to systems of linear predicates. ATTEST's testing criteria include recognizing two types of structural subcases: loop boundary conditions and language dependent conditions such as index range check, division by zero; and three methods of path selections: statement coverage, branch coverage, and total path coverage. The current system is under development [8].

SETAR (1979)

The SETAR system is a manual and dynamic pathwise test data generator. The method of selecting program paths is somewhat different from those previously described. New data is generated by negating one or more constraints in the path conditions gained from executing previous data. The new path is then used to generate the new input. The main benefit from this approach is that it helps to generate test cases that are relevant to the problem, since a user can use his knowledge of the problem to constrain the input domain by manipulating path conditions during the process of generating a new path. The user also has the control of the dependency of the test cases on the detailed structure of the program [14]. The main drawbacks of the system are that the system requires user's assistance for effective test data generation and there is no system-provided measure of coverage. The user must have knowledge of the functional specification of the program to be tested. The system is in the research phase, and no assessment of the system's performance is available.

SMOTL (1979)

The system is for batch processing of programs written in SMOD, a COBOL-like language without means of direct access to secondary storage. The current implementation shows that for data-processing-style programs, it is possible to construct a complete test set (for branch coverage) in acceptable time on widely used computers. Claims of 85% automatic generation of test data are made. A system for programs written in PL/1 is being developed [13].

CASEGEN (1976)

The system is a pathwise test data generator for programs written in Fortran. It consists of about 10,000 Fortran statements. Data base generation and path selection are processed at a rate of about 10 statements/CPU second on the CDC 6400. The processing time required for symbolic execution and test data generation is about half a second per constraint up to ten constraints. Experience shows that a large portion of execution is spent in backtracking within the test data generation phase. In order to improve efficiency, a user-oriented language has been designed to allow a user to specify additional information about the range of input variables, the number of loop iterations and relations among program variables. The final system will be integrated into the FACES system [17].

REFERENCES

[1] W. R. Adrion. M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

[2] J. F. Benders.
Partitioning Procedures for Solving Mixed-Variables Programming Problems.
Numerical Math, Vol.4:238, 1962.

[3] J. Bicevskis, J. Borozovs, U. Straujums, A. Zarins, and E. F. Miller, Jr.
SMOTL - A System to Construct Samples for Data Processing Program Debugging.
IEEE Transactions on Software Engineering, Vol.SE-5(1):60-6, January 1979.

[4] Y. V. Borzov.
Program Testing Using Symbolic Execution.
Programming and Computer Software, Vol.16:39-45, 1980.

[5] R. S. Boyer, B. Elspas, and K. N. Levitt.
SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution.
SIGPLAN Notices, Vol.10(6):234-45, June 1975.

[6] B. Chandrasasekaran.
Test Tools: Usefulness Must Extend to Everyday Programming Environment.
Computer, Vol.12(3):102-3, March 1979.

[7] L. A. Clarke.
A System to Generate Test Data and Symbolically Execute Programs.
IEEE Transactions on Software Engineering, Vol.SE-2(3):215-22, September 1976.

[8] L. A. Clarke.
Automatic Test Data Selection Techniques.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 43-63.
Infotech International, 1979.

[9] J. A. Darringer.
The Use of Symbolic Execution in Program Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 67-85.
Infotech International, 1979.

[10] R. E. Gomory.
An Algorithm for Integer Solutions to Linear Programs.
In Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe, Editors. McGraw-Hill, New York, 1963.

[11] R. C. Houghton, Jr.
Software Development Tools
NBS Special Publication 500-88, National Bureau of Standards, 1982.

[12] Infotech State of the Art Report, Software Reliability, Volume 2: Invited Papers, pages 184-215.
Infotech International, 1977.

[13] Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography, pages 167-79 and 229-43.
Infotech International, 1979.

[14] S. Kundu.
SETAR - New Approach to Test Case Generation.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 163-87. Infotech International, 1979.

[15] N. R. Lyons.
An Automatic Data Generation System for Data Base Simulation and Testing.
Data Base, Vol.8(4):10-3, 1977.

[16] G. J. Myers.
The Art of Software Testing
John Wiley & Sons, New York, 1979.

[17] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen.
On the Automated Generation of Program Test Data.
IEEE Transactions on Software Engineering, Vol.SE-2(3):293-300, December 1976.

[18] Software Engineering Automated Tools Index.
Software Research Associates, P. O. Box 2432, San Francisco, CA, 94126.

[19] Test Coverage.
In Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography, pages 68-78. Infotech International, 1979.

REFERENCES NOT CITED IN THE TEXT

L. A. Clarke, J. Hassell and D. J. Richardson.
A Close Look at Domain Testing.
IEEE Transactions on Software Engineering, Vol.SE-8(4):380-90, July 1982.

L. A. Clarke and D. J. Richardson.
A Partition Analysis Method to Increase Program Reliability.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 244-53.

D. W. Fife.
Test Data Generation: Three Approaches Prevail.
Computer, Vol.12:103-4, March 1979.

W. E. Howden.
Methodology for the Generation of Program Test Data.
IEEE Transactions on Computers, Vol.C-24(5):554-9, May 1975.

R. J. Peterson.
TESTER/1: An Abstract Model for the Automatic Synthesis of Program Test Case Specifications.
Proceedings of the Symposium of Computer Software Engineering, pages 629-35, IEEE, New York, 1976.

Proceedings of the Specifications of Reliable Software Conference.
IEEE Catalog No. CH1401-9c, IEEE, New York, 1979.

C. V. Ramaoorthy.
Techniques for Automated Test Data Generation.
Conference Record of the Ninth Asilomar Conference on Circuits, Systems and Computers, November 1975.

D. Teichroew and F. A. Hershey, III.
PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering, Vol.SE-3(1):41-8, 1977.

CATALOG LISTING OF TEST DATA GENERATORS

The following tools have been listed as test data generators by one or more of the sources in Appendices A or B.

TEST DATA GENERATORS	SOURCES OF INFORMATION
1. AMPIC	SEE APPENDIX B
2. ASSIST-1	[1,10,11]
3. ATDG	SEE APPENDIX B
4. ATTEST	SEE APPENDIX B
5. CASEGEN	SEE APPENDIX B
6. COBOL/DV	SEE APPENDIX B
7. DATAMACS	SEE APPENDIX B
8. DISSECT	SEE APPENDIX B
9. ECA AUTOMATION	[1]
10. EFFIGY	SEE APPENDIX B
11. GENTESTS	[1]
12. GENTEXTS	SEE APPENDIX B
13. NASA-VATS	[1,10,11]
14. RXVP80	SEE APPENDIX B
15. SADAT	SEE APPENDIX B
16. SELECT	SEE APPENDIX B
17. SETAR	[1]
18. SMOTL	SEE APPENDIX B
19. TEST PREDICTOR	[1,10,11]
20. TEVERE-1	SEE APPENDIX B
21. TPT	[1]

TEST DATA GENERATORS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 1 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
/AMPIC	Symbolic Evaluator	Symbolic Execution Path Predicate Calculation Global Cross-Ref. Structured and Un-Structured Flow Charts	-/-/-	IBM 360, 370	SNOBOL	WSC FORTRAN Assembly (WSC, Litton L4516D	N/A	N/A	LOGICON, Inc.
Automated Test Data /ATDG	Test Data Generation	Test Data Generation Path Structure Analysis Anomaly Detection Variable Analysis	-/-/-	UNIVAC	FORTRAN	N/A	N/A	N/A	TRW for NASA in Houston
Automatic Test Enhancement System /ATTEST	Test Data Generation	Test Data Generation Data Flow Analysis Automatic Path Selection Constraint Simplification Symbolic Evaluation	-/P/N	VAX	FORTRAN 77	FORTRAN 66	YES	N/A	Software Development Laboratory University of MA.
/CASEGEN	Test Data Generation	Path Generation Automatic Test Data Generation Path Constraint Generation	-/-/	N/A	FORTRAN	FORTRAN	N/A	N/A	N/A

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

TEST DATA GENERATORS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 2 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
/COBOL/DV	Test Data Generation	Documentation Aid Test Data Generation Run-Time Debugging Aid I/O Specification Analysis Tracing	A/L/S	N/A	COBOL	COBOL	N/A	N/A	Applied Data Research
/DATAMACS	Test Data Generator I/O Specification Analyzer Software Management Control and Maintenance	Test File Generation I/O Specification Analysis Regression Testing File Structure Testing	-/-/-	IBM 360, 370	BAL	COBOL	N/A	$16,000	Management and Computer Services, Inc.
/DISSECT	Symbolic Evaluator	Path Structure Analysis Symbolic Execution Assertion Checking Static Analysis	A/P/-	PDP-10 LISP System	LISP	ANSI FORTRAN	N/A	N/A	N/A
/EFFIGY	Symbolic Evaluator	Assertion Checking Interactive Symbolic Execution Proof of Correctness	-/-/-	IBM/370 Model 168	PL/1	PL/1 restricted to integer valued variables and one dimensional arrays	N/A	N/A	IBM

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

TEST DATA GENERATORS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 3 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
/GENTEXTS	Test Data Generator for Compilers	Test Data Generation to test some particular aspects of a compiler	A/L/S	CIT-HB	PL/1 Pascal SIMULA 67	N/A	YES	N/A	IRISA, University of Rennes, France
Symbolic Evaluation Language to Enable Comprehensive Testing/SELECT	Symbolic Evaluator Test Data Generator Assertion Processor	Static Analysis Path Structure Analysis Assertion Checking Test Data Generation Symbolic Execution	-/-/-	PDP-11	LISP	LISP Subset	N/A	N/A	SRI International
A Software System for Programs Testing and Evaluation /TEVERE-1	Test Data Generator	Symbolic Evaluation Path Structure Analysis	A/P/N	PDP-11	LISP 1.4	IFTRAN	YES	N/A	S. Bologna, J. R. Taylor, ENEA CRE-CASACCIA, S.P. ANGUILLARESE IN1 300,00060 Roma-Italy
/SMOTL	Test Data Generator	Test Data Generator Regression Testing Run-Time Error Detecting Coverage Analysis Batch Operation	-/-/-	MINSK-32	SMOD	N/A	N/A	N/A	Software Research Associates

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

3.3.4. PROGRAM INSTRUMENTERS

SUMMARY

Program instrumenters are systems that insert software probes into source code in order to reveal its internal behavior and performance. Their main applications are coverage analysis, assertion checking, and detection of data flow anomalies. There are three types of program instrumenters: dynamic execution verifiers, self-metric instrumenters, and dynamic assertion processors. The basic operation of instrumenters consists of a preprocessing phase, a compilation and execution phase, and a post-processing phase. Instrumenters are found to be effective tools in evaluating the coverage of test cases. Some extensive instrumenters include PET, FAVS, JAVS, and PACE.

GENERAL DESCRIPTION

Program instrumenters gather execution data to reveal characteristics of a program's internal behavior and performance by inserting monitoring statements into the source code.

Instrumentation is the principal dynamic analysis tool used to detect errors that cannot or may not economically be found by static analysis. The main applications of program instrumentation in software testing can be divided into three groups:

1. Coverage Analysis

The determination and assessment of measures associated with the invocation of program structural elements is used to determine the adequacy of a test run [1]. This information is useful to evaluate the success of test cases and to design a better set of test data to improve the test coverage.

2. Monitors and Assertions

To aid debugging, instrumentation is used to trace the change of variable values. Assertion statements are inserted at critical points in a program to check that certain conditions must be true for a valid operation. An assertion is a statement that specifies a condition or relation about certain program variables and is placed in a program in the form of a comment [6]. Assertion checking techniques allow a programmer to express validation requirements in a way that reflects the program's intended function [2].

3. Detection of Data Flow Anomalies

Data flow instrumention records the minimum and maximum values of each variable, so the violations of predefined range conditions can be detected. When combined with a state transition table, instrumentation can also detect errors concerning references to uninitialized variables and variables that are defined but are not used [7,8,9]. Although these errors can also be detected by static analyzers, it may be more economical to use program instrumentation because a program will be tested in its construction and, with instrumentation, the useful information is obtained as the by-product of a test [9].

CLASSIFICATION OF PROGRAM INSTRUMENTERS

Program instrumenters can be classified, according to their main functions, into three categories:

1. Dynamic Execution Verifiers

These systems are sometimes also called coverage analyzers or automated verification systems. Dynamic verifiers are programs that evaluate the effectiveness of individual tests in terms of some constant measure of the degree each test exercises portions of a program [11].

Downs defines automated verification systems as programs that instrument the source code by generating and inserting counters at strategic points to provide measures of test effectiveness. They provide data that details how thoroughly the source code has been exercised [3].

In general, test coverage is interpreted in terms of the number of times a program segment is exercised during a test. A program is grouped into segments in the form of decision-to-decision (D-D) graphs. A counter is placed at each D-D path to count the frequency of execution of that segment. After the modified program is compiled and executed, the post-processor analyzes the collected data to present the result in terms of the relative percentage of time spent executing particular segments, the list of unexercised segments, etc. There has been research on methods to minimize the execution overheads by using a minimal number of counters in deriving the coverage analysis. For more information, see Section 2.2.3 and [8,13].

2. Self-Metric Instrumenters

These systems have the capability to instrument programs to report information concerning their internal behavior. A user specifies a list of variables and the scope of the instrumentation with provided user commands. The system automatically inserts probe statements at appropriate locations.

Self-metric instrumenters provide more general information than dynamic execution verifiers. The information provided by full self-metric instrumentation varies with the statement type. Typical information includes the number of times executed, the initial, final, maximum, minimum, and average value assigned to a variable, the number of times a condition of a branch statement is true or false, and the number of words that an input or an output statement transfers.

The advantage of this type of tool is the comprehensiveness of the information provided. However, the execution-time overhead costs tend to be high; between 50% and 200% additional processing time and approximately 50% additional execution space [11].

3. Assertion Checkers

Assertion checkers are programs that convert assertions into modifications of the source program that issue warnings whenever the assertions are false [11].

The assertions are transparent to the normal language compilers and must be preprocessed in order for dynamic execution checking to occur. The assertion checker will replace an assertion with corresponding probe statements to instrument the program [14].

The main advantage of automated assertion processing is the simplification of the process of removing the assertions once they are no longer needed [11]. These assertions are entered as comments in program code and are meant to be permanent. They provide both documentation and means for regression testing [1].

Assertion checking is used for program validation, error detection, dynamically checking critical parameters for range, value, and order violations based on the prescribed bounds of the assertions [2]. This technique is also used to aid in proving program correctness. This application is covered in Section 1.3 and [5,10,12].

Some programming languages such as EUCLID, PLAIN, Ada, ALGOL-W, and PL/CS provide assertional capability. Most of them provide only elementary assertional capability. EUCLID, however, provides extensive assertion statements [9].

The capability provided by an assertion checker is usually more powerful and flexible than that provided in programming languages. A user specifies the scope of the assertion statement to certain statements or to any part of a data structure such as a portion of a column of an array. Once the testing is completed, these assertions have no functional effect on the execution of the program since they are completely ignored as comments.

BASIC OPERATION

The operation of all instrumenters can be divided into three phases:

1. Preprocessing Phase

The main operation of this phase is to insert appropriate instrument statements into a source program. In a self-metric system, the run-time data base is also updated and the probe statements are mapped into the compile file using templates. For an assertion checker, an assertion is checked to determine whether it is active. The active assertions are then replaced by corresponding statements. In the case of an execution verifier, the source program is first processed by a program syntax recognizer before the probes can be inserted.

2. Compilation and Execution Phase

The augmented program is then compiled and executed. During the execution of a self-metric system, a run-time package accesses the stored descriptions of internal information to produce reports that describe the computation performed. Similarly in an assertion checker system, when a condition of an assertion is violated, a report is generated and execution continues.

3. Post-processing Phase

Statistics generated during program execution are matched with individual source program statements to produce an annotated program listing and summary report. For an execution verifier, the post-processing activities analyze the contents of the trace file and produce coverage reports.

TOOL DESCRIPTIONS AND EVALUATIONS

Most available tools are integrated systems, employing both static and dynamic analysis. The descriptions here are concerned with the instrumentation features provided.

PET (Program Evaluator and Tester)

PET is a dynamic tool which combines the features of a self-metric instrumenter, a dynamic execution verifier, assertion checker, and also some static analysis. The PET system provides MONITOR commands of the form:

```
MONITOR [NUMERIC/CHARACTER] [RANGE] FIRST [n VALUES]
     LAST [n VALUES] [(List of variables) / ALL]

MONITOR SUBSCRIPT RANGE [(List of array names) / ALL].
```

These commands will cause the instrumention of the specified variables or all variables to report the first, last, maximum, and minimum values [14].

The PET system provides extensive assertional capabilities. There are two types of assertions: global and local. Global assertions and monitor commands are located with the declarations and have effect over the length of their enclosing module or block. Local assertions are position dependent and consist of any legal logical expression of the host language. Examples of PET's assertions are:

Global assertions:

ASSERT RANGE (List of variables) (min, max)
- This assertion examines each specified variable and reports the new values that fall outside the range.

ASSERT VALUES (List of variables) (List of legal values)
- This assertion inspects and reports the new values that are not of the specified legal values.

Local assertions:

ASSERT (extended logical expression) [HALT on n [VIOLATION]]
- The HALT option will stop the program if n violations occur.

ASSERT ORDER (array cross-section) [ascending/descending]
[HALT ON n [VIOLATIONS]]
- This assertion checks the array cross-section values to verify that they are in the selected sequence.

The ease of use of the system was evaluated at 1 in a scale of 1 = easy to 5 = difficult. The operating cost was evaluated at 2 or 3, depending on the options used, in a scale of 1 = low to 5 = high. PET was recommended for use in situations where operating cost is not a major factor in selection [2,15]. PET has been used to analyze the test coverage of an operational system of 40,269 program statements along with test data that had been used to test the system prior to release. It showed that the test data covered only 44.5% of the executable source statements and only 35.1% of branches. The increase in execution time of the instrumented program varies from 25% to 150% depending on the options used [2,4].

FAVS (Fortran Automated Verification System)

FAVS produces reports indicating which modules, D-D paths and program statements have been exercised, the number of times each statement was executed and each D-D path that was transversed. The reports are generated for the current test and cumulatively for all past test cases, for a single module or a group of modules. D-D paths not transversed for the current test case and for all test cases are also identified [2].

JAVS (Jovial Automated Verification System)

The JAVS system performs coverage analysis and produces comprehensive reports identifying the paths remaining to be exercised. Execution analysis indicates which modules, decision paths, and statements have been exercised, including the number of times each statement was executed and the execution time spent in each module. The program provides tracing capabilities to monitor the invocations and returns of all modules, values of variables, and important events, such as overlay link loading. The system also provides assertional capability to check logic expression and an EXPECT directive to check the boundaries of expected variables [2].

TRW researchers report that JAVS is an advance automated verification system with well-organized documentation but the output report is D-D path oriented and requires manual correlation to interpret the meaningful results [2,16]. In general, the execution of a JAVS-instrumented program requires 1.5 times the execution time of an uninstrumented program and approximately twice the load core size. The TRW group points out that the overhead caused by recording execution monitoring data on a mass storage trace file would be unacceptable for the instrumentation of an entire medium to large scale system [2].

PACE (Product Assurance Confidence Evaluator)

PACE is a collection of tracing and managerial tools which assist in assessing test coverage for Fortran programs. In testing and maintenance of the Houston Operations Predictor/Estimator (HOPE) program, the system helped to save $8,000 per year. It revealed that the existing test file consisting of 33 test cases covered only 85% of the program and that one-half of this number were exercised by almost every test case. The test results evaluation required 4.5 hours of computer time and 35-50 manhours. From the statistics provided by PACE, a more effective test file, consisting of six test cases, was generated. With the test cases, 43% of the subprograms were exercised and required less than 24 manhours of test examination.

REFERENCES

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

[2] J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY, February 1980.

[3] H. R. Downs.
Automated Tools for the Verification of Computer Programs.
Transactions of the American Nuclear Society and the European Nuclear Society, 1980 International Conference, November 16-21, 1980, Washington, DC, pages 253-4.

[4] T. Gilb.
Software Metrics.
Winthrop Publishers, Inc., Cambridge, MA, 1977, page 282.

[5] C. A. Hoare.
Proof of a Program: FIND.
Communications of the ACM, Vol.14(1), January 1971.

[6] R. C. Houghton, Jr.
Software Development Tools.
NBS Special Publication 500-88, National Bureau of Standards, 1982.

[7] J. C. Huang.
Instrumenting Programs for Data Flow Analysis.
Technical Report TR-UH-CS-77-4, University of Houston, May 1977.

[8] J. C. Huang.
Program Instrumentation: A Tool for Software Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 144-59. Infotech International, 1979.

[9] Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography.
Infotech International, 1979.

[10] R. L. London
Proof of Algorithms: A New Kind of Certification.
Communications of the ACM, Vol.13(6), June 1970.

[11] E. F. Miller, Jr.
Program Testing Tools and Their Use.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 184-215. Infotech International, 1979.

[12] Proceedings of an ACM Conference on Proving Assertions about Programs. SIGPLAN Notices, Vol.7(1), January 1972. Also reprinted in SIGACT News, Vol.14, January 1972.

[13] C. V. Ramamoorthy, K. H. Him, and W. T. Chen.
Optimal Placement of Software Monitors Aiding Systematic Testing.
IEEE Transactions on Software Engineering, Vol.SE-1(4):403-11, December 1975.

[14] L. G. Stucki and G. L. Foshee.
New Assertion Concepts for Self-Metric Software Validation.
Proceedings of IEEE Conference on Reliable Software, April 1975, Los Angeles, CA, pages 59-65.

[15] L. G. Stucki.
Software Development Tools - Acquisition Consideration - A Position Paper.
National Computer Conference, AFIPS Proceedings, Vol.46:267-8, 1977.

[16] TRW Systems and Space Group.
NSW Feasibility Study, Final Technical Report.
RADC-TR-78-23, February 1978.

REFERENCES NOT CITED IN TEXT

J. M. Adams.
Experiments on the Utility of Assertions for Debugging.
Proceedings Eleventh Hawaii International Conference on System Science, Honolulu, HI, January 1978, pages 31-9.

S. H. Saib.
Executable Assertions - An Aid to Reliable Software.
Proceedings Eleventh Annual Asilomar Conference on Circuits, Systems and Computers, November 1977, Pacific Grove, CA, pages 277-81.

Software Engineering Automated Tools Index.
Software Research Associates, P.O. Box 2432, San Francisco, CA, 94126.

CATALOG LISTING OF PROGRAM INSTRUMENTERS

The following tools have been listed as program instrumenters by one or more of the sources in Appendices A or B.

PROGRAM INSTRUMENTERS	SOURCE OF INFORMATION
1. ADS	[1]
2. AFFIRM	[1,10,11]
3. AISIM	[1]
4. AMPIC	SEE APPENDIX B
5. ARGUS/MICRO	[1,11]
6. ASSIST-1	[1,10,11]
7. ATA-FASP	[1,10]
8. ATA-SAI	[1,11]
9. ATDG	SEE APPENDIX B
10. ATTEST	SEE APPENDIX B
11. BEST/1	[1,10]
12. CADA	[1]
13. CAPTURE/MVS	[1]
14. CASEGEN	[1,2,11]
15. CAVS	[1,10,11]
16. CGJA	[1,10,11]
17. COBOL/ADE	[1]
18. COBOL OPTIMIZER	SEE APPENDIX B
19. COBOL TESTING	SEE APPENDIX B
20. COBOL TRACING	SEE APPENDIX B
21. CONFIGURATION	[1]
22. COTUNE II	[1,10,11]
23. CRYSTAL	[1,10]
24. CUE	[1,10]
25. DARTS	[1]
26. DDPM	[1]
27. DPAD	[1,10]
28. DYNA	SEE APPENDIX B
29. EAVS	[1,10,11]
30. EFFIGY	SEE APPENDIX B
31. EXPEDITER	SEE APPENDIX B
32. FASP	[1,11]
33. FAVS	SEE APPENDIX B
34. FORTRAN OPTIMIZER	SEE APPENDIX B
35. FORTRAN TESTING	SEE APPENDIX B
36. FORTRAN TRACING	SEE APPENDIX B
37. FTN-77 ANALYZ	[1,11]
38. FTN ANALYZER	[1]
39. HARDWARE SIM	[1,11]
40. IFTRAN	[1,10]
41. INSERT	[1,10]

PROGRAM INSTRUMENTERS	SOURCE OF INFORMATION
42. INSTRU	[1,10]
43. IPDS	[1,10,11]
44. ITB	[1,11]
45. JAVS	SEE APPENDIX B
46. JIGSAW	[1,10,11]
47. JOVIAL TCA	[1,11]
48. LOGIC	[1,10,11]
49. LOOK	[1,10]
50. MEDL-P	[1]
51. MONITOR	[1,10]
52. NASA-VATS	[1,10,11]
53. NODAL	[1,10,11]
54. OCM	SEE APPENDIX B
55. PACE	SEE APPENDIX B
56. PACE-C	[1,10]
57. PDS	[1,10]
58. PERCAM	[1,10,11]
59. PET	SEE APPENDIX B
60. POD	[1]
61. PPE	[1,10]
62. PROGLOOK	[1]
63. PRONET	[1]
64. QMC	[1]
65. REFLECT II	[1]
66. RXVP 80	SEE APPENDIX B
67. SADAT	SEE APPENDIX B
68. SALSIM	[1,10,11]
69. SARA	[1,10]
70. SARA-H	[1,10]
71. SARA-U	[1]
72. SARA-III	[1,11]
73. SARA-IV	[1]
74. SCAN/370	[1,11]
75. SCERT	[1]
76. SDVS	[1,10,11]
77. SELECT	SEE APPENDIX B
78. SLIM	[1,10]
79. SMT	[1,10]
80. SOFTOOL 80	[1,10,11]
81. SPRINT	[1,10]
82. SREM	[1,10]
83. SYSTEM MONITOR	[1,2]
84. TAFIRM	[1,10,11]
85. TAPS/AM	[1,10]
86. TATTLE	[1,10,11]
87. TCAT	[1,10,11]
88. TDEM	[1,10,11]

PROGRAM INSTRUMENTERS	SOURCE OF INFORMATION
89. TEST PREDICTOR	[1,10]
90. TEVERE-1	SEE APPENDIX B
91. TFA	[1,10]
92. THE ENGINE	[1]
93. TIMECS	[1]
94. TOOLPAK	[1,10]
95. TPT	[1]
96. TRAILBLAZER	[1,11]
97. TSA/PPE	[1]
98. XPEDITER	SEE APPENDIX B

PROGRAM INSTRUMENTERS USED ACCORDING TO FUNCTION

The following list classifies, by function, the program instrumenter tools cataloged above.

COVERAGE ANALYSIS

- ARGUS/MICRO
- ASSIST-I
- ATA-FASP
- ATA-SAI
- ATTEST
- CADA
- CAVS
- CGJA
- COBOL TESTING
- COTUNE II
- DYNA
- EAVS
- FASP
- FAVS
- FORTRAN TESTING
- FTN-77 ANALYZER
- FTN ANALYZER
- IFTRAN (TM)
- ITB
- JAVS
- JIGSAW
- JOVIAL TCA
- LOGIC
- NODAL
- PACE
- PACE-C
- PDS
- PET
- RXVP80 (TM)
- SADAT
- SOFTOOL 80 (TM)
- TATTLE
- TCAT
- TDEM
- TEST PREDICTOR
- TFA
- THE ENGINE
- TOOLPACK
- TPT
- TRAILBLAZER

ASSERTION CHECKING

- AFFIRM
- ATA-SAI
- CAVS
- EFFIGY
- FTN-77 ANALYZER
- IFTRAN (TM)
- IPDS
- RXVP80 (TM)
- SELECT

SYMBOLIC EXECUTION

- AMPIC
- ASSIST-I
- ATTEST
- CASEGEN
- EFFIGY
- NASA-VATS
- RXVP80 (TM)
- SADAT
- SELECT
- TEVERE-1

SIMULATION

- AISIM
- BEST/1 (TM)
- CONFIGURATOR
- CRYSTAL (TM)
- DARTS
- DDPM
- DPAD
- HARDWARE SIMULA
- MEDL-P
- PERCAM
- POD
- SALSIM
- SARA
- SCAN/370
- SCERT
- SDVS
- SLIM
- SREM
- TAPS/AM

STEP - State-of-the-Art Overview

TUNING

- CAVS
- COBOL OPTIMIZER
- CUE
- FASP
- FAVS
- FORTRAN OPTIMIZER
- FTN-77 ANALYZER
- FTN ANALYZER
- IFTRAN (TM)
- INSERT
- JAVS
- MONITOR
- NODAL
- POD
- PROGLOOK
- QCM
- RXVP80 (TM)
- SADAT
- SARA-H
- SARA-U
- SARA-IV
- SARA-III
- SCAN/370
- SMT
- SOFTOOL 80 (TM)
- SPRINT
- SYSTEM MONITOR
- TIMECS
- TSA/PPE

RESOURCE UTILIZATION

- BEST/1 (TM)
- CAPTURE/MVS (TM)
- CUE
- DARTS
- DDPM
- HARDWARE SIMULA
- LOOK
- PPE
- PRONET
- QCM
- REFLECT II
- SARA-H
- SARA-IV
- SARA-III
- SARA-U
- SMT
- TSA/PPE

TIMING

- CADA
- COBOL/ADE
- COTUNE II
- DARTS
- DDPM
- FASP
- HARDWARE SIMULA
- LOGIC
- MONITOR
- PPE
- PROGLOOK
- REFLECT II
- SMT
- SOFTOOL 80 (TM)
- SPRINT
- TFA
- TIMECS

TRACING

- ADS
- ASSIST-I
- ATA-FASP
- ATA-SAI
- COBOL/DV
- COBOL TRACING
- COBOL/ADE
- EAVS
- EFFIGY
- EXPEDITER
- FORTRAN TRACING
- FTN-77 ANALYZER
- IFTRAN (TM)
- INSERT
- INSTRU
- ITB
- JAVS
- LOGIC
- MONITOR
- RXVP80 (TM)
- SADAT
- SCAN/370
- SELECT
- SOFTOOL 80 (TM)
- TAFIRM
- THE ENGINE
- TOOLPACK
- TPT
- TRAILBLAZER
- XPEDITER

BREAKPOINT CONTROL

- ADS
- EFFIGY

STEP - State-of-the-Art Overview

	FORTRAN TRACING	COBOL TRACING
<u>PATH FLOW TRACING</u> EAVS INSTRU MONITOR SELECT	JAVS SADAT TAFIRM	INSERT LOGIC SCAN/370 TRAILBLAZER
<u>DATA FLOW TRACING</u>		INSTRU
<u>LOGIC FLOW TRACING</u> INSTRU		ASSIST-I
<u>CONSTRAINT EVALUATION</u> RXVP80 (TM)	TEST PREDICTOR	ATDG

PROGRAM INSTRUMENTERS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 1 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
COBOL Optimization Instrumenter/	Instrumenter	Tuning	A/L/S	DEC DG IBM Gould-SEL	FORTRAN	COBOL	YES	$7,000	Softool Corporation
COBOL Testing/	Instrumenter	COBOL Testing	A/L/S	DEC DG IBM Gould-SEL	FORTRAN	COBOL	YES	$7,000	Softool Corporation
COBOL Tracing Instrumenter/	Instrumenter	Tracing Path Flow Tracing	A/L/S	DEC DG IBM Gould-SEL	FORTRAN	COBOL	YES	$7,000	Softool Corporation
Dynamic Analyzer for FORTRAN/DYNA	Dynamic Analyzer	Coverage Analysis Tuning	A/L/S	CDC (EKS II) VAX (UNIX, VMS) IBM (VMS)	FORTRAN 77	FORTRAN 66 or FORTRAN 77	YES	N/A	Boeing Computer Services
FORTRAN Automated Verification System/FAVS	Source Program Analysis Testing Static Analyzer Coverage Analyzer Self-metric Instrumenter Documenter	Coverage Analysis Tuning Static Analysis Dynamic Analysis	A/P/N	Honeywell HG180 UNIVAC 1100	DMATRAN	DMATRAN or FORTRAN IV	YES	$850	General Research Corporation
FORTRAN Optimization Instrumenter/	Instrumenter	Tuning Tracing Data Flow Tracing	A/L/S	DEC DG IBM Gould-SEL	FORTRAN	FORTRAN	YES	$7,000	Softool Corporation

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

PROGRAM INSTRUMENTERS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 2 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
FORTRAN Testing Instrumenter/	Instrumenter	Coverage Analysis	A/L/S	DEC DG IBM Gould-SEL	FORTRAN	FORTRAN	YES	$7,000	Softool Corporation
FORTRAN Tracing Instrumenter/	Instrumenter	Tracing Path Flow Tracing	A/L/S	DEC DG IBM Gould-SEL	FORTRAN	FORTRAN	YES	$7,000	Softool Corporation
JOVIAL Automated Verification System/JAVS	Static Analyzer Instrumenter Coverage Analyzer Assertion Checker Automatic Documenter	Test Completion Analysis Test Data Generation Aid Path Flow Analysis Path Structure Analysis Reachability Analysis Interface Checking Assertion Checking Automatic Documentation Debug Tools	-/-/-	HIS 6180 CDC 6400	JOVIAL J3	JOVIAL	N/A	N/A	General Research Corporation

STATUS 1/2/3	1.	2.	3.
	A = Available	L = License Agreement	S = Supported
	N = Not Available	P = Public Domain	N = Not Supported
	- = No Information Supplied	- = No Information Supplied	- = No Information Supplied

N/A = No Information Available

PROGRAM INSTRUMENTERS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 3 of 3)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
Product Assurance Confidence Evaluator/PACE	Static Analyzer Instrumenter Test Completion Analyzer Path Structure Analyzer Coverage Analyzer	Coverage Analysis Path Flow Analysis Instrumentation Optimization Aid Test Case Selection Aid Regression Testing	-/-/-	CDC 6500/ 7600 UNIVAC 1800	FORTRAN	N/A	N/A	N/A	TRW, SEID Software Product Assurance
Program Evaluator and Tester/PET	Instrumenter Dynamic Assertion Processor Coverage Analyzer	Static Analysis Instrumentation Statistical Analyses Profile Generation Coverage Analysis Assertion Checking	A/L/S	IBM CDC Honeywell UNIVAC	FORTRAN	FORTRAN	YES	N/A	McDonnell-Douglas Corp.
Static and Dynamic Analysis and Test/ SADAT	Static Analyzer Instrumenter Test Data Generator Symbolic Evaluator	Instrumentation Statistical Analysis Coverage Analysis Symbolic Execution Tuning, Tracing, Path Flow Tracing Auditing Data Flow Analysis	-/-/-	IBM 370/ 168 IBM 3033	PL/1	FORTRAN	N/A	N/A	Kernforschungszentrum, Karlsruhe GMBH

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

3.3.5. MUTATION TESTING TOOLS

SUMMARY

An automatic mutation system is a test entry, execution, and data evaluation system that evaluates the quality of test data based on the results of program mutation. In addition to a mutation "score" that indicates the adequacy of the test data, a mutation system provides an interactive test environment and reporting and debugging operations which are useful for locating and removing errors.

TOOL DESCRIPTION

Program Mutation tools are interactive test harnesses that compute the mutation score of test data for a given program (see also Section 2.2.4). The mutation score is a number in the interval [0,1]: high scores indicate high quality test data and low scores indicate low quality test data.

Program mutation assumes that the programs to be tested have been written by experienced programmers ("competent" programmers in the terminology of [1]). Such programs, if they are not correct, are "almost" correct. That is, if such a program is not correct, then it is a mutant of a correct program -- it differs from a correct program only in containing simple errors. A mutation analyzer subjects a program P which is correct on test data D to a series of mutant operators to produce mutant programs which differ from P in simple ways. The mutant programs are then executed by the analyzer on D. If all mutants give incorrect results on execution (they are said to "die"), then it is highly likely that D is adequate and therefore, P is very likely to be correct. On the other hand, if some mutants also give correct results (i.e., they are "live"), then either the live mutants are functionally equivalent to P or D is not adequate. In the latter case, D should be augmented by examination of the non-equivalent live mutants. This procedure forces the tester to closely examine P with respect to the mutants that are still live. If D is determined to be adequate (i.e., if the mutation score is 1), then there still might be complex errors in P which are not simple mutants and have not been explicitly examined in the analysis. This, however, is unlikely since there is a coupling effect which states that test data causing all simple mutants to die is so sensitive that with high probability complex mutants also die on the test data.

There is a variation of mutation, known as weak mutation [11] in which the conditions for killing mutants are modified to improve performance.

Mutation analyzers include tools for Fortran and Cobol programs. The tools vary with respect to scope of language coverage, amount of statistical information returned to the user, and extent of mutation coverage. Language coverage ranges from simple subsets -- useful only in prototype systems -- to complete ANSI subsets. Statistical information and other features available to users include the following:

1) results of executing P on test data
2) instrumentation reports (number of lines executed, etc.)
3) test case handlers
4) raw counts of live/dead mutants
5) accounting of methods of mutant failure (incorrect output, illegal operation, etc.)
6) listing of live mutants
7) listing of dead mutants
8) selected mutants (randomly, by type, etc.)
9) listing of equivalent mutants
10) commands for declaring mutants to be equivalent
11) graphical representations of live/dead mutants
12) test report generators
13) commands for selecting/augmenting available mutant operators
14) commands for selecting the percentage of mutants to be executed
15) commands for automatic equivalent mutant detection
16) predicates for comparing program/mutant output
17) commands for selecting portions of program text to test
18) archive commands

In addition to these features, some analyzers allow redefinition of mutant operators so that tests can be "fine-tuned" to particular applications and environments. One Fortran system accepts multi-module systems.

BASIC OPERATION AND ORGANIZATION

The Fortran mutation analyzers PIMS [4], EXPER [1,6,7], and FMS.3 [12] (as well as TEC/1, the commercial version of EXPER and FMS.3) and the Cobol mutation analyzer CMS.1 [1,2,10] have similar operations (see Figure 1). The subject program is input, parsed into an internal form, test data is accessed and mutant description records are created during a prerun phase. The user may then execute the program on the test data, checking results manually or by means of a preprogrammed "predicate." During the mutation phase, the mutant description records are used to modify the internal form and the resulting mutants are executed interpretively with appropriate accounting during the mutation phase. During the post run phase results and standard reports are displayed.

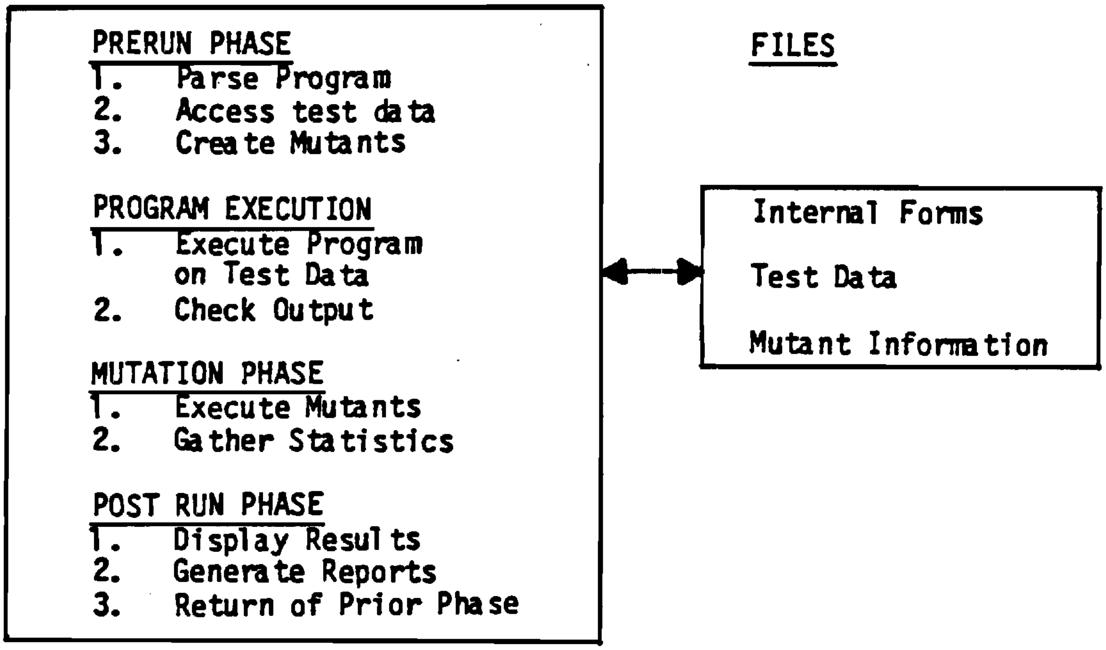


Figure 1. Operational Flow of Mutation Analyzer

The organization of these systems is shown in Figure 2.

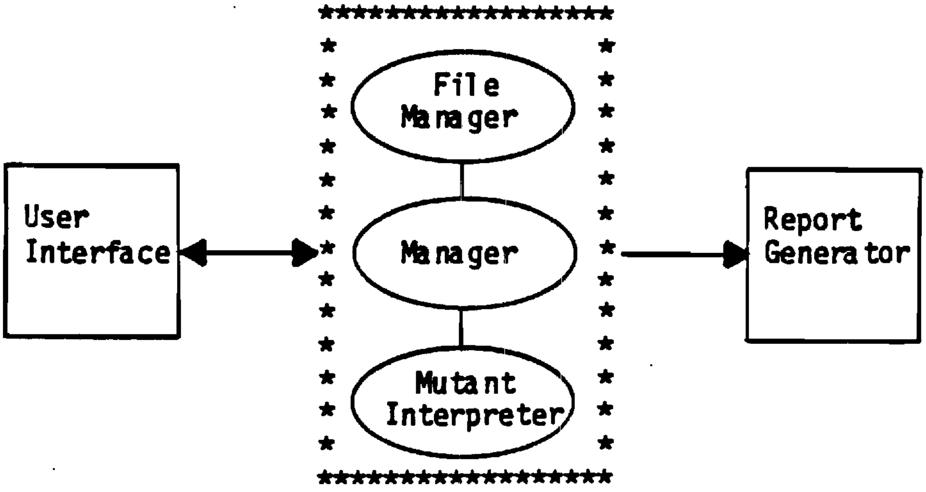


Figure 2. Organization of Mutation Analyzers

A limited mutation system in use at the University of Nottingham [13] has a slightly different structure, since the mutants are induced in the subject program by test editing operations and the resulting mutants are recompiled.

Mutant operators are mappings between the subject program and descriptions of mutants. Mutants operators can change control flow, or any of the objects a program manipulates (constants, scalars, arrays in Fortran, and additionally, files, records and elementary data items in Cobol). At each point of reference to an object, or at each control transfer point, a different, syntactically correct program is constructed by applying a single mutation operator. In addition to control mutations and object replacements, operators are replaced with all other operators of the same type, relational and logical expressions are replaced by true or false.

The following is a list of the mutant operators provided by EXPER, FMS.3, and TEC/1:

1. Constant Replacement
2. Scalar Variable Replacement
3. Scalar Variable for Constant Replacement
4. Constant for Scalar Variable Replacement
5. Array Reference for Constant Replacement
6. Array Reference for Scalar Variable Replacement
7. Constant for Array Reference Replacement
8. Scalar Variable for Array Reference Replacement
9. Array Reference for Array Reference Replacement
10. Source Constant Replacement
11. Data Statement Alteration
12. Comparable Array Name Replacement
13. Arithmetic Operator Replacement
14. Relational Operator Replacement
15. Logical Connector Replacement
16. Absolute Value Insertion
17. Unary Operator Insertion
18. Statement Analysis
19. Statement Deletion
20. Return Statement Replacement
21. GOTO Target Replacement
22. DO Statement End Replacement

Additionally, CMS.1 offers a number of mutation operators which are unique to the Cobol language. For details of all mutant operators, see [1,2,7].

EXPER and FMS.3 offer a limited facility for automatic detection of equivalent mutants. In the EXPER systems, it is possible to assert properties of variables that hold at specific control points in the programs. The assertions constitute invariant properties which can be used to detect equivalence. The underlying theory for equivalence detection is described in [1,3] and essentially rests on the observation that many mutations are actually equivalence preserving optimizing/deoptimizing transformations. In FMS.3 the extraction of properties of variable (e.g., liveness) is carried out by a dataflow analyzer. Automatic equivalence detection is especially useful in detecting such mutants as absolute value replacements, which typically account for up to 75% of the equivalent mutants.

Reporting and display features vary from system to system. All mutation analyzers offer a method to display the remaining live and equivalent mutants. In addition, EXPER and FMS.3 offer several user-oriented features such as a histogram of remaining live mutants by statement (in either textual or sorted order). All systems offer a means for archiving test runs for later resumption.

PERFORMANCE AND EVALUATIONS

Mutation Analyzers have been used in several controlled field tests of production programs (see, e.g., [1,9,10]). Although the number of mutants grows nonlinearly in program size, a number of efficiencies have been incorporated to reduce the total cost of operation. It is not necessary to "turn on" all mutant operators simultaneously. For example, the statement mutants (which provide basic statement coverage) are usually the easiest and least costly to kill. In addition, it is possible to select a fixed percentage of the substitution mutants. These mutants are then sampled and the mutation score provided for the randomly chosen subset. Results in [2] indicate that this technique is well over 95% as effective as complete analysis.

In terms of testing time, the most reliable estimates relate to the usual size measures. In one report study [10], roughly 1000 lines of Fortran source code were completely analyzed in five person days. This represents testing rates that are 2-3 times faster than the code-debug rates for the same code.

Several studies [1,8,9] have compared mutation systems to other testing techniques (see also Section 2.2.4). Since there are explicit mutations for statement coverage and branch coverage, mutation tools provide a strictly more powerful test environment than those coverage tools. On the other hand, mutation tools provide only primitive instrumentation facilities.

LIMITATIONS

Mutation analyzers require significant memory and processor speed. For systems with inefficient memory management, the creation of large numbers of mutant description may cause thrashing or other memory management problems.

Running large programs at 100% of the substitution mutants can create mutant description files of a million or more mutants. In slow processors, this may create unacceptably long compute-bound tasks. On the other hand, 100% mutation execution actually overtests the program and high quality test data can be obtained by sampling a small percentage of the mutants. It may be more advantageous to use mutation analyzers at the unit test level, when test units are less than 1000 lines of source code, combining units and testing interfaces at integration stages.

REFERENCES

[1] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Mutation Analysis.
Report GIT/ICS-79-08, Georgia Institute of Technology, 1979.

[2] A. T. Acree.
On Mutation.
Ph.D. Thesis, Georgia Institute of Technology, 1980.

[3] D. Baldwin and F. Sayward.
Heuristics for Determining Equivalence of Program Mutations.
Technical Report 161, Yale University, 1979.

[4] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
The Design of a Prototype Mutation System for Program Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:623-7, 1978. Also reprinted in Tutorial: Automated Tools for Software Engineering, E. F. Miller, Editor, IEEE Computer Society, 1979.

[5] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Program.
7th ACM Symposium on Principles of Programming Languages, January 1980.

[6] T. A. Budd, R. Hess, and F. G. Sayward.
EXPER Implementor's Guide.
Department of Computer Science, Yale Univeristy.

[7] T. A. Budd.
Mutation Analysis of Program Test Data.
Ph.D. Thesis, Yale University, 1980.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Hints on Test Data Selection: Help for the Practicing Programmer.
Computer, Vol.11(4):34-43, April 1978.

[9] R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Program Mutation: A New Approach to Program Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 107-26.
Infotech International, 1979.

[10] J. M. Hanks.
Testing Cobol Programs by Mutation: Volume I - Introduction to the CMS.1 System, Volume II - CMS.1 System Documentation.
Report GIT/ICS-80-04, Georgia Institute of Technology, 1980.

[11] W. E. Howden.
Weak Mutation Testing and Completeness of Test Sets.
IEEE Transactions on Software Engineering, Vol.SE-8(4):371-9, July 1982.

[12] A. Tanaka.
Equivalence Testing for Fortran Mutation System Using Data Flow Analysis.
Department of Information and Computer Science, Georgia Institute of Technology, 1981.

[13] M. R. Woodward, M. A. Hennell, and D. Hedley.
A Limited Mutation Approach to Program Testing.
University of Nottingham, Nottingham, UK, 1980.

CATALOG LISTING OF MUTATION TESTING TOOLS

The following tools have been listed as mutation testing tools by one or more of the sources in Appendices A or B.

MUTATION TESTING TOOLS	SOURCE OF INFORMATION
1. PORTABLE FORTRAN MUTATION	SEE APPENDIX B
2. TEC/1	SEE APPENDIX B

MUTATION TESTING TOOLS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 1 of 1)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
FORTRAN Mutation System/TEC/1	Automatic Mutation System	Test Harness and Driver Computes Mutation Scores Produces Test Reports and Statistics	A/-/N	PRIME 550	FORTRAN	FORTRAN	YES	N/A	Georgia Institute of Technology
Portable FORTRAN Mutation System/	Automatic Mutation System	Test Harness and Driver Computes Mutation Scores	A/P/N	N/A	N/A	FORTRAN	YES	N/A	T. A. Budd University of Arizona

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

3.4. TEST SUPPORTING TOOLS

Some testing tools have the main function of test execution coordination, rerunning test cases for a modified program (regression testing), or comparing the resulting output. More sophisticated system provide test environment and documentation aid to support other testing techniques. In this section only test drivers and comparators are described.

3.4.1. AUTOMATIC TEST DRIVERS

SUMMARY

Automatic test drivers are software systems that simulate an environment for running module tests. They provide standard notation for specifying test cases and automate the testing process. Some systems also compare the resulting output to the expected output and report any discrepancy. There are systems that operate on object modules and others that operate on high level languages. Benefits of using automatic test drivers include standardization of test case description and ease of regression testing. The main drawback is the difficulty in learning and writing a test language. Experience using automatic test drivers indicates that they are effective debugging tools and help improve productivity of programmers.

GENERAL DESCRIPTION

Automatic test drivers, test harnesses, or testbeds are systems that provide an environment for running software component tests, simulating missing modules or sub-systems [5].

In order to test an individual module, a testbed must provide a driver and stubs. A driver is auxiliary code that sets up an appropriate environment and calls the tested module [1]. Stubs replace low-level subprograms called by the module that are not available at the time of testing. The testbed must also provide data interfaces such as input/output parameters, files, messages, and common blocks. This means that the system must have the ability to allocate storage, bind the external references from the test module to it and initialize the data values prior to each execution of the test module. Furthermore, the system must satisfy all of the data requests made by the module upon its parallel processes, peripheral devices and its subordinate modules [5,9].

Automatic software test drivers provide a standard notation for specifying software tests, a standard set-up for verifying software tests, automate the verification of test results, and eliminate the need for writing separate drivers and stubs for the module and subsystems testing [7].

Although all test drivers perform the same basic function, they differ in their level of sophistication. Simple test drivers merely reinitiate the program with various input sets and save the output. The more sophisticated testbeds accept data inputs, expected outputs, the name of routines to be executed, values to be returned by called routines, and other parameters. These systems also compare the actual output with the expected output and issue concise reports of the performance [1]. Others also provide facilities for static and dynamic testing.

CLASSIFICATION AND BASIC OPERATION

Automatic software test drivers can be classified into two main categories according to the type of the target modules:

1) The first type operates on object modules and is independent of the language of the target module. An example is the Automatic Unit Test (AUT) developed at IBM. The AUT provides a formal language that is used to code the test cases and the stubs. The driver controls the execution of the target module and generates a test-execution report listing errors in the target program outputs. Drawbacks of this tool include the difficulty of learning the low-level language and the lack of facilities for simulating I/O devices and files [3,7].

2) The second type of driver operates at the source code level. Source-level testing offers dual advantages: (1) test cases may be specified in terms of the internal structure of the target program and (2) measures of testing thoroughness, based on the program's source representation, may be taken while a test procedure is executing [7]. However, this type of module drivers is highly dependent on the language of the target module. It is more difficult to implement and requires language translators for different target languages.

Examples of this type are the TPL/F, TPL/2.0 systems developed at General Electric Corporation Research and Development Center. TPL/F is the first generation system which operates on Fortran Software. Its test procedure requires three steps: initialization, execution, and verification of the results. The main drawback of the system is its lengthy and difficult to read test procedure. TPL/2.0 is the improved version. It provides a more concise form of environment definition and significantly reduces the labor required for coding and maintaining test procedures by automating the initial generation and subsequent revision of the module outputs [2,7].

Benefits and Drawbacks

The benefits of module drivers can be summarized as follows:

1) reduction in testing effort,
2) standardization of test cases,
3) ease of regression testing (see Section 2.1), and
4) the automatic verification of results forces the programmer to state explicitly the expected outputs and therefore lessens the "eye seeing what it wants to see" problem [6].

However, there are many drawbacks that should be considered:

1) Additional work and difficulty associated with learning a specific testing language.
2) Test procedures of some drivers are lengthy and difficult to read. As the result, programmers sometimes feel that writing test procedures is tedious, lacking the challenge and interest of coding a program. However, sophisticated test drivers usually simplify the environment definition process, especially those interactive systems such as PRUFSTAND and TESTMANAGER.
3) The language dependent nature of the module drivers that operate on source-level code makes it difficult to test multiple target languages.

TOOL EVALUATIONS

Automatic test drivers are very effective debugging tools and improve productivity. TESTMANAGER users have claimed a 30% reduction in debugging time, and considerably reduced maintenance due to better tested programs [5]. The EXPEDITER system, which provides facilities for unit testing, problem isolation, and verification is responsible for improving productivity in a COBOL environment from 10 lines of procedure division code per programmer per day to 45 lines [4].

REFERENCES

[1] W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

[2] J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RACS-TR-80-13, Interim Report, Rome Air Development Center, NY, February 1980.

[3] C. A. Heuerman, G. J. Myers, and J. H. Winterton.
Automated Test and Verification.
IBM Technical Disclosure Bulletin, Vol.17(7):2030-5, December 1974.

[4] R. C. Houghton, Jr.
Software Development Tools.
NBS Special Publication 500-88, National Bureau of Standards, 1982.

[5] Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography, pages 212-28.
Infotech International, 1979.

[6] G. J. Myers
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

[7] D. J. Panzl.
A Language for Specifying Software Tests.
National Computer Conference, AFIPS Proceedings,, Vol.47:609-19, 1978.

[8] H. M. Sneed.
Prufstand - A Testbed for Systems Software Components.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 246-70. Infotech International, 1979.

REFERENCES NOT CITED IN THE TEXT

D.J. Panzl.
Test Procedures: A New Approach to Software Verification.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 477-85.

CATALOG LISTING OF AUTOMATIC TEST DRIVERS

The following tools have been listed as automatic test drivers by one or more of the sources in Appendices A or B.

AUTOMATIC TEST DRIVERS	SOURCE OF INFORMATION
1. AUTORETEST	SEE APPENDIX B
2. DATAMACS	SEE APPENDIX B
3. DRIVER	[1,10,11]
4. EXPEDITER	SEE APPENDIX B
5. PRUFSTAND	SEE APPENDIX B
6. SEF	[1,2,10,11]
7. TESTMANAGER	SEE APPENDIX B
8. XPEDITER	SEE APPENDIX B

AUTOMATIC TEST DRIVERS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 1 of 1)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
AUTOMATED UNIT TEST /AUT	Test Driver	Regression Testing Simulation of Test Environment	-/L/-	IBM 360, 370	N/A	BAL PL/q COBOL	N/A	$1,200	IBM
/EXPEDITER	Test Driver	Tracing Regression Testing	A/L/S	N/A	BAL	N/A	YES	N/A	Application Development System, Inc.
/TESTMANAGER	Test Driver	Regression Testing	A/L/-	IBM 360,370 30XX, 43XX ICL 1900	N/A	Assembly COBOL CORAL	N/A	$ 9,000 to $14,000	MSP Incorporated
The Programmer Productivity Tool for the 80's/XPEDITOR	Test Driver Test Bed	Regression Testing Test Environment	A/-/S	IBM 360,370	N/A	N/A	NO	$45,000	Application Development System, Inc.

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

3.4.2. COMPARATORS

SUMMARY

A comparator is a program that compares two versions of data to identify their differences. It is used in the validation process to limit the scope of reverification of revised software. The main differences among comparators are the form of the data and the flexibility in specifying tolerance for each comparison.

GENERAL DESCRIPTION

A comparator is a program that compares two versions of data to identify the differences between the two versions. The data may be program code, output of an execution, or data files [1,2,3,4].

Comparators serve primarily as tools for validating modified software to assure that the revised software contains only particular modifications. The use of a comparator helps limit the scope of reverification that must be performed on modified programs.

Other than the form of data to be compared, comparators differ in the level of flexibility in specifying some tolerance, i.e., allowing a certain number of differences in comparisons. AUTO-RETEST provides an automated comparison between selected old and new test parameters. The system also provides flexibility in specifying a tolerance criterion for each comparison [5].

DIFFS is a file comparator that compares files of fixed-length records with user selectable options to omit portions of the record from the comparison. The system can detect and recover from missing or extra records in either files [5].

TDBCOMP compares and summarizes the difference between two data bases, where one data base is on tape and the other is active on disk [5].

Output comparators are usually included in automatic test drivers. FADEBUG-I is an integrated system which performs the function of a test driver, an output comparator, and a static analyzer. The system initiates execution of a module using the user provided input data. After the execution, it compares the actual output data against the desired output data and reports discrepancies [1].

REFERENCES

[1] J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY, February 1980, pages 4-88 to 4-91.

[2] H. R. Downs.
Automated Tools for the Verification of Computer Programs.
Transaction of the American Nuclear Society and the European Nuclear Society, 1980 International Conference, November 16-21, 1980, Washington, DC, pages 253-4.

[3] Infotech State of the Art Report, Software Reliability, Volume 2: Invited Papers, pages 184-215.
Infotech International, 1977.

[4] D. J. Reifer and S. Trattner.
A Glossary of Software Tools and Techniques.
Computer, Vol.10(7):52-60, July 1977.

[5] Software Engineering Automated Tools Index.
Software Research Associates, P. O. Box 2432, San Francisco, CA 94126.

CATALOG LISTING OF COMPARATORS

The following tools have been listed as comparators by one or more of the sources in Appendices A or B.

	COMPARATORS	SOURCE OF INFORMATION
1.	AUTORETEST	SEE APPENDIX B
2.	COMPARISON	[1]
3.	CCS	[1,2,10]
4.	COMPARE DBCOMP	[1]
5.	DECKBOY COMPAR	[1]
6.	DIFFS	SEE APPENDIX B
7.	DRIVER	[1,10,11]
8.	FADEBUG-1	SEE APPENDIX B
9.	FASP	[1,11]
10.	MSEF	[1]
11.	PROG COMP ANAL	[1]
12.	PWB FOR VAX/VMS	[1]
13.	SCAN/370	[1,11]
14.	SOFTOOL 80	[1,10,11]
15.	TBDCOMP	SEE APPENDIX B
16.	TRAILBLAZER	[1,11]
17.	VIRTUAL OS	[1]

COMPARATORS
TESTING TOOL DATA SHEETS SUMMARIES
(Table 1 of 1)

TOOL NAME/ACRONYM	TOOL TYPE	FUNCTION PERFORMED	STATUS	HARDWARE	IMPLEMENT LANGUAGE	TARGET LANGUAGE	PORTABLE	COST	SOURCE
/AUTORETEST	Comparator Test Driver	Test Data Management Regression Testing Automatic Comparison between selected old and new test parameters	-/L/-	IBM 360, 370	FORTRAN IV Assembly	N/A	N/A	N/A	TRW, Defense Systems Software Department
/DIFFS (TM)	File Comparator	File Comparison	A/L/S	N/A	COBOL SCOBOL	N/A	YES	$500	Software Consulting Services
FACOM Automatic Debug/FADEBUG-I	Output Comparator Anomaly Detector	Comparison I/O Specification Analysis Debug Aid	A/L/S	FACOM 230-	Assembly	N/A	N/A	N/A	Fujitsu, Ltd.
TDBCOMP Program/ TDBCOMP	Comparator Maintenance Tool	Automatic Data Comparison	-/L/-	CDC 3XXX	JOVIAL J4	N/A	N/A	N/A	TRW, Operational Software Operations

STATUS 1/2/3

1.
A = Available
N = Not Available
- = No Information Supplied

2.
L = License Agreement
P = Public Domain
- = No Information Supplied

3.
S = Supported
N = Not Supported
- = No Information Supplied

N/A = No Information Available

CHAPTER 4

COMPREHENSIVE BIBLIOGRAPHY

This bibliography is intended to give the reader a comprehensive guide to the current literature on software test and evaluation. Also included are references on the related areas of reliability, software metrics, program proving, and software development methodology. The bibliography is organized to follow the topics in Chapters 1-3. If a cited source treats multiple topics, the source is listed under all relevant subsections. Each subsection is divided into three parts. The first part consists of textbooks, reference books, and book-length reports treating the current topic. The next two sections list articles and shorter reports on the current topic; these are classified according to whether the articles are survey or detailed. Survey articles are generally those which are suitable for an introduction to the topic, listing important definitions, issues, and techniques. Detailed articles give the reader a research-level view of the field and require considerably more background in the topic than the survey articles.

The reader should note that not all the sources cited in this bibliography have appeared in formally refereed media. Many present work in progress that will be reported in more polished form elsewhere.

1. THEORY OF TESTING

Books

H. K. Berg, W. E. Boebert, W. R. Franta, and T. G. Moher.
Formal Methods of Program Verification and Specification.
Prentice-Hall, Inc., Englewood Cliffs, NJ.

R. S. Boyer and J. S. Moore.
A Computational Logic.
Academic Press, New York, 1979.

C-L. Chang and R. C-T. Lee.
Symbolic Logic and Mechanical Theorem Proving.
Academic Press, New York, 1973.

R. A. DeMillo.
Program Mutation: An Approach to Software Testing.
Report GIT/ICS-83-03, Georgia Institute of Technology,
January 1983.

L. P. Deutsch.
An Interactive Program Verifier.
Ph.D. Thesis, University of California, Berkeley, May 1973.

T. Gilb.
Software Metrics.
Winthrop Publishers, Inc., Cambridge, MA, 1977.

J. Gourlay.
Theory of Testing Computer Programs.
Ph.D. Thesis, University of Michigan, 1981.

D. Gries.
The Science of Programming.
Springer-Verlag, New York, 1981.

J. V. Guttag.
The Specification and Application to Programming of Abstract Data Types.
Ph.D. Thesis, University of Toronto, Report CSRG-59, 1975.

Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography.
Infotech International, 1979.

Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers.
Infotech International, 1979.

Z. Manna.
Mathematical Theory of Computation.
McGraw-Hill, 1974.

G. J. Myers.
Software Reliability: Principles and Practices.
Wiley Publishers, 1976.

A. J. Perlis, F. G. Sayward and M. Shaw.
Software Metrics: An Analysis and Evaluation.
MIT Press, Cambridge, MA, 1981.

R. A. Thayer, M. Lipow and E. C. Nelson.
Software Reliability.
North-Holland, Amsterdam, 1978.

P. Wegner.
Research Directions in Software Technology.
MIT Press, Cambridge, MA, 1979.

Survey Articles

L. A. Belady.
On Software Complexity.
Proceedings of the IEEE Workshop on Quantitative Software Models, pages 90-94. IEEE, Piscataway, NJ, 1979.

B. Curtis.
Measurement and Experimentation in Software Engineering.
Proceedings of the IEEE, Vol.68(9):1144-57, September 1980.

R. A. DeMillo.
Validating Computer Software - Two Views.
Transactions of the American Nuclear Society, Vol.35:251-2, November 1980.

R. A. DeMillo, R. J. Lipton, and A. J. Perlis.
Social Processes and Proofs of Theorems and Programs.
Communications of the ACM, May 1979.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Hints on Test Data Selection: Help for the Practicing Programmer.
Computer, Vol.11(4):34-41, April 1978.

S. L. Gerhart and L. Yelowitz.
Observations of Fallibility in Applications of Modern Programming Methodologies.
IEEE Transactions on Software Engineering, Vol.SE-2(3):195, September 1976.

J. B. Goodenough.
A Survey of Program Testing Issues.
In Research Directions in Software Technology, Peter Wegner, Editor, pages 316-42. MIT Press, Cambridge, MA, 1979.

J. B. Goodenough and S. L. Gerhart.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering, Vol.SE-1(2):156-73, June 1975.

S. L. Hantler and J. C. King.
An Introduction to Proving the Correctness of Programs.
ACM Computing Surveys, Vol.8(3):332-53, September 1976.

D. Levy, D. Guy, and J. Ronback.
A Place for Metrics in Software Development.
Telesis (Canada), Vol. 6(5):17-22, October 1979.

E. F. Miller, Jr.
Notes on the Theoretical Foundations of Testing.
In Tutorial: Program Testing Techniques, pages 51-4. IEEE Computer Society, Piscataway, NJ, 1977.

Z. Mital.
Software Reliability.
Informatyka (Poland), Vol.14(11):7-9, November 1979.

Proceedings of an ACM Conference on Proving Assertions about Programs. SIGPLAN Notices, Vol.7(1), January 1972. Also reprinted in SIGACT News, Vol.14, January 1972.

F. R. Richards.
Computer Software Testing, Reliability Models, and Quality Assurance.
Naval Postgraduate School, Monterey, CA, July 1974.

R. J. Rubey.
Quantitative Aspects of Software Validation.
SIGPLAN Notices, Vol.10(6):246-51, June 1975.

R. J. Rubey, J. A. Dana, and P. W. Biche.
Quantitative Aspects of Software Validation.
IEEE Transactions on Software Engineering, Vol.SE-1(2):150-5, June 1975.

A. S. Tanenbaum.
In Defense of Program Testing or Correctness Proofs Considered Harmful.
SIGPLAN Notices, Vol.11(5):64-8, May 1976.

Detailed Articles

A. L. Ambler, et al.
GYPSY: A Language for Specification and Implementation of Verifiable Programs.
Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol.12(3), March 1977.

A. Ballard and D. Tsichritzis.
System Correctness.
SIGPLAN Notices, Vol.8(9):38-41, September 1973.

D. E. Bell and L. J. LaPadula.
Secure Computer Systems.
Report ESD-TR-73-278, MITRE Corporation, Beford, MA.
November 1973.

R. G. Bennetts.
A Comment on Reliability Evaluation of Software.
Proceedings of the NATO Advanced Study Institute on Generic Techniques in Systems Reliability Assessment, July 17-28, 1973, Liverpool, England, pages 55-9.

P. I. P. Boulton and M. A. R. Kittler.
Estimating Program Reliability.
Computer Journal (GB), Vol.22(4):328-31, November 1979.

J. B. Bowen.
Standard Error Classification to Support Software Reliability Assessment.
National Computer Conference, AFIPS Proceedings, May 19-22, 1980, Anaheim, CA, pages 697-705.

T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
The Design of a Prototype Mutation System for Program Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:623-7, 1978.

R. M. Burstall.
Proving Correctness as Hand Simulation with a Little Induction.
Proceedings of the International Federation of Information Processing Societies, North Holland, Amsterdam, The Netherlands, 1974, pages 308-12.

R. H. Campbell and A. N. Haberman.
The Specification of Process Synchronization by Path Expressions.
Lecture Notes on Computer Science, Vol.16, 1974.

J. C. Cherniavsky.
On Finding Test Data Sets for Loop Free Programs.
Information Processing Letters (Netherlands), Vol.8(2):106-7,
February 15, 1979.

R. M. Cohen.
Formal Specifications for Real-time Systems.
Proceedings of the Seventh Texas Conference on Computing
Systems, October 1978.

A. De Luca and A. Restivo.
On Some Properties of Local Testability.
Automata, Languages and Programming, Seventh Colloquium, July
14-18, 1980, Noordwijkerhout, Netherlands, pages 385-93.

R. W. Elliott, M. P. Marchbanks, Jr., M. G. McWilliams,
L. J. Ringer, and D. B. Simmons.
Measuring Computer Software Reliability.
Computers and Industrial Engineering (GB), Vol.2(3):141-51, 1978.

L. Flon and A. N. Habermann.
Towards the Construction of Verifiable Software Systems.
Proceedings of Conference on Data: Abstraction, Definition, and
Structure, SIGPLAN Notices, Vol.B(2):141-8, 1976.

R. W. Floyd.
Assigning Meanings to Programs.
Proceedings of the Symposium on Applied Mathematics,
Vol.19:19-32, American Mathematical Society, Providence, RI,
1967.

E. H. Forman.
Statistical Models and Methods for Measuring Software
Reliability.
George Washington University, Washington, DC.

M. Geller.
Test Data as an Aid in Proving Program Correctness.
Communications of the ACM, Vol.21(5):368-75, May 1978.

S. L. Gerhart.
Program Validation.
Computing Systems Reliability, pages 66-108, 1979.

A. L. Goel.
A Software Error Detection Model with Applications.
Second Software Life Cycle Management Workshop, August 21-22,
1978, Atlanta, GA, pages 133-9.

A. L. Goel.
Software Error Detection Model with Applications.
Journal of Systems and Software, Vol.1(3):243-9, 1980.

J. D. Gould and P. Drongowski.
An Exploratory Study of Computer Program Debugging.
Human Factors, Vol.16(3):258-77, June 1974.

R. Hamlet.
Test Reliability and Software Maintenance.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 315-20.

R. Hamlet.
Testing Programs with Finite Sets of Data.
Computer Journal, Vol.20(3):232-7, March 1977.

S. L. Hantler.
The Relation Between Symbolic Program Testing and Program Verification.
Computer Science Conference, February 18-20, 1975, Washington, DC, page 41.

M. A. Hennell, D. Hedley, and M. R. Woodward.
Quantifying the Test Effectiveness of Algol 68 Programs.
SIGPLAN Notices, Vol.12(6):36-41, June 1977.

M. A. Herndon and A. P. Keenan.
Analysis of Error Remediation Expenditures During Validation.
Proceedings of the 3rd International Conference on Software Engineering, May 10-12, 1978, Atlanta, GA, pages 202-6.

C. A. Hoare.
Proof of a Program: FIND.
Communications of the ACM, Vol.14(1), January 1971.

W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions on Software Engineering, Vol.SE-2(3):208-14, September 1976.

W. E. Howden.
Theoretical and Empirical Studies of Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-4(4):293-7, July 1978.

J. C. King.
Proving Programs to be Correct.
IEEE Transactions on Computers, Vol.C-20(11):1331-6, November 1971.

P. Knezevic.
Programs for Correctness Testing of Other Programs in a Real Time Processor System.
Informatica 78 XIII Yugoslav International Symposium on Information Processing, October 2-7, 1978, Bled, Yugoslavia.

B. Littlewood.
A Bayesian Differential Debugging Model for Software Reliability.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 511-9.

B. Littlewood.
The Littlewood-Verrall Model for Software Reliability Compared with Some Rivals.
Journal of Systems and Software, Vol.1(3):251-8, 1980.

B. Littlewood and J. L. Verrall.
Likelihood Function of a Debugging Model for Computer Software Reliability.
IEEE Transactions on Reliability, Vol.R-30(2):145-8, June 1981.

R. L. London
Proof of Algorithms: A New Kind of Certification.
Communications of the ACM, Vol.13(6), June 1970.

Z. Manna and R. Waldinger.
The Logic of Computer Programming.
IEEE Transactions on Software Engineering, Vol.SE-4:199-229, 1978.

S. N. Mohanty.
Models and Measurements for Quality Assessment of Software.
ACM Computing Surveys, Vol.11(3):251-75, September 1979.

P. B. Moranda.
Limits to Program Testing with Random Number Inputs.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 521-6.

P. B. Moranda.
Prediction of Software Reliability During Debugging.
Proceedings of the 1975 Annual Reliability and Maintainability Symposium, January 28-30, 1975, Washington, DC, pages 327-32.

P. B. Moranda and Z. Jelinski.
Software Reliability Predictions.
I. Reliability Estimates Based on Failure-Rate Models.
Proceedings of the 6th Triennial World Congress of the International Federation of Automatic Control, August 24-30, 1975, Boston and Cambridge, MA.

M. Morelli.
Software Reliability - Principles and Criteria for Online Testing of the Software.
Manage. and Inf. (Italy), Vol.27(5):293-6, May 1978.

J. D. Musa.
Software Reliability Measurement.
Journal of Systems and Software, Vol.1(3):223-41, 1980.

S. C. Ntafos and S. L. Hakimi.
On Structured Digraphs and Program Testing.
IEEE Transactions on Computers, Vol.C-30(1):67-71, January 1981.

M. R. Paige.
Program Graphs, an Algebra, and Their Implication for Programming.
IEEE Transactions on Software Engineering, Vol.SE-1:286-91, September 1975.

F. J. Schick and R. W. Wolverton.
An Analysis of Competing Software Reliability Models.
IEEE Transactions on Software Engineering, Vol.SE-4(2):104-20, March 1978.

N. F. Schneidewind.
Application of Program Graphs and Complexity Analysis to Software Development and Testing.
IEEE Transactions on Reliability, Vol.R-28(3):192-8, August 1979.

M. L. Shooman.
Managing Software Testing Using Reliability Estimates.
National Conference on Software Test and Evaluation, February 1983.

A. N. Sukert.
A Four-Project Empirical Study of Software Error Prediction Models.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 577-82.

A. N. Sukert.
An Investigation of Software Reliability Models.
Proceedings of the 1977 Annual Reliability and Maintainability Symposium, January 18-20, 1977, Philadelphia, PA, pages 478-84.

P. K. Suri and K. K. Aggarwal.
Reliability Evaluation of Computer Programs.
Microelectronics and Reliability (GB), Vol.20(4):465-70, 1980.

K.-C. Tai.
Program Testing Complexity and Test Criteria.
IEEE Transactions on Software Engineering, Vol.SE-6(6):531-8, November 1980.

J. K. Wall and P. A. Ferguson.
Pragmatic Software Reliability Prediction.
Proceedings of the 1977 Annual Reliability and Maintainability Symposium, January 18-20, 1977, Philadelphia, PA, pages 485-8.

E. J. Weyuker and T. J. Ostrand.
Theories of Program Testing and the Application of Revealing Subdomains.
IEEE Transactions on Software Engineering, Vol.SE-6(3):236-46, May 1980.

P. M. Zislis.
Semantic Decomposition of Computer Programs: An Aid to Program Testing.
Acta Informatica (Germany), Vol.4(3):245-69, 1975.

2. SOFTWARE TESTING

2.1 TESTING STRATEGIES

Books

M. S. Deutsch.
Software Verification and Validation Realistic Project Approaches.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

R. Dunn and R. Ullman.
Quality Assurance for Computer Software.
McGraw Hill Book Company, New York, 1982.

R. Glass.
Software Reliability Guidebook.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

Infotech State of the Art Report, Software Reliability, Volume 2: Invited Papers.
Infotech International, 1977.

Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography.
Infotech International, 1979.

M. Jackson.
Principles of Program Design.
Academic, London, 1975.

G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

R. A. Thayer, M. Lipow and E. C. Nelson.
Software Reliability.
North-Holland, Amsterdam, 1978.

D. Van Tassel.
Program Style, Design, Efficiency, Debugging and Testing.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1974.

E. Yourdan and L. L. Constantine.
Structured Design.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

E. Yourdon and L. L. Constantine.
Structured Design Fundamentals of a Discipline of Computer Program and Systems Design.
Prentice-Hall, Inc., Englewood Cliffs, NJ.

Survey Articles

W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

M. A. Branstad, J. C. Cherniavsky, and W. R. Adrion.
Validation, Verification, and Testing for the Individual Programmer.
Computer, Vol.13(12):24-30, December 1980.

J. A. Dobbins and R. D. Buck.
Software Quality in the 80's.
Proceedings of Trends and Applications 1981. Advances in Software Technology, May 28, 1981, Gaithersburg, MD, pages 31-7.

M. Finfer, et al.
Software Debugging Methodology.
Final Technical Report, RADC-TR-79-57 (three volumes), Rome Air Development Center, NY, April 1979.

L. Gmeiner and U. Voges.
Methods, Criteria and Automatic Tools for Software Testing.
Practice in Software Adaption and Maintenance. Proceedings for the SAM Workshop, April 5-6, 1979, Berlin, Germany, pages 183-92.

J. B. Goodenough and C. L. McGowan.
Software Quality Assurance - Testing and Validation.
Proceedings of the IEEE, Vol.68(9):1093-8, September 1980.

W. E. Howden.
Life-Cycle Software Validation.
In Life-Cycle Management, State of the Art Report, pages 101-16, 1980.

W. E. Howden.
Life-Cycle Software Validation.
Computer, Vol.15(2):71-78, February 1982.

W. E. Howden and E. Miller.
A Survey of Static Analysis Methods.
Tutorial: Software Testing and Validation Techniques, IEEE, 1981, pages 101-15.

J. C. Huang.
An Approach to Program Testing.
ACM Computing Surveys, Vol.7(3):113-28, September 1975.

A. Lepper.
Testing Large-Scale Systems.
Computer Weekly (GB), Vol.16(383):6,16, March 7, 1974.

E. F. Miller, Jr.
Program Testing: Art Meets Theory.
Computer, Vol.10(7):42-51, July 1977.

C. V. Ramamoorthy, S. F. Ho, and H. H. So.
The Status and Structure of Software Testing Procedures.
Proceedings of COMPCON 77, February 28 - March 3, 1977, San Francisco, CA, pages 367-9.

M. Schindler.
Software Practice -- A Scarce Art Struggles to Become a Science.
Electronic Design, pages 85-102, July 22, 1982.

P. Schmitz, R. Van Megen, and H. Bons.
Methods and Techniques of Dynamic Program Testing.
Practice in Software Adaption and Maintenance. Proceedings for the SAM Workshop, April 5-6, 1979, Berlin, Germany, pages 209-21.

K. Schroeder.
The Ten Deadly Sins Against the Software Test.
Online-Adl-Nachr. (Germany), No.10:822-5, October 1977.

L. G. Stucki.
Tutorial on Program Testing Techniques.
Slide Masters, COMPSAC-77, November 8-11, 1977, Chicago, IL.

Z. Varkonyi.
Program Testing in the Light of Modern Program Development.
Inf. Elektron. (Hungary), Vol.10(2):134-42, 1975.

V. Zsolt.
Up to Date Methods and Means of Program Testing.
Inf. Elektron. (Hungary), Vol.13(5):276-83, 1978.

Detailed Articles

J. Allain.
Reliability Approach to Software.
Second International Conference on Reliability and Maintainability, September 8-12, 1980, Tregastel, France, pages 60-6.

B. I. Blum.
The Life Cycle -- A Debate For Alternative Models
Software Engineering Notes, Vol. 7(12), 1982.

B. W. Boehm.
Software Engineering.
IEEE Transactions on Computers, Vol.C-25(12), December 1976.

E. M. Boehm, R. K. McClean, and D. D. Urfrig.
Some Experience with Automated Aids to the Design of Large-Scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1:125-33, 1975.

B. Brehme, W. Pahlke, and M. Thomas.
Organisation for Testing in an OS/ES Installation.
Rechentech. Datenverarb. (Germany), Vol.13(1):21-4, January 1976.

R. A. Brook.
Progressive Integration of Hardware and Software.
IEEE Colloquium on Bringing Hardware and Software Together in Microprocessor Systems, March 9, 1981, London, England.

J. C. Caille and J. Heller.
Tests in a Data Base Context.
Convention Informatique, September 20-24, 1976, Paris, France, pages 26-8.

R. Carey and M. Benedic.
The Control of a Software Test Process.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 327-33.

B. A. Carre.
Software Validation. II. Semantic Analysis.
In Advanced Techniques for Microprocessor Systems, pages 119-25, 1980.

L. A. Clarke.
Testing - Achievements and Frustrations.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 310-4.

M. T. Compton.
Testing with Inspection Procedures (Computer Software).
Conference Digest of the International Electrical, Electronics Conference and Exposition, October 2-4, 1979, Toronto, Canada, pages 22-3.

L. H. Cooke, Jr.
Express Testing (Programs).
Datamation, Vol.24(9):219-22, September 1978.

D. W. Cooper.
Adaptive Testing.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 102-5.

E. B. Daily.
Software Development.
Proceedings of European Computing Review, Infotech International, Ltd., 1978.

R. V. Dmitrishin.
On the Testing of Optimization Programs.
Izv. Vuz Radioelectron. (USSR), Vol.21(6):106-9, June 1978.

E. G. Dupnick.
A Zero-One Integer Programming Solution for Determining the Minimum Number of Test Cases Required for Fortran Program Checkout.
Bulletin of the Operations Research Society of America, Vol.21(2):B12, 1973.

J. W. Duran and S. Ntafos.
A Report on Random Testing.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 179-83.

J. O. Ellis.
A Methodology for Software Testing in the Material Acquisition Process.
Proceedings of the Sixth Texas Conference on Computing Systems, November 14-15, 1977, University of Texas, Austin, TX.

W. R. Elmendorf.
Computer-Assisted Design of Program Test Libraries.
IBM Technical Disclosure Bulletin, Vol.16(3):804-7, August 1973.

A. Endres and W. Glatthaar.
A Complementary Approach to Program Analysis and Testing.
Information Systems Methodology, October 10-12, 1978, Venice, Italy, pages 380-401.

The Evaluation and Organization of Software Testing.
Software World (GB), Vol.10(4):2-13, 1979.

M. E. Fagan.
Design and Code Inspections to Reduce Errors in Program Development.
IBM Systems Journal, Vol.15(3):182-211, 1976.

R. E. Fairley.
Tutorial: Static Analysis and Dynamic Testing of Computer Software.
Computer, Vol.11(4):14-23, April 1978.

J. Farradane and D. Thompson.
The Testing of Relational Indexing Procedures by Diagnostic Computer Programs.
Journal of Information Science Principles and Practices (Netherlands), Vol.2(6):285-97, December 1980.

L. D. Fosdick.
Detecting Errors in Programs.
Performance Evaluation of Numerical Software, December 11-15, 1978, Baden, Austria, pages 77-87.

W. B. Foss.
A Structured Approach to Computer Systems Testing.
Canadian Datasystems (Canada), Vol.9(8):28-9, 31-2, September 1977.

M. S. Fugi.
Independent Verification of Highly Reliable Programs.
Proceedings of COMPSAC 77, pages 38-44, IEEE, 1977.

J. Gannon, P. McMullin, R. Hamlet, and M. Ardis.
Testing Traversable Stacks.
SIGPLAN Notices, Vol.15(1):58-65, January 1980.

G. R. Gladden.
Stop the Life Cycle, I Want to Get Off.
Software Engineering Notes, Vol. 7(10), 1982.

R. L. Glass.
Persistent Software Errors.
IEEE Transactions on Software Engineering, Vol.SE-7(2):162-8, March 1981.

A. L. Goel.
When to Stop Testing and Start Using Software?
Performance Evaluation Review, Vol.10(1):131-8, Spring 1981.

B. A. Goldman, R. S. Kilty, and J. J. Johnston.
Installation Testing with Software Systems.
I. Development of Test Software.
Western Electric Engineer, Vol.25(1):44-53, Winter 1981.

P. A. V. Hall.
In Defense of Life Cycles.
Software Engineering Notes, Vol.7(11), 1982.

T. G. Hallin and R. C. Hansen.
Toward a Better Method of Software Testing.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 153-7.

R. Hamlet.
Application of 'Dovetailing' to Program Testing.
SIGACT News, Vol.8(2):25-6, April-June 1976.

W. Harrison.
Testing Strategy (Programs).
Journal System Management, Vol.32(5):34-7, May 1981.

M. A. Hennell.
Management of Validation and Testing.
In Life-Cycle Management, State of the Art Report, pages 85-100, 1980.

M. A. Hennell, M. R. Woodward, and D. Hedley.
On Program Analysis.
Information Processing Letters (Netherlands), Vol.5(5):136-40, November 1976.

M. A. Hennell, M. R. Woodward, and D. Hedley.
Towards More Advanced Testing Techniques.
Workshop on Reliable Software, September 22-23, 1978, Bonn, Germany, pages 19-30.

W. Hock and K. Schittkowski.
Test Examples for Nonlinear Programming Codes.
Journal of Optimization Theory and Application, Vol.30(1):127-9, January 1980.

H. Holighaus.
Direct Production and Testing of Programs.
Elektronik, Vol.26(5):57-8, May 1977.

R. House.
Comments on Program Specification and Testing.
Communications of the ACM, Vol.23(6):324-9, June 1980.

W. E. Howden.
Applicability of Software Validation Techniques to Scientific Programs.
ACM Transactions in Programming Languages and Systems, Vol.2(3):307-20, July 1980.

W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions of Software Engineering, Vol.SE-2(3):208-14, September 1976.

K. Kirchhof and H. Neunert.
The Controlled Program Test.
Online-Adl-Nachr. (Germany), Vol.13(6):425-9, June 1975.

J. W. Laski.
A Hierarchical Approach to Program Testing.
SIGPLAN Notices, Vol.15(1):77-85, January 1980.

R. Loeser and E. M. Gaposchkin.
The Second Law of Debugging.
Software Practice and Experience, Vol.6(4):577-8, October-December 1976.

D. D. McCracken and M. A. Jackson.
Life Cycle Concept Considered Harmful.
Software Engineering Notes, Vol.7(10), 1982.

E. F. Miller, Jr.
Engineering Software for Testability.
10th IEEE Computer Society International Meeting on Computer Technology to Reach the People, (Digest of Papers), February 25-27, 1976, San Francisco, CA, pages 7-10.

E. F. Miller, Jr.
Program Testing - An Overview for Managers.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 114-19.

M. Morelli.
Software Engineering Testing.
Manage. and Inf. (Italy), No.5:357-9, May 1980.

M. Morelli.
Software Reliability - Principles and Criteria for Online Testing of the Software.
Manage. and Inf. (Italy), Vol.27(5):293-6, May 1978.

F. J. Mullin.
Software Test Management.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 321-6.

G. J. Myers.
A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.
Communications of the ACM, Vol.21(9):760-88, 1978.

M. A. Neighbors.
Assuring Software Reliability.
Computer Decisions, Vol.8(12):44-6, December 1976.

D. Novak.
Detection of Errors in Software.
Mech. Autom. Adm. (Czechoslovakia), Vol.16(10):389, 1976.

D. Novak.
Strategy of Finding Errors in Programs.
Mech. Autom. Adm. (Czechoslovakia), Vol.19(2):55-6, 1979.

L. M. Ottenstein.
Quantitative Estimates of Debugging Requirements.
IEEE Transactions on Software Engineering, Vol.SE-5(5):504-14, September 1979.

M. Paige.
Cost-Effective Software Test Methodologies.
Conference Record of the Fourteenth Asilomar Conference on Circuits, Systems and Computers, Pages 66-71, November 17-19, 1980, Pacific Grove, CA.

M. Paige.
Software Test Metrics.
Conference Record of the Thirteenth Asilomar Conference on Circuits, Systems and Computers, November 5-7, 1979, Pacific Grove, CA, pages 320-4.

M. R. Paige.
An Analytical Approach to Software Testing.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 527-32.

M. R. Paige.
Software Design for Testability.
Proceedings of the Eleventh Hawaii International Conference on System Sciences, January 5-6, 1978, Honolulu, Hawaii, pages 113-8.

M. R. Paige and M. A. Holthouse.
On Sizing Software Testing for Structured Program.
7th Annual International Conference on Fault-Tolerant Computing, IEEE, June 28-30, 1977, Los Angeles, CA, page 217.

D. J. Panzl.
Automatic Revision of Formal Test Procedures.
Proceedings of the 3rd International Conference on Software Engineering, May 10-12, 1978, Atlanta, GA, pages 320-6.

M. P. Perriens.
An Application of Formal Inspections to Top-Down Structured Program Development.
RADC-TR-77-212, IBM Federal Systems Division, Gaithersburg, MD, 1977, (NTIS AD/A-041645).

Program Writing and Testing.
Manage. and Inf. (Italy), No.7-8:433-8, July-August 1980.

A. Rathsack.
Fundamental Procedures for Program Testing.
Rechentech. Datenverarb. (Germany), Vol.10(12):37-9, December 1973.

J. Ronback.
Test Metrics for Software Quality.
Performance Evaluation Review, Vol.10(1):107, Spring 1981.

W. B. Samson.
Testing Overflow Algorithms for a Table of Variable Size.
Computer Journal (GB), Vol.19(1):92, February 1976.

L. L. Scharer.
Improving System Testing Techniques.
Datamation, Vol.23(9):115,117,120,124,128,132, September 1977.

N. F. Schneidewind.
Analysis of Error Processes in Computer Software.
Naval Postgraduate School, Monterey, CA, July 1974.

B. Shneiderman and D. McKay.
Experimental Investigations of Computer Program Debugging and Modification.
Proceedings of the 6th Congress of the International Ergonomics Association - Old World, New World, One World, and Technical Program of the 29th Annual Meeting of the Human Factors Society, July 11-16, 1976, College Park, MD, pages 557-63.

M. L. Shooman and M. I. Bolsky.
Types, Distribution, and Test Correction Times for Programming Errors.
SIGPLAN Notices, Vol.10(6):347-57, June 1975.

J. Simon.
A Comment on Do Traces.
SIGPLAN Notices, Vol.11(10):49-52, October 1976.

A. Smith.
Criteria for System Testing.
Computer Bulletin (GB), Ser.2(7):14-15, March 1976.

I. A. Smith.
Criteria for System Testing.
In Software Engineering Techniques, State of the Art Report, pages 319-25, 1977.

R. J. Smolenski.
Test Plan Development.
Journal of System Management, Vol.32(2):32-7, February 1981.

H. M. Sneed.
Systematic Program Testing is a Tedious Business.
Online-Adl-Nachr. (Germany), No.11:904-8, November 1978.

The Software Design and Testing.
Manage. and Inf. (Italy), No.7-8:433-8, July-August 1980.

S. L. Squires, M. Zelkowitz, and M. Branstad.
Rapid Prototyping Workshop: An Overview.
Software Engineering Notes, Vol.7(11), 1982.

D. Teichrow and E. A. Hershey, III.
PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering, Vol.SE-3(1):41-8, January 1977.

H. Trauboth.
Software Testing and Validation Techniques for Highly Reliable Process-Information Systems.
In Computer Audit and Control - State of the Art Report, pages 203-29, 1980.

Z. Varkonyi.
Problems of Program Quality Testing.
Inf. Elektron. (Hungary), Vol.12(5):258-63, 1977.

D. A. Walsh.
Structured Test Design for More Reliable Structured Cobol Programs.
Proceedings of the Online Conference on Pragmatic Programming and Sensible Software, February 1978, London, England, pages 471-89.

D. A. Walsh.
Structured Test Plans for Effective Product Test.
Software World (GB), Vol.9(2):2-8, 1978.

D. A. Walsh.
Structured Testing.
Datamation, Vol.23(7):111-4, 116-8, July 1977.

D. A. Walsh.
The Structured Test of Cobol Programs.
Sist. and Autom. (Italy), Vol.25(196):591-600, September 1979.

J. Warnke.
Diagnostic Functions and Test Programs.
Rechentech. Datenverarg. (Germany), Vol.18(2):28-31, February 1981.

A. I. Wasserman.
Testing and Verification Aspects of Pascal-Like Languages.
Computer Languages (GB), Vol.4(3-4):155-69, 1979.

J. Weinberg.
Attacking those Program Errors.
Datalink (GB), Vol.13, July 30, 1979.

A. Westley.
Software Testing - Planning Your Moves.
Data Processing (GB), Vol.21(5):12-4, May 1979.

T. J. Wheeler.
Embedded System Design with Ada as the System Design Language.
Journal of Systems and Software, Vol.2(1):11-22, February 1981.

B. H. Yin.
Software Design Testability Analysis.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 729-34.

P. M. Zislis.
Semantic Decomposition of Computer Programs: An Aid to Program Testing.
Acta Informatica (Germany), Vol.4(3):245-69, 1975.

2.2. TESTING TECHNIQUES

2.2.1. STATIC ANALYSIS TECHNIQUES

Books

M. S. Deutsch.
Software Verification and Validation Realistic Project Approaches.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

R. Dunn and R. Ullman.
Quality Assurance for Computer Software.
McGraw Hill Book Company, New York, 1982.

R. Glass.
Software Reliability Guidebook.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

Survey Articles

M. E. Fagan.
Design and Code Inspections to Reduce Errors in Program Development.
IBM Systems Journal, Vol.15(3):182-211, 1976.

R. E. Fairley.
Tutorial: Static Analysis and Dynamic Testing of Computer Software.
Computer, Vol.11(4):14-23, April 1978.

W. E. Howden and E. Miller.
A Survey of Static Analysis Methods.
Tutorial: Software Testing and Validation Techniques, IEEE, 1981, pages 101-15.

Detailed Articles

V. R. Basili and D. M. Weiss.
Evaluation of a Software Requirements Document by Analysis of Change Data.
Proceedings of the 5th International Conference on Software Engineering, March 1981, pages 314-324.

E. M. Boehm, R. K. McClean, and D. D. Urfrig.
Some Experience with Automated Aids to the Design of Large-Scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1:125-33, 1975.

S. H. Caine and E. K. Gordon.
PDL -- A Tool for Software Development.
National Computer Conference, AFIPS Proceedings, 1975.

T. E. Cheatham, Jr. and J. A. Townley.
Program Analysis Techniques for Software Reliability.
Workshop on Reliable Software, September 22-23, 1978, Bonn, Germany, pages 9-17.

E. B. Daily.
Software Development.
Proceedings of European Computing Review, Infotech International, Ltd., 1978.

M. E. Fagan.
Design and Code Inspections in the Development of Programs.
1975 International Symposium on Fault-Tolerant Computing. Digest of Papers, June 18-20, 1975, Paris, France, page 248.

M. E. Fagan.
Design and Code Inspections to Reduce Errors in Program Development.
IBM Systems Journal, Vol.15(3):182-211, 1976.

M. E. Fagan.
Inspecting Software Design and Code.
Datamation, Vol.23(10):133-44, October 1977.

M. S. Fugi.
Independent Verification of Highly Reliable Programs.
Proceedings of COMPSAC 77, pages 38-44, IEEE, 1977.

C. Gannon.
Error Detection Using Path Testing and Static Analysis.
Computer, Vol.12(8):26-32, August 1979.

F. J. Hill and B. Huey.
A Design Language Approach to Test Sequence Generation.
Computer, Vol.10(6):28-34, June 1977.

M. Ibramsha and V. Rajaraman.
Detection of Logical Errors in Decision Table Programs.
Communications of the ACM, Vol.21(12):1016-25, December 1978.

R. S. Lemos.
An Implementation of Structured Walk-Throughs in Teaching
Cobol Programming.
Communications of the ACM, Vol.22(6):335-40, June 1979.

G. J. Myers.
A Controlled Experiment in Program Testing and Code
Walkthroughs/Inspections.
Communications of the ACM, Vol.21(9):760-88, September 1978.

G. J. Myers.
Program Design Validation System.
IBM Technical Disclosure Bulletin, Vol.19(10):3806-8, March 1977.

L. J. Osterweil.
The Detection of Unexecutable Program Paths Through Static Data
Flow Analysis.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL,
pages 406-13.

M. P. Perriens.
An Application of Formal Inspections to Top-Down Structured
Program Development.
RADC-TR-77-212, IBM Federal Systems Division, Gaithersburg, MD,
1977, (NTIS AD/A-041645).

R. H. Perrott and A. K. Raja.
Quasiparallel Tracing.
Software Practice and Experience, Vol.7(4):483-92, July-August
1977.

S. Pimont and J. C. Rault.
A Software Reliability Assessment Based on a Structural and
Behavioral Analysis of Programs.
Proceedings of the 2nd International Conference on Software
Engineering, October 13-15, 1976, San Francisco, CA, pages
486-91.

S. K. Robinson and I. S. Torsun.
An Empirical Analysis of Fortran Programs.
Computer Journal (GB), Vol.19(1):56-62, February 1976.

R. N. Taylor and L. J. Osterweil.
Anomaly Detection in Concurrent Software by Static Data Flow
Analysis.
IEEE Transactions on Software Engineering, Vol.SE-6(3):278-285,
May 1980.

D. Teichrow and E. A. Hershey, III.
PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering, Vol.SE-3(1):41-8, January 1977.

D. Vitas.
On the Automatic Analysis of the Structure of Fortran Programs.
Informatica 78 XIII Yugoslav International Symposium on Information Processing, October 2-7, 1978, Bled, Yugoslavia.

I. K. Wendel and R. L. Kleir.
Fortran Error Detection through Static Analysis.
Software Engineering Notes, Vol.2(3):22-8, March 1977.

M. R. Woodward, M. A. Hennell, and D. Hedley.
A Measure of Control Flow Complexity in Program Text.
IEEE Transactions on Software Engineering, Vol.SE-5(1):45-50, January 1979.

2.2.2. SYMBOLIC TESTING

Survey Articles

T. E. Cheatham, Jr., G. H. Holloway, and J. A. Townley.
Symbolic Evaluation and the Analysis of Programs.
IEEE Transactions on Software Engineering, Vol.SE-5(4):402-17, July 1979.

J. A. Darringer and J. C. King.
Application of Symbolic Execution to Program Testing.
Computer, Vol.11(4):51-60, April 1978.

J. C. Huang.
An Approach to Program Testing.
ACM Computing Surveys, Vol.7(3):113-28, September 1975.

Detailed Articles

S. Bologna.
TEVERE-1: A Software System for Program Testing and Verification.
AICA 79 Conference, October 10-13, 1979, Bari, Italy, pages 71-8.

Y. V. Borzov.
Program Testing Using Symbolic Execution.
Programming and Computer Software, Vol.16:39-45, 1980.

R. S. Boyer, B. Elspas, and K. N. Levitt.
SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution.
SIGPLAN Notices, Vol.10(6):234-45, June 1975.

L. Clarke.
Test Data Generation and Symbolic Execution as an Aid in Software Validation.
Computer Science Conference, February 18-20, 1975, Washington, DC, page 41.

L. A. Clarke.
A System to Generate Test Data and Symbolically Execute Programs.
IEEE Transactions on Software Engineering, Vol.SE-2(3):215-22, September 1976.

J. A. Darringer.
The Use of Symbolic Execution in Program Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 67-85.
Infotech International, 1979.

C. Ghezzi and M. Jazayeri.
Syntax Directed Symbolic Execution.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 539-45.

S. L. Hantler.
The Relation Between Symbolic Program Testing and Program Verification.
Computer Science Conference, February 18-20, 1975, Washington, DC, page 41.

J. Horejs.
Finite Semantics - A Technique for Program Testing.
Proceedings of the 4th International Conference on Software Engineering, September 17-29, 1979, Munich, Germany, pages 433-40.

W. E. Howden.
An Evaluation of the Effectiveness of Symbolic Testing.
Software Practice and Experience, Vol.8(4):381-97, July - August 1978.

W. E. Howden.
Experiments with a Symbolic Evaluation System.
National Computer Conference, AFIPS Proceedings, June 1976, pages 899-908.

W. E. Howden.
Lindenmayer Grammars and Symbolic Testing.
Information Processing Letters (Netherlands), Vol.7(1):36-9, January 12, 1978.

W. E. Howden.
Reliability of Symbolic Evaluation.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 442-7.

W. E. Howden.
Symbolic Testing and the DISSECT Symbolic Evaluation System.
IEEE Transactions on Software Engineering, Vol.SE-3(4):276-78, July 1977.

W. E. Howden.
Symbolic Testing - Design Techniques, Costs and Effectiveness.
NBS Report GCR77-89, National Bureau of Standards, Springfield, VA, 1977, (NTIS PB268517).

J. C. Huang.
A Method for Program Analysis and Its Applications to Program-Correctness Problems.
International Journal of Computer Mathematics, Vol.5(3):203-27, June 1976.

J. C. King.
A New Approach to Program Testing.
SIGPLAN Notices, Vol.10(6):228-33, June 1975.

J. C. King.
Program Testing by Symbolic Execution.
Computer Science Conference, February 18-20, 1975, Washington, DC, page 41.

J. C. King.
Symbolic Execution and Program Testing.
Communications of the ACM, Vol.19(7):385-94, July 1976.

D. Matuszek.
The Case for the Assert Statement.
SIGPLAN Notices, Vol.11(8):36-7, August 1976.

R. W. Topor and R. M. Burstall.
Verification of Programs by Symbolic Execution - Progress Report.
Unpublished Report, Department of Machine Intelligence, University of Edinburg, Scotland, December 1972.

2.2.3. PROGRAM INSTRUMENTATION

Books

R. Glass.
Software Reliability Guidebook.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

Infotech State of the Art Report, Software Testing, Volume 1:
Analysis and Bibliography.
Infotech International, 1979.

G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

Survey Articles

J. B. Goodenough and S. L. Gerhart.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering, Vol.SE-1(2):156-73,
June 1975.

J. C. Huang.
Program Instrumentation.
In Infotech State of the Art Report, Software Testing,
Volume 1: Analysis and Bibliography, pages 144-50.
Infotech International, 1979.

J. C. Huang.
Program Instrumentation and Software Testing.
Computer, Vol.11(4):25-31, April 1978.

L. G. Stucki.
Tutorial on Program Testing Techniques.
Slide Masters, COMPSAC-77, November 8-11, 1977, Chicago, IL.

Detailed Articles

J. M. Adams.
Experiments on the Utility of Assertions for Debugging.
Proceedings Eleventh Hawaii International Conference on System
Science, Honolulu, HI, January 1978, pages 31-9.

D. M. Andrews.
Using Executable Assertions for Testing and Fault Tolerance.
Ninth Annual International Symposium on Fault-Tolerant
Computing, June 20-22, 1979, Madison, WI, pages 102-5.

D. M. Andrews and J. P. Benson.
Using Executable Assertions for Testing.
Conference Record of the Thirteenth Asilomar Conference on Circuits, Systems and Computers, November 5-7, 1979, Pacific Grove, CA, pages 302-5.

Automated Testing Analyzer for Cobol.
Software Technology Center, Science Applications, Inc.,
San Francisco, CA, April 1976.

T. L. Booth, R. Ammar, and R. Lenk.
An Instrumentation System to Measure User Performance in Interactive Systems.
Journal of Systems and Software, Vol.2(2):139-46, June 1981.

T. S. Chow.
A Generalized Assertion Language.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 392-9.

R. E. Fairley.
Tutorial: Static Analysis and Dynamic Testing of Computer Software.
Computer, Vol.11(4):14-23, April 1978.

J. C. Huang.
Detection of Data Flow Anomaly Through Program Instrumentation.
IEEE Transactions on Software Engineering, Vol.SE-5(3):226-36, May 1979.

J. C. Huang.
Instrumenting Programs for Symbolic-Trace Generation.
Computer, Vol.13(12):17-23, December 1980.

J. C. Huang.
Program Instrumentation: A Tool for Software Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 147-59. Infotech International, 1979.

M. A. Malik.
An Assertion Language for the Annotation of Program Modules.
Australian Computer Science Community (Australia), Vol.2(2):217-38, March 1980.

D. Matuszek.
The Case for the Assert Statement.
SIGPLAN Notices, Vol.11(8):36-7, August 1976.

R. L. Probert.
Optimal Insertion of Software Probes in Well-Delimited Programs.
IEEE Transactions on Software Engineering, Vol.SE-8(1):34-42,
January 1982.

C. V. Ramamoorthy and K. H. Kim.
Software Monitors Aiding Systematic Testing and Their Optimal Placement.
Proceedings of the 1st National Conference on Software Engineering, September 11-12, 1975, Washington, DC, pages 21-6.

C. V. Ramamoorthy, K. H. Kim, and W. T. Chen.
Optimal Placement of Software Monitors Aiding Systematic Testing.
IEEE Transactions on Software Engineering, Vol.SE-1(4):403-11,
December 1975.

S. H. Saib.
Executable Assertions - An Aid to Reliable Software.
Proceedings Eleventh Annual Asilomar Conference on Circuits, Systems, and Computers, November 1977, Pacific Grove, CA, pages 277-281.

L. G. Stucki.
New Directions in Automated Tools for Improving Software Quality.
In Current Trends in Programming Methodology, Volume II: Program Validation, R. T. Yeh, Editor, pages 80-111.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1977.

L. G. Stucki and G. L. Foshee.
New Assertion Concepts for Self-Metric Software Validation.
Proceedings of IEEE Conference on Reliable Software, April 1975, Los Angeles, CA, pages 59-65.

R. N. Taylor.
Assertions in Programming Languages.
SIGPLAN Notices, Vol.15(1):58-65, January 1980.

User's Manual, Fortran Automated Verification System (FAVS), Volume 1.
General Research Corporation, Santa Barbara, CA, January 1979.

2.2.4. PROGRAM MUTATION TESTING

Books

A. T. Acree.
On Mutation.
Ph.D. Thesis, Georgia Institute of Technology, 1980.

T. A. Budd.
Mutation Analysis of Program Test Data.
Ph.D. Thesis, Yale University, 1980.

R. A. DeMillo.
Program Mutation: An Approach to Software Testing.
Report GIT/ICS-83-03, Georgia Institute of Technology,
January 1983.

J. Gourlay.
Theory of Testing Computer Programs.
Ph.D. Thesis, University of Michigan, 1981.

Infotech State of the Art Report, Software Testing, Volume 1:
Analysis and Bibliography.
Infotech International, 1979.

Survey Articles

T. A. Budd.
Mutation Analysis: Ideas, Examples, Problems and Prospects.
In Computer Program Testing, B. Chandrasekaran and S. Radicchi,
Editors. North-Holland, 1981.

R. A. DeMillo.
Mutation Analysis as a Tool For Software Quality Assurance.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Hints on Test Data Selection: Help for the Practicing Programmer.
Computer, Vol.11(4):34-41, April 1978.

R. J. Lipton and F. G. Sayward.
The Status of Research on Program Mutation.
Digest for the Workshop on Software Testing and Test
Documentation, Ft. Lauderdale, FL, 1978, pages 355-73.

Detailed Articles

A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward.
Mutation Analysis.
Report GIT/ICS-79-08, Georgia Institute of Technology, 1979.

V. K. Agarwall and G. M. Masson.
Recursive Coverage Projection of Test Sets.
IEEE Transactions on Computers, Vol.C-28(11): 865-70, November 1979.

D. Baldwin and F. Sayward.
Heuristics for Determining Equivalence of Program Mutations.
Technical Report 161, Yale University, 1979.

M. Brooks.
Testing, Tracing, and Debugging Recursive Programs Having Simple Errors.
Department of Computer Science, Stanford University, 1980.

T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
The Design of a Prototype Mutation System for Program Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:623-7, 1978. Also reprinted in Tutorial: Automated Tools for Software Engineering, E. F. Miller, Editor, IEEE Computer Society, 1979.

T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Program.
7th ACM Symposium on Principles of Programming Languages, January 1980.

T. A. Budd and W. C. Miller.
Detecting Typographical Errors in Numerical Programs.
University of Arizona, Tuscon, AZ, 1982.

J. Burns.
Stability of Test Data from Program Mutation.
Digest for the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, 1978, pages 324-34.

R. A. DeMillo, D. Hocking, and M. J. Merritt.
A Comparison of Some Reliable Test Data Generation Procedures.
Report GIT/ICS-81-08, Georgia Institute of Technology, 1981.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Program Mutation: A New Approach to Program Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 107-26.
Infotech International, 1979.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Program Mutation as a Tool for Managing Large-Scale Software Development.
1978 ASQC Technical Conference Transactions, American Society for Quality Control Engineers, Chicago, 1978.

K. A. Foster.
Error Sensitive Test Cases Analysis (ESTCA).
IEEE Transactions on Software Engineering, Vol.SE-6(3):258-64, May 1980.

R. G. Hamlet.
Testing Programs with the Aid of a Compiler.
IEEE Transactions on Software Engineering, Vol.SE-3(4):279-90, July 1977.

J. M. Hanks.
Testing Cobol Programs by Mutation: Volume I - Introduction to the CMS.1 System, Volume II - CMS.1 System Documentation.
Report GIT/ICS-80-04, Georgia Institute of Technology, 1980.

W. E. Howden.
Weak Mutation Testing and Completeness of Test Sets.
IEEE Transactions on Software Engineering, Vol.SE-8(4):371-9, July 1982.

M. Morelli.
Software Engineering Testing.
Manage. and Inf. (Italy), No.5:357-9, May 1980.

S. C. Ntafos.
On Required Element Testing.
Proceedings of COMPSAC 81.

D. L. Ostapko and S-J. Hong.
Fault Analysis and Test Generation for Programmable Logic Arrays (PLA's).
IEEE Transactions on Computers, Vol.C-28(9):617-27, September 1979.

I. J. Riddle, J. A. Hennel, M. R. Woodward, and D. Hedley.
Practical Aspects of Program Mutation.
University of Nottingham, Nottingham, UK.

A. Tanaka.
Equivalence Testing for Fortran Mutation System Using Data Flow Analysis.
Department of Information and Computer Science, Georgia Institute of Technology, 1981.

E. J. Weyuker and T. J. Ostrand.
Theories of Program Testing and the Application of Revealing Subdomains.
IEEE Transactions on Software Engineering, Vol.SE-6(3):236-46, May 1980.

M. R. Woodward, M. A. Hennell, and D. Hedley.
A Limited Mutation Approach to Program Testing.
University of Nottingham, Nottingham, UK, 1980.

2.2.5. INPUT SPACE PARTITIONING

Detailed Articles

B. A. Carre.
Software Validation. I. Control Flow and Data Flow Analysis.
Advanced Techniques for Microprocessor Systems, pages 112-8, 1980.

J. C. Cherniavsky.
On Finding Test Data Sets for Loop Free Programs.
Information Processing Letters (Netherlands), Vol.8(2):106-7, February 15, 1979.

L. A. Clarke, J. Hassell, and D. J. Richardson.
A Close Look at Domain Testing.
IEEE Transactions on Software Engineering, Vol.SE-8(4):380-90, July 1982.

R. A. DeMillo, E. D. Hocking, and M. J. Merritt.
A Comparison of Some Reliable Test Data Generation Procedures.
Report GIT/ICS-81-08, Georgia Institute of Technology, April 1981.

H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil.
On Two Problems in the Generation of Program Test Paths.
IEEE Transactions on Software Engineering, Vol.SE-2(3):227-31, September 1976.

C. Gannon.
Error Detection Using Path Testing and Static Analysis.
Computer, Vol.12(8):26-32, August 1979.

W. E. Howden.
Applicability of Software Validation Techniques to Scientific Programs.
Transactions on Programming Languages and Systems, Vol.2(3):307-20, July 1980.

W. E. Howden.
Methodology for the Generation of Program Test Data.
IEEE Transactions on Computers, Vol.C-24(5):208-14, May 1975.

W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions on Software Engineering, Vol.SE-2(3):208-14, September 1976.

S. C. Ntafos and S. L. Hakimi.
On Path Cover Problems in Digraphs and Applications to Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-5(5):520-9, September 1979.

S. C. Ntafos and S. L. Hakimi.
On Structured Digraphs and Program Testing.
IEEE Transactions on Computers, Vol.C-30(1):67-71, January 1981.

D. J. Richardson and L. A. Clarke.
A Partition Analysis Method to Increase Program Reliability.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 244-53.

E. J. Weyuker and T. J. Ostrand.
Theories of Program Testing and the Application of Revealing Subdomains.
IEEE Transactions on Software Engineering, Vol.SE-6(3):236-46, May 1980.

L. J. White and E. I. Cohen.
A Domain Strategy for Computer Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-6(3):247-57, May 1980.

L. J. White, F. C. Teng, H. Kuo, and D. Coleman.
An Error Analysis of the Domain Testing Strategy.
Technical Report 78-2, Computer Information Science Research Center, Ohio State University, Columbus, September 1978.

G. R. Wilmot.
Wrong Branch, Wrong Store Program Error Detection.
IBM Technical Disclosure Bulletin, Vol.16(7):2122, December 1973.

M. R. Woodward, D. Hedley, and M. A. Hennell.
Experience with Path Analysis and Testing of Programs.
IEEE Transactions on Software Engineering, Vol.SE-6(3):278-85, May 1980.

S. J. Zeil and L. J. White.
Sufficient Test Sets for Path Analysis Testing Strategies.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 184-91.

2.2.6. FUNCTIONAL PROGRAM TESTING

Books

M. Jackson.
Principles of Program Design.
Academic, London, 1975.

E. Yourdan and L. L. Constantine.
Structured Design.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

Survey Articles

J. Goodenough and S. L. Gerhart.
Toward a Theory of Test Data Selection.
IEEE Transactions on Software Engineering, Vol.SE-1(2):156-73, June 1975.

W. E. Howden.
Functional Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-6(2):162-9, March 1980.

Detailed Articles

M. W. Alford.
A Requirements Engineering Methodology for Real-time Processing Requirements.
IEEE Transactions on Software Engineering, Vol.SE-3(1):60-8, January 1977.

W. E. Howden.
An Analysis of Software Validation Techniques for Scientific Programs.
Report No. DM-171-IR, Department of Mathematics, University of Victoria, March 1979.

W. E. Howden.
Applicability of Software Validation Techniques to Scientific Programs.
ACM Transactions in Programming Languages and Systems, Vol.2(3):307-20, July 1980.

W. E. Howden.
Completeness Criteria for Testing Elementary Program Functions.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 235-43.

W. E. Howden.
Functional Testing and Design Abstractions.
The Journal of Systems and Software, Vol.1(4):307-13, January 1980.

M. Karpovsky.
Testing for Numerical Computations.
IEEE Proceedings, Vol.127(2):69-76, March 1980.

G. E. Miller.
Functional Test Completeness.
New Electronics (GB), Vol.12(2):30,33-4, December 23, 1979.

J. J. More, B. S. Garbow, and K. E. Killstrom.
Testing Unconstrained Optimization Software.
ACM Transactions Mathematical Software, Vol.7(1):17-41, March 1981.

L. J. Osterweil and L. D. Fosdick.
Simulated Program Execution as a Strategy for Error Detection and Validation.
Proceedings of the 1976 Summer Computer Simulation Conference, July 12-14, 1976, Washington, DC, pages 704-7.

R. H. Wampler.
Problems Used in Testing the Efficiency and Accuracy of the Modified Gram-Schmidt Least Squares Algorithm.
National Bureau of Standards, Washington, DC, August 1980.

2.2.7. ALGEBRAIC PROGRAM TESTING

Books

B. F. Caviness.
On Canonical Forms and Simplification.
Ph.D. Thesis, Carnegie-Mellon University, 1968.

Detailed Articles

J. C. Cherniavsky.
On Finding Test Data Sets for Loop Free Programs.
Information Processing Letters (Netherlands), Vol.8(2):106-7, February 1979.

R. A. DeMillo and R. J. Lipton.
A Probabilistic Remark on Algebraic Program Testing.
Information Processing Letters (Netherlands), Vol.7(4):193-5, June 1978.

M. Geller.
Test Data as an Aid in Proving Program Correctness.
Proceedings of Second Symposium on Principles of Programming Languages, pages 209-18. ACM Publications, New York, 1976.

R. Hamlet.
Testing Programs with Finite Sets of Data.
Computer Journal, Vol.20(3):232-7, March 1977.

W. E. Howden.
Algebraic Program Testing.
Acta Informatica (Germany), Vol.10(1):53-66, 1978.

W. E. Howden.
Elementary Algebraic Program Testing Techniques.
Computer Science Technical Report 12, Applied Physics and Information Sciences, University of California, San Diego, CA, 1976.

Y. Isomoto and Y. Goto.
Test Method of a Computer Program for Eigenvalue Equations.
Information Processing Society of Japan (Joho Shori) (Japan), Vol.16(11):967-73, 1975.

J. H. Rowland and P. J. Davis.
On the Selection of Test Data for Recursive Mathematical Subroutines.
SIAM Journal Computers, Vol.10(1):59-72, February 1981.

J. H. Rowland and P. J. Davis.
On the Use of Transcendentals for Program Testing.
Journal of the Association for Computing Machinery,
Vol.28(1):181-90, January 1981.

2.2.8. RANDOM TESTING

Detailed Articles

J. M. Barzdin, J. J. Bicevskis, and A. A. Kalninsh.
Construction of Complete Sample System for Correctness Testing.
Mathematical Foundations of Computer Science, September 1-5, 1975, Marianske Lazne, Czechoslovakia, pages 1-12.

J. W. Duran and S. Ntafos.
A Report on Random Testing.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 179-83.

J. W. Duran and J. J. Wiorkowski.
Capture-Recapture Sampling for Estimating Software Error Content.
IEEE Transactions on Software Engineering, Vol.SE-7(1):147-8, January 1981.

J. W. Duran and J. J. Wiorkowski.
Quantifying Software Validity by Sampling.
IEEE Transactions on Reliability, Vol.R-29(2):141-4, June 1980.

E. H. Forman and N. D. Singpurwalla.
Optimal Time Intervals for Testing Hypotheses on Computer Software Errors.
IEEE Transactions on Reliability, Vol.R-28(3):250-3, August 1979.

S. F. Lundstrom.
Adaptive Random Data Generation for Computer Software Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:505-12, 1978.

P. B. Moranda.
Limits to Program Testing with Random Number Inputs.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 521-6.

P. B. Moranda.
Prediction of Software Reliability During Debugging.
Proceedings of the 1975 Annual Reliability and Maintainability Symposium, January 28-30, 1975, Washington, DC, pages 327-32.

2.2.9. GRAMMAR-BASED TESTING

Detailed Articles

J. A. Bauer and A. B. Finger.
Test Plan Generation Using Formal Grammars.
Proceedings of the 4th International Conference on Software Engineering, September 1979.

T. S. Chow.
Testing Software Design Modeled by Finite State Machines.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 58-64. Also printed in IEEE Transactions on Software Engineering, Vol.SE-4(3):178-86, May 1978.

A. M. Davis and W. J. Rataj.
Requirements Language Processing for the Effective Testing of Real-Time Systems.
Software Engineering Notes, Vol.3(5), November 1978.

A. M. Davis and T. G. Rauscher.
Formal Techniques and Automatic Processing to Ensure Correctness in Requirements Specifications.
Proceedings of the Specifications of Reliable Software Conference, April 3-5, 1979, Cambridge, MA.

A. G. Duncan and J. S. Hutchison.
Using Attribute Grammars to Test Designs and Implementations.
Proceedings of the 5th International Conference on Software Engineering, March 1981.

D. R. Milton and D. N. Fischer.
LL(k) Parsing for Attribute Grammars.
Proceedings of the 6th International Colloquium on Automata, Languages and Programming, July 1979.

A. J. Payne.
A Formalized Technique for Expressing Compiler Exercisers.
SIGPLAN Notices, Vol. 13(1), January 1978.

2.2.10. DATA-FLOW GUIDED TESTING

Survey Articles

L. D. Fosdick and L. J. Osterweil.
Data Flow Analysis in Software Reliability.
ACM Computer Surveys, Vol.8(3):305-330, September 1976.

W. E. Howden.
A Survey of Static Analysis Methods.
In Tutorial: Software Testing & Validation Techniques, E. Miller and W. E. Howden, Editors, pages 101-15.
IEEE, 1981.

Detailed Articles

F. E. Allen.
Interprocedural Data Flow Analysis.
Proceedings of the IFIP Congress 1974, pages 398-402, North Holland Publishers, Amsterdam, 1974.

F. E. Allen and J. Cocke.
A Program Data Flow Analysis Procedure.
Communications of the ACM, Vol.19(3):137-47, March 1976.

B. A. Carre.
Software Validation.
I. Control Flow and Data Flow Analysis.
Advanced Techniques for Microprocessor Systems, pages 112-8, 1980.

P. M. Herman.
A Data Flow Analysis Approach to Program Testing.
Australian Computer Journal, Vol.8(3):92-6, November 1976.

J. C. Huang.
Instrumenting Programs for Data Flow Analysis.
Technical Report TR-UH-CS-77-4, University of Houston, May 1977.

J. Laski.
On Data Flow Guided Program Testing.
SIGPLAN Notices, Vol.17(9), September 1982.

L. J. Osterweil.
The Detection of Unexecutable Program Paths Through Static Data Flow Analysis.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 406-13.

L. J. Osterweil and L. D. Fosdick.
Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection.
University of Colorado, Boulder, CO, September 1974.

M. Paige.
Data Space Testing.
Performance Evaluation Review, Vol.10(1):117-27, Spring 1981.

R. N. Taylor and L. J. Osterweil.
Anomaly Detection in Concurrent Software by Static Data Flow Analysis.
IEEE Transactions on Software Engineering, Vol.SE-6(3):265-76, May 1980.

2.2.11. COMPILER TESTING

Detailed Articles

P. W. Abrahams and L. A. Clarke.
Compile-Time Analysis of Data List-Format List Correspondences.
IEEE Transactions on Software Engineering, Vol.SE-5(6):612-7, November 1979.

F. Bazzichi and I. Spadafora.
An Automatic Generator for Compiler Testing.
IEEE Transactions on Software Engineering, Vol.SE-8(4):343-53, July 1982.

Federal Compiler Testing Center.
Report FCTC-81-40, Falls Church, VA.

Federal Compiler Testing Center.
Report FCTC-81-106, Falls Church, VA.

Federal Compiler Testing Center.
Report FCTC-81-112, Falls Church, VA.

Federal Compiler Testing Center.
Report FCTC-81-115, Falls Church, VA.

Federal Compiler Testing Center.
Report FCTC-81-118, Falls Church, VA.

Federal Compiler Testing Center.
Report FCTC-81-135, Falls Church, VA.

Federal Compiler Testing Center.
Report FCTC-81-146, Falls Church, VA.

W. H. Harrison.
Compiler Analysis of the Value Ranges for Variables.
IEEE Transactions on Software Engineering, Vol.SE-3(3):243-9, May 1977.

W. Miller and D. L. Spooner.
Automatic Generation of Floating-Point Test Data.
IEEE Transactions on Software Engineering, Vol.SE-2(3):223-6, September 1976.

A. J. Payne.
A Formalized Technique for Expressing Compiler Exercisers.
SIGPLAN Notices, Vol.13(1), 1978.

P. A. Pravil-Shchikov and V. S. Shchepin.
Compilation of Structural Programs in a Dialogue Mode with Concurrent Test Generation.
Avtom. and Telemekh. (USSR), Vol.40(8):129-38, August 1979.

H. Samet.
A Machine Description Facility for Compiler Testing.
IEEE Transactions on Software Engineering, Vol. SE-3(5):343-51, September 1977.

R. S. Scowen.
Testing the Diagnostic Features of the Babel Compiler.
National Physics Laboratory, Teddington, England, September 1974.

R. P. Seaman.
Testing Compiler Operations.
IBM Technical Disclosure Bulletin, Vol.17(11):3345, April 1975.

A. I. Wasserman.
Some Testing and Verification Issues in the Design of Pascal-Like Languages.
3rd USA-Japan Computer Conference Proceedings, October 10-12, 1978, San Francisco, CA, pages 280-5.

B. A. Wichmann and B. Jones.
Testing Algol 60 Compilers.
Software Practice and Experience, Vol.6(2):261-70, April-June 1976.

2.2.12. REAL-TIME SOFTWARE AND TESTING

Survey Articles

W. Geiger, L. Gmeiner, H. Trauboth, and U. Voges.
Program Testing Techniques for Nuclear Reactor Protection Systems.
Computer, Vol.12(8):10-8, August 1979.

R. L. Glass.
Real-Time: The "Lost World" of Software Debugging and Testing.
Communications of the ACM, Vol.23(5):264-71, May 1980.

J. Ludewig.
Computer-Aided Specification of Process Control Systems.
Computer, Vol.15(5):12-20, May 1982.

Detailed Articles

U. Agosti, V. Giannini, and O. Murro.
A Study of the Integrated Environment for the Testing and Debugging of Interpretative Language.
Annual Conference AICA, October 29-31, 1980, Bologna, Italy, pages 841-50.

M. W. Alford.
A Requirements Engineering Methodology for Real-time Processing Requirements.
IEEE Transactions on Software Engineering, Vol.SE-3(1):60-8, January 1977.

J. J. Bailey, M. Horton, and S. B. Itscoitz.
The Importance of Reproducibility Testing of Computer Programs for Electrocardiographic Interpretation: Application to the Automatic Vectorcardiographic Analysis Program (AVA 3.4).
Computers and Biomedical Research, Vol.9(4):307-16, August 1976.

K. Ban, T. Fujima, H. Ota, S. Iikawa, and A. Takubo.
Development and Testing of Software for KDD's Automex System.
Mitsubishi Electrical Engineering (Japan), No.39:1-7, March 1974.

V. R. Basili and R. E. Noonan.
A Testing Tool for a Fire-Control Environment.
Proceedings of COMPCON 76, September 7-10, 1976, Washington, DC, pages 341-5.

J. D. Baum and J. M. Baca.
Real-Time Event Trace Monitor for Embedded Computer Systems.
Proceedings of the IEEE 1979 National Aerospace and Electronics Conference, May 15-17, 1979, Dayton, OH, pages 833-9.

F. Beinvogl and H. Siebert.
Switching System Software Test Methods, Aids and Procedures.
Proceedings of the Third International Conference on Software Engineering for Telecommunication Switching Systems, June 27-29, 1978, Helsinki, Finland, pages 91-6.

J. P. Benson and R. A. Melton.
A Laboratory for the Development and Evaluation of BMD Software Quality Enhancements Techniques.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 106-9.

W. E. Boebert, J. M. Kamrad, and E. R. Rang.
The Analytic Verification of Flight Software - A Case Study.
Proceedings of the IEEE 1978 National Aerospace and Electronics Conference, May 16-18, 1978, Dayton, OH, pages 242-8.

E. M. Boehm, R. K. McClean, and D. D. Urfrig.
Some Experience with Automated Aids to the Design of Large-Scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1:125-33, 1975.

D. M. Bradley and C. P. Miller.
The Implementation, Testing and Use of CASPA.
Third International Conference and Exhibition on Computers in Engineering and Building Design, March 14-16, 1978, Brighton, England, pages 351-63.

B. Brehme and J. Rosenpflanzer.
Rationalization of Program Tests in the Application of EDP of the ESER.
Rechentech. Datenverarb. (Germany), Vol.11(1):8-11, January 1974.

P. Burnett, P. A. Kidd, and A. M. Lister.
Simulation of Real-Time Program Faults.
Computer Journal (GB), Vol.17(1):25-7, February 1974.

A. R. Chandler.
Software Verification and Validation for Command and Control Systems.
RCA Engineer, Vol.19(5):32-5, Feburary-March, 1974.

T. S. Chow.
Integration Testing of Distributed Software.
Proceedings of COMPCON 80, September 23-25, 1980, Washington, DC, pages 706-10.

J. F. Clemons.
Verification of the Onboard Flight Software Developed for the NASA Space Shuttle Program.
Proceedings of the Eighth Texas Conference on Computing Systems, 1979.

R. M. Cohen.
Formal Specifications for Real-time Systems.
Proceedings of the Seventh Texas Conference on Computing Systems, October 1978.

B. P. Cosell, J. M. McQuillan, and D. C. Walden.
Techniques for Detecting and Preventing Multiprogramming Bugs.
Proceedings of the IFIP Conference on Software for Minicomputers, September 8-12, 1975, Keszthely, Hungary, pages 301-8.

C. G. Davis.
The Testing of Large, Real Time Software Systems.
Proceedings of the Seventh Texas Conference on Computing Sytems, October 30 - November 1, 1978, Houston, TX.

W. E. Ehrenberger and K. P. Plogert.
Nuclear Power Plant Control and Instrumentation, April 24-28, 1978, Cannes, France, pages 543-63.

W. Ehrenberger, G. Rauch, and K. Okroy.
Program Analysis - A Method for the Verification of Software for the Control of a Nuclear Reactor.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 611-6.

F. Erler.
Testing of Interactive Programs for the ES 1020 Computer.
Wiss. Z. Technical Hochsch. Karl-Marx-Stadt (Germany), Vol.17(6):699-706, 1975.

R. E. Fairley.
Dynamic Testing of Simulation Software.
Proceedings of the 1976 Summer Computer Simulation Conference, July 12-14, 1976, Washington, DC, pages 708-10.

H. E. Frye and R. L. Hoffman.
Program Event Monitoring Instruction.
IBM Technical Disclosure Bulletin, Vol.18(5):1557-9, October 1975.

R. E. Fryer.
The User Interface for a Real-Time Software Debugging System.
Conference Record of the Fourteenth Asilomar Conference on Circuits, Systems and Computers, November 17-19, 1980, Pacific Grove, CA, pages 462-9.

L. F. Giaccone and L. J. Woodrum.
Test and Debug of Device/Process Control Code.
IBM Technical Disclosure Bulletin, Vol.19(2):609-11, July 1976.

C. Hanisch.
Experience with a Computer Operating System Test Environment.
Rechentech. Datenverarb. (Germany), Vol.17(5):11-3, May 1980.

P. B. Hansen.
Reproducible Testing of Monitors.
Software Practice and Experience, Vol.8(6):721-9, November - December 1978.

R. V. Head.
Testing Real-time Systems. Part 1: Development and Management.
Datamation, page 42, July 1964.

M. J. R. Healey.
Generating Test Cases for Percent Include Processors.
IBM Technical Disclosure Bulletin, Vol.16(5):1665-6, October 1973.

C. E. Hughes and C. P. Pfleeger.
Assist-V - An Environment Simulator for IBM 360 Systems Software Development.
IEEE Transactions on Software Engineering, Vol.SE-4(6):526-30, November 1978.

E. Husu.
PROSIT - A Tool for the Testing of Proteo Central Control Software (Electronic Switching Systems).
Telecomunicazioni (Italy), No.71-72:63-8, July-October 1979.

K. Joudu.
Reliability of Real Time Computer Software.
1st IFAC/IFIP Symposium on Software for Computer Control, May 25-28, 1976, Tallina, USSR, pages 105-8.

J. R. Kane and S. S. Yau.
Concurrent Software Fault Detection.
IEEE Transactions on Software Engineering, Vol.SE-1(1):87-99, March 1975.

R. Katz.
Analysis of the AWACS Passive Tracking Algorithms on the RADCAP STARAN.
Proceedings of the 1976 International Conference on Parallel Processing, August 24-27, 1976, Walden Woods, MI, pages 177-86.

P. Knezevic.
Programs for Correctness Testing of Other Programs in a Real Time Processor System.
Informatica 78 XIII Yugoslav International Symposium on Information Processing, October 2-7, 1978, Bled, Yugoslavia.

H. Kopetz.
Systematic Error Treatment in Real Time Software.
Proceedings of the 6th Triennial World Congress of the International Federation of Automatic Control, August 24-30, 1975, Boston and Cambridge, MA.

E. Krieger.
Online Test System and Hardware Experiences on the R40 ES Computer OS/ES Operating System.
Inf. Elektron. (Hungary), Vol.15(3):150-3, 1980.

J. Larson.
Automatic Error Analysis for Serial and Parallel Algorithms.
High Speed Computer and Algorithm Organization, April 13-15, 1977, Champaign, IL, pages 457-9.

R. Martin and G. Memmi.
Specification and Validation of Sequential Processes Communicating by FIFO Channels.
IEEE Fourth Internation Conference on Software Engineering for Telecommunication Switching Systems, July 20-24, 1981, Coventry, England, pages 54-7.

S. C. McSweeney and A. J. Bass.
A Realization of Automated ECS Software Test.
Proceedings of the IEEE 1979 National Aerospace and Electronics Conference, May 15-17, 1979, Dayton, OH, pages 107-12.

I. Miyamoto.
Software Reliability in Online Real Time Environment.
SIGPLAN Notices, Vol.10(6):194-203, June 1975.

M. J. Norton.
Experience in Software Test Techniques for Packet Switching Exchanges.
Proceedings of the Third International Conference on Software Engineering for Telecommunication Switching Systems, June 27-29, 1978, Helsinki, Finland, pages 149-55.

W. R. Odgen.
Independent Feedback Method for Program Field Test.
IBM Technical Disclosure Bulletin, Vol.23(8):3824-5, January 1981.

J. A. Painter.
Software Testing in Support of Worldwide Military Command and Control System ADP.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 316-20.

B. A. Pozin.
A Method of Test Structuring for Debugging Control Programs.
Programmirovanie (USSR), Vol.6(2):62-9, March-April 1980.

C. M. Rader and S. L. Scharf.
Do Not Use a Chirp to Test a DFT Program.
IEEE Transactions on Acoustics, Speech and Signal Processes, Vol.ASSP-27(4):430-2, August 1979.

P. Schluchtmann.
Efficient Program Testing on Robotron 4200/4201.
Rechentech. Datenverarb. (Germany), Vol.16(3):32-3, March 1979.

K. Soos, A. Szeplaki, and Z. Varkonyi.
Program Testing on R-10 Computer with Stem.
Inf. Elektron (Hungary), Vol.12(1):39-47, 1977.

D. E. Storey.
The Testing of Interactive Micro-Computer Software on a Host Computer.
Trend in On-Line Computer Control Systems, April 21-24, 1975, Sheffield, Yorkshire, England, pages 206-11.

R. N. Taylor and L. J. Osterweil.
Anomaly Detection in Concurrent Software by Static Data Flow Analysis.
IEEE Transactions on Software Engineering, Vol.SE-6(3):278-285, May 1980.

U. Voges.
Aspects of Design, Test and Validation of the Software for a Computerized Reactor Protection System.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 606-10.

M. L. Watkins.
A Technique for Testing Command and Control Software.
Communications of the ACM, Vol.25(4):228-32, April 1982.

C. D. Williams and C. Nemec.
Verification of Operational Flight Programs by Simulation.
Proceedings of the IEEE 1979 National Aerospace and Electronics Conference, May 15-17, 1979, Dayton, OH, pages 826-8.

J. J. Zelasco, Jr.
An Interactive Debug and Test Program for Avionic Software.
Proceedings of the IEEE 1976 National Aerospace and Electronics Conference, May 18-20, 1976, Dayton, Ohio, page 911.

2.3. OTHER STRATEGIES FOR CONSTRUCTING RELIABLE SOFTWARE

Books

M. S. Deutsch.
Software Verification and Validation Realistic Project Approaches.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

T. Gilb.
Software Metrics.
Winthrop Publishers, Inc., Cambridge, MA, 1977.

R. L. Glass.
Modern Programming Practices, A Report from Industry.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1982.

G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

E. Yourdan and L. L. Constantine.
Structured Design.
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1979.

Survey Articles

R. P. Abbott.
Towards the Audit of Computer Software.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 1-12.
Infotech International, 1979.

W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

E. L. Battiste.
Reliable-Warrantable Code.
Performance Evaluation of Numerical Software, December 11-15, 1978, Baden, Austria, pages 151-7.

L. A. Belady.
On Software Complexity.
Proceedings of the IEEE Workshop on Quantitative Software Models, pages 90-94. IEEE, Piscataway, NJ, 1979.

G. D. Bergland.
A Guided Tour of Program Design Methodologies.
Computer, Vol.14(10):12-37, October 1981.

J. B. Bowen.
A Survey of Standards and Proposed Metrics for Software Quality Testing.
Computer, Vol.12(8):37-42, August 1979.

W. L. Bryan, S. G. Siegel, and G. L. Whiteleather.
Auditing Throughout the Software Life Cycle: A Primer.
Computer, Vol.15(3):57-67, March 1982.

B. Curtis.
Measurement and Experimentation in Software Engineering.
Proceedings of the IEEE, Vol.68(9):1144-57, September 1980.

T. C. Jones.
A Survey of Programming Design and Specification Techniques.
IEEE Catalog 79 CH1401, April 1979.

D. Levy, D. Guy, and J. Ronback.
A Place for Metrics in Software Development.
Telesis (Canada), Vol. 6(5):17-22, October 1979.

C. C. Liu.
A Look at Software Maintenance.
Datamation, Vol.22(11):51-5, November 1976.

D. Markham, J. McCall, and G. Walters.
Software Metrics Application Techniques.
Proceedings of Trends and Applications 1981. Advances in Software Technology, May 28, 1981, Gaithersburg, MD, pages 38-46.

C. V. Ramamoorthy and H. H. Ho.
A Survey of Principles and Techniques of Software Requirements and Specifications.
In Software Engineering Techniques, Volume 2 -- Invited Papers, pages 265-318.
Infotech International, 1977.

D. T. Ross.
Reflections on Requirements.
IEEE Transactions on Software Engineering, Vol.SE-3(1):2-5, January 1977.

B. A. Silverberg.
An Overview of the SRI Hierarchical Development Methodology.
Proceedings of the Symposium on Software Engineering Environments, June 16-20, 1980, Lahnstein, Germany, pages 235-42.

A. R. Sorkowitz.
Certification Testing: A Procedure to Improve the Quality of Software.
Computer, Vol.12(8):20-5, August 1979.

L. G. Stucki, et al.
Methodology for Producing Reliable Software.
NASA CR 144769 (two volumes), McDonnel-Douglas Astronautics Company, March 1976.

Detailed Articles

M. W. Alford.
A Requirements Engineering Methodology for Real-time Processing Requirements.
IEEE Transactions on Software Engineering, Vol.SE-3(1):60-8, January 1977.

C. T. Bailey and W. L. Dingee.
A Software Study Using Halstead Metrics.
Performance Evaluation Review, Vol.10(1):189-97, Spring 1981.

E. E. Balkovich and G. P Engelberg.
Research Towards a Technology to Support the Specification of Data Processing System Performance Requirements.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 110-5.

P. Barker.
How A Bubble-Sort Can Test Code Efficiency.
Practical Computing (GB), Vol.4(7):125-31, July 1981.

V. R. Basili and D. M. Weiss.
Evaluation of a Software Requirements Document by Analysis of Change Data.
Proceedings of the 5th International Conference on Software Engineering, March 1981, pages 314-324.

T. E. Bell, D. C. Bixler, and M. E. Dyer.
An Extendable Approach to Computer-Aided Software Requirements Engineering.
IEEE Transactions on Software Engineering, Vol.SE-3(1):49-59, January 1977.

J. P. Benson.
Adaptive Search Techniques Applied to Software Testing.
Performance Evaluation Review, Vol.10(1):109-16, Spring 1981.

J. P. Bielski and W. H. Blankertz.
The General Acceptance Test System (GATS).
Proceedings of COMPCON 77, February 28 - March 3, 1977, San Francisco, CA, pages 207-10.

B. W. Boehm.
Software Engineering.
IEEE Transactions on Computers, Vol.C-25(12), December 1976.

E. M. Boehm, R. K. McClean, and D. D. Urfrig.
Some Experience with Automated Aids to the Design of Large-Scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1(1):125-33, March 1975.

V. Breitfeld.
New Methods and Techniques of Programming.
V. Dialogue Program Testing for Cobol Users with Cobol Interactive Debug.
IBM Nachr. (Germany), Vol.25(226):207-13, July 1975.

F. Buckley.
A Standard for Software Quality Assurance Plans.
Computer, Vol.12(8):43-51, August 1979.

S. H. Caine and E. K. Gordon.
PDL -- A Tool for Software Development.
National Computer Conference, AFIPS Proceedings, 1975.

C. S. Chandersekaran and R. C. Linger.
Software Specification Using the SPECIAL Language.
Journal of Systems and Software, Vol.2(1):31-8, February 1981.

F. Cristian.
Exception Handling and Software-Fault Tolerance.
10th International Symposium on Fault-Tolerant Computing, October 1-3, 1980, Kyoto, Japan, pages 97-103.

A. M. Davis.
The Design of a Family of Application-Oriented Requirements Languages.
Computer, Vol.15(5):21-28, May 1982.

J. L. Elshoff.
Measuring Commercial PL/1 Programs Using Halstead's Criteria.
SIGPLAN Notices, Vol.11(5):38-46, May 1976.

E. Fassbinder.
New Methods and Programming Techniques.
III. Fortran Interactive Debug - A New Test Facility.
IBM Nachr. (Germany), Vol.25(224):62-6, February 1975.

K. F. Fischer.
A Test Case Selection Method for the Validation of Software Maintenance Modifications.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 421-6.

A. Fitzsimmons and T. Love.
A Review and Evaluation of Software Science.
ACM Computing Surveys, Vol.10(1):3-18, March, 1978.

A. B. Fitzsimmons.
Relating the Presence of Software Errors to the Theory of Software Science.
Proceedings of the Eleventh Hawaii International Conference on System Sciences, January 5-6, 1978, Honolulu, HI, pages 40-6.

M. S. Fuji.
Independent Verification of Highly Reliable Programs.
Proceedings of COMPCON 77, November 8-11, 1977, Chicago, IL, pages 38-44.

G. R. Gladden.
Stop the Life Cycle, I Want to Get Off.
Software Engineering Notes, Vol. 7(10), 1982.

A. Grnarov, J. Arlat and A. Avizienis.
On the Performance of Software Fault-Tolerance Strategies.
10th International Symposium on Fault-Tolerant Computing, October 1-3, 1980, Kyoto, Japan, pages 25-13.

P. A. V. Hall.
In Defense of Life Cycles.
Software Engineering Notes, Vol.7(11), 1982.

K. L. Heninger.
Specifying Software Requirements for Complex Systems: New Techniques and Their Applications.
IEEE Transactions on Software Engineering, Vol.SE-6(1):2-12, January 1982.

S. Henry and D. Kafura.
Software Structure Metrics Based on Information Flow.
IEEE Transactions on Software Engineering, Vol.SE-7(5):510-8, September 1981.

M. A. Herndon.
Cost Effectiveness in Software Error Analysis Systems.
Second Software Life Cycle Management Workshop, August 21-22, 1978, Atlanta, GA, pages 180-1.

F. J. Hill and B. Huey.
A Design Language Approach to Test Sequence Generation.
Computer, Vol.10(6):28-34, June 1977.

H. Holighaus.
Direct Production and Testing of Programs.
Elektronik, Vol.26(5):57-8, May 1977.

T. C. Jones.
Measuring Programming Quality and Productivity.
IBM Systems Journal, Vol.17(1):39-63, 1978.

R. E. Keirstead.
On Software Certification.
Proceedings of COMPCON 76, February 24-26, 1976, San Francisco, CA, pages 222-4.

R. Lemiere.
The Control of Quality - A Concept to be Practiced During Development.
Choisir Son Informatique, pages 79-80, September 1979, Paris, France.

A. A. Levene and G. P. Mullery.
An Investigation of Requirement Specification Languages: Theory and Practice.
Computer, Vol.15(5):50-9, May 1982.

C. R. Litecky and G. B. Davis.
A Study of Errors, Error-Proneness, and Error Diagnosis in Cobol.
Communications of the ACM, Vol.19(1):33-7, January 1976.

T. E. Matysek.
HOL in Operational Software - From a User's Point of View.
Proceedings of the IEEE 1977 National Aerospace and Electronics Conference, May 17-19, 1977, Dayton, OH, pages 494-501.

T. J. McCabe.
A Complexity Measure.
IEEE Transactions on Software Engineering SE-2(4):308-320, December 1976.

D. D. McCracken and M. A. Jackson.
Life Cycle Concept Considered Harmful.
Software Engineering Notes, Vol.7(10), 1982.

E. F. Miller, Jr.
Engineering Software for Testability.
10th IEEE Computer Society International Meeting on Computer Technology to Reach the People, (Digest of Papers), February 25-27, 1976, San Francisco, CA, pages 7-10.

G. J. Myers.
Program Design Validation System.
IBM Technical Disclosure Bulletin, Vol.19(10):3806-8, March 1977.

J. D. Naumann, G. B. Davis, and J. D. McKeen.
Determining Information Requirements: A Contingency Method for Selection of a Requirements Assurance Strategy.
Journal of Systems and Software, Vol.1(4):273-82, 1980.

L. Ottenstein.
Predicting Numbers of Errors Using Software Science.
Performance Evaluation Review, Vol.10(1):157-67, Spring 1981.

M. R. Paige.
Software Design for Testability.
Proceedings of the Eleventh Hawaii International Conference on System Sciences, January 5-6, 1978, Honolulu, HI, pages 113-8.

B. Parhami.
The Concept of Self-Checking Programs.
7th Annual International Conference on Fault-Tolerant Computing, IEEE, June 28-30, 1977, Los Angeles, CA, page 216.

R. R. Prudhomme.
Software Verification and Validation and SQA.
American Society for Quality Control 34th Annual Technical Conference Transactions, May 20-22, 1980, Atlanta, GA, pages 397-404.

N. S. Prywes, A. Pnueli, and A. Shastry.
Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development.
Transactions on Programming Languages and Systems, Vol.1(2):196-217, October 1979.

J. P. Renault.
Towards A Comprehensive and Homogenous Software Development Methodology.
1979 International Conference on Communications, June 10-14, 1979, Boston, MA.

D. T. Ross and K. E. Schoman.
Structured Analysis for Requirements Definition.
IEEE Transactions on Software Engineering, Vol.SE-3(1):6-15, January 1977.

K. Sakata.
Formulation of Predictive Methods in Software Production Control-Dynamic Prediction: Quality Probe.
Systems Computer Control, Vol.5(3):27-34, 1974.

N. F. Schneidewind.
The Use of Simulation in the Evaluation of Software.
Computer, Vol.10(4):47-53, April 1977.

N. F. Schneidewind and H. M. Hoffmann.
An Experiment in Software Error Data Collection and Analysis.
IEEE Transactions on Software Engineering, Vol.SE-5(3):276-86, May 1979.

N. F. Schneidewind and H. M. Hoffman.
Software Error Data Collection and Analysis.
Proceedings of the 1978 Summer Computer Simulation Conference, July 24-26, 1978, Los Angeles, CA, pages 748-53.

D. N. Shorter.
Requirements for Reliable Software.
IEEE Colloquium on the Engineering of Industrial Microprocessor-Based Systems - The Users Point of View, March 5, 1981, London, England.

R. J. Smolenski.
Test Plan Development.
Journal of System Management, Vol.32(2):32-7, February 1981.

I. M. Soi and K. Gopal.
Error Prediction in Software.
Microelectronics and Reliability (GB), No.1-2:39-47, January-February 1979.

S. L. Squires, M. Zelkowitz, and M. Branstad.
Rapid Prototyping Workshop: An Overview.
Software Engineering Notes, Vol.7(11), 1982.

L. G. Stucki.
Automatic Generation of Self-Metric Software.
Proceedings IEEE Symposium on Computer Software Reliability, New York, 1973, pages 94-100.

D. Teichrow and E. A. Hershey, III.
PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering, Vol.SE-3(1):41-8, January 1977.

K. Terplan.
Measurements for Improving Reliability.
Computer Science Conference, February 18-20, 1975, Washington, DC, page 30.

L. L. Walker.
Acceptance Testing for Software.
Software World (GB), Vol.4(9):11-5, December 1973.

T. J. Wheeler.
Embedded System Design with Ada as the System Design Language.
Journal of Systems and Software, Vol.2(1):11-22, February 1981.

D. E. Wright and B. D. Carroll.
An Automated Data Collection System for the Study of Software Reliability.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 571-6.

B. H. Yin.
Software Design Testability Analysis.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 729-34.

P. Zave.
An Operational Approach to Requirements Specification for Embedded Systems.
IEEE Transactions on Software Engineering, Vol.SE-8(3):250-69, May 1982.

2.4. COMPARATIVE EVALUATION OF TESTING TECHNIQUES

Books

T. A. Budd.
Mutation Analysis of Program Test Data.
Ph.D. Thesis, Yale University, 1980.

Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography.
Infotech International, 1979.

Survey Articles

W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

J. M. Bilton.
A Survey of Self-Test and Bite Program Generation.
Euromicro Journal (Netherlands), Vol.6(3):168-74, May 1980.

W. E. Howden.
Empirical Studies of Software Validation.
Microelectronics and Reliability (GB), No.1-2:39-47, January-February 1979.

Detailed Articles

A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward.
Mutation Analysis.
Report GIT/ICS-79-08, Georgia Institute of Technology, 1979.

R. A. DeMillo, D. Hocking, and M. J. Merritt.
A Comparison of Some Reliable Test Data Generation Procedures.
Report GIT/ICS-81-08, Georgia Institute of Technology, April 1981.

A. B. Fitzsimmons.
Relating the Presence of Software Errors to the Theory of Software Science.
Proceedings of the Eleventh Hawaii International Conference on System Sciences, January 5-6, 1978, Honolulu, HI, pages 40-6.

C. Gannon.
Error Detection Using Path Testing and Static Analysis.
Computer, Vol.12(8):26-32, August 1979.

W. E. Howden.
Reliability of Symbolic Evaluation.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 442-7.

W. E. Howden.
Reliability of the Path Analysis Testing Strategy.
IEEE Transactions on Software Engineering, Vol.SE-2(3):208-14, September 1976.

W. E. Howden.
Theoretical and Empirical Studies of Program Testing.
IEEE Transactions on Software Engineering, Vol.SE-4(4):293-7, July 1978.

G. J. Myers.
A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.
Communications of the ACM, Vol.21(9):760-8, September 1978.

R. Thibodeu.
GRC Report to AIRMICS, 1981.

L. J. White, F. C. Teng, H. Kuo, and D. Coleman.
An Error Analysis of the Domain Testing Strategy.
Technical Report 78-2, Computer Information Science Research Center, Ohio State University, Columbus, September 1978.

M. R. Woodward, D. Hedley, and M. A. Hennell.
Experience with Path Analysis and Testing of Programs.
IEEE Transactions on Software Engineering, Vol.SE-6(3):278-86, May 1980.

3. TESTING AND EVALUATION TOOLS

3.1. GENERAL

Books

Software Engineering Automated Tools Index.
Software Research Associates, P. O. Box 2432, San Francisco, CA 94126.

Survey Articles

T. Budd, M. Majoras, and H. Sneed.
Experiences with a Software Test Factory.
Proceedings of Spring COMPCON 79, February 26 - March 1, 1979, San Francisco, CA, pages 319-29.

B. Chandrasasekaran.
Test Tools: Usefulness Must Extend to Everyday Programming Environment.
Computer, Vol.12(3):102-3, March 1979.

J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY, February 1980.

H. R. Downs.
Automated Tools for the Verification of Computer Programs.
Transactions of the American Nuclear Society and the European Nuclear Society, 1980 International Conference, November 16-21, 1980, Washington, DC, pages 253-4.

L. Gmeiner and U. Voges.
Methods, Criteria and Automatic Tools for Software Testing.
Practice in Software Adaption and Maintenance. Proceedings for the SAM Workshop, April 5-6, 1979, Berlin, Germany, pages 183-92.

H. Hecht.
Final Report: A Survey of Software Tools Usage.
NBS Special Publication 500-82, National Bureau of Standards, November 1981.

E. F. Miller, Jr.
Program Testing Tools and Their Use.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 184-215, Infotech International, 1979.

I. Miyamoto.
Automated Testing-Aid Tools Survey.
Information Processing Society of Japan (Joho Shori) (Japan), Vol.20(8):688-93, August 1979.

D. J. Reifer and R. L. Ettenger.
Test Tools: Are They A Cure-All?
Proceedings of the 1975 Annual Reliability and Maintainability Symposium, January 28-30, 1975, Washington, DC, pages 492-7.

D. J. Reifer and H. A. Montgomery.
SEATECS Software Tools Survey.
RCI-TR-008 (Compiled for the NOSC SEATECS Project), Reifer Consultants, Inc., March 1981.

D. J. Reifer and S. Trattner.
A Glossary of Software Tools and Techniques.
Computer, Vol.10(7):52-60, July 1977.

M. Schindler.
Software Practice -- A Scarce Art Struggles to Become a Science.
Electronic Design, pages 85-102, July 22, 1982.

Software Tools: Catalog and Recommendations.
TRW Defense and Space Systems Group, January 1979.

K. Ushijima and K. Harada.
Tools for Analysis and Evaluation of Software.
Information Processing Society of Japan (Joho Shori) (Japan), Vol.20(8):703-11, 1979.

Detailed Articles

R. R. Bate and G. T. Ligler.
An Approach to Software Testing Methodology and Tools.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 476-80.

J. P. Bielski and W. H. Blankertz.
The General Acceptance Test System (GATS).
Proceedings of COMPCON 77, February 28 - March 3, 1977, San Francisco, CA, pages 207-10.

M. E. Boehm, R. K. McClean, and D. B. Urfrig.
Some Experience with Automated Aids to the Design of Large-scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1(1):125-33, March 1975.

J. C. Cherniavsky, W. R. Adrion, and M. A. Branstad.
The Role of Testing Tools and Techniques in the Procurement of Quality Software and Systems.
Conference Record of the Thirteenth Asilomar Conference on Circuits, Systems and Computers, November 5-7, 1979, Pacific Grove, CA, pages 309-13.

R. E. Fairley.
An Experimental Program Testing Facility.
Proceedings of the 1st National Conference on Software Engineering, September 11-12, 1975, Washington, DC, pages 47-55.

S. Katz and Z. Manna.
Towards Automatic Debugging of Programs.
SIGPLAN Notices, Vol.10(6):143-55, June 1975.

V. V. Lipaev, L. A. Serebrovskii, and V. V. Filippovich.
A System for the Automation of Programming and Testing of Complexes of Control Programs, Yauza-6.
Transactions in: Programming and Computer Software, Vol.3(3):233-9, May-June 1977.

D. J. Panzl.
Experience with Automatic Program Testing.
Proceedings of Trends and Applications 1981. Advances in Software Technology, May 28, 1981, Gaithersburg, MD, pages 25-8.

W. Pfadler.
Software Engineering-Principles and Tools for Testing and Measuring.
Data Rep, Vol.12(2):12-6, April 1977.

Progress in Software Engineering.
EDP Analyst, Vol.16(3):1-13, March 1978.

R. R. Prudhomme.
Software Verification and Validation and SQA.
American Society for Quality Control 34th Annual Technical Conference Transactions, May 20-22, 1980, Atlanta, GA, pages 397-404.

R. N. Taylor and L. J. Osterweil.
A Facility for Verification, Testing and Documentation of Concurrent Process Software.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 36-41.

3.2 STATIC ANALYSIS TOOLS

Books

J. G. P. Barnes.
Programming in Ada.
Addisson-Wesley Publishing Company, 1982.

Infotech State of the Art Report, Software Reliability, Volume 1: Analysis and Bibliography.
Infotech International, 1977.

Infotech State of the Art Report, Software Reliability, Volume 2: Invited Papers.
Infotech International, 1977.

G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

Software Engineering Automated Tools Index.
Software Research Associates, P. O. Box 2432, San Francisco, CA 94126.

Survey Articles

W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

AIAA Technical Committee on Computers.
AIAA/Grumman Survey of Software Development Tools Source Data, 1979.

J. C. Browne and D. B. Johnson.
FAST - A Second Generation Program Analysis System.
Proceedings of the 3rd International Conference on Software Engineering, May 10-12, 1978, Atlanta, GA, pages 142-8.

J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY, February 1980.

H. R. Downs.
Automated Tools for the Verification of Computer Programs.
Transactions of the American Nuclear Society and the European Nuclear Society, 1980 International Conference, November 16-21, 1980, Washington, DC, pages 253-4.

R. E. Fairley.
Tutorial - Static Analysis and Dynamic Testing of Computer Software.
Computer, Vol.11(4):14-23, April 1978.

L. D. Fosdick and L. J. Osterweil.
Data Flow Analysis in Software Reliability.
ACM Computer Surveys, Vol.8(3):305-330, September 1976.

R. C. Houghton, Jr.
Software Development Tools.
NBS Special Publication 500-88, National Bureau of Standards, 1982.

W. E. Howden.
A Survey of Static Analysis Methods.
In Tutorial: Software Testing & Validation Techniques, E. Miller and W. E. Howden, Editors, pages 101-15.
IEEE, 1981.

E. F. Miller, Jr.
Notes on Tools and Techniques of Testing.
In Tutorial: Program Testing Techniques, pages 107-11, IEEE Computer Society, Piscataway, NJ, 1977.

I. Miyamoto.
Automated Testing-Aid Tools Survey.
Information Processing Society of Japan (Joho Shori) (Japan), Vol.20(8):688-93, August 1979.

D. J. Reifer and H. A. Montgomery.
SEATECS Software Tools Survey.
RCI-TR-008 (Compiled for the NOSC SEATECS Project), Reifer Consultants, Inc., March 1981.

Detailed Articles

F. E. Allen.
Interprocedural Data Flow Analysis.
Proceedings of the IFIP Congress 1974, pages 398-402, North Holland Publishers, Amsterdam, 1974.

F. E. Allen and J. Cocke.
A Program Data Flow Analysis Procedure.
Communications of the ACM, Vol.19(3):137-47, March 1976.

Automated Testing Analyzer for Cobol.
Software Technology Center, Science Applications, Inc., San Francisco, CA, April 1976.

J. M. Barzdin, J. J. Bicevskis, and A. A. Kalninsh.
Construction of Complete Sample System for Correctness Testing.
Mathematical Foundations of Computer Science, September 1-5, 1975, Marianske Lazne, Czechoslovakia, pages 1-12.

M. E. Boehm, R. K. McClean, and D. B. Urfrig.
Some Experience with Automated Aids to the Design of Large-scale Reliable Software.
IEEE Transactions on Software Engineering, Vol.SE-1(1):125-33, March 1975.

S. H. Caine and E. K. Gordon.
PDL -- A Tool for Software Development.
National Computer Conference, AFIPS Proceedings, 1975.

T. E. Cheatham, Jr. and J. A. Townley.
Program Analysis Techniques for Software Reliability.
Workshop on Reliable Software, September 22-23, 1978, Bonn, Germany, pages 9-17.

A. M. Davis.
The Design of a Family of Application-Oriented Requirements Languages.
Computer, Vol.15(5):21-28, May 1982.

Diagnostic and Debugging Aids for Reliable Software.
In Computer Systems Reliability: International State of the Art Report, W. A. Sampson and C. J. Bunyan (Editors), pages 273-89, 1974.

W. H. Enright.
Using a Testing Package for the Automatic Assessment of Numerical Methods for Codes.
Performance Evaluation of Numerical Software, December 11-15, 1978, Baden, Austria, pages 199-213.

M. E. Fagan.
Design and Code Inspections in the Development of Programs.
1975 International Symposium on Fault-Tolerant Computing. Digest of Papers, June 18-20, 1975, Paris, France, page 248.

M. E. Fagan.
Inspecting Software Design and Code.
Datamation, Vol.23(10):133-44, October 1977.

R. E. Fairley.
An Experimental Program Testing Facility.
Proceedings of the 1st National Conference on Software Engineering, September 11-12, 1975, Washington, DC, pages 47-55.

R. E. Fairley.
Proceedings, Infotech State of the Art Conference on Program Testing, September 1978, London.

B. Flavigny.
A Program for the Detection of Logical Errors in Programs.
Rev. Fr. Autom. Inf. Rech. Oper. (France), Vol.9(B.2):43-59, July 1975.

B. J. Frost.
Program Examination Facility.
New Electronics (GB), Vol.12(11):46, May 29, 1979.

N. Fujimura and K. Ushijima.
Experience with a Cobol Analyzer.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 640-5.

Y. Furukawa and S. Otsuki.
Structure Analysis of Fortran Programs and Its Application to Debugging.
Technology. Rep. Kyushu University (Japan), Vol.49(6):815-21, December 1976.

C. Gannon
A Debugging, Testing and Documentation Tool for Jovial J73.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 634-9.

M. A. Hennell, M. R. Woodward, and D. Hedley.
On Program Analysis.
Information Processing Letters (Netherlands), Vol.5(5):136-40, November 1976.

F. J. Hill and B. Huey.
A Design Language Approach to Test Sequence Generation.
Computer, Vol.10(6):28-34, June 1977.

B. C. Hodges.
A System for Automatic Software Evaluation.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 617-23.

M. Ibramsha and V. Rajaraman.
Detection of Logical Errors in Decision Table Programs.
Communications of the ACM, Vol.21(12):1016-25, December 1978.

W. H. Jessop, J. R. Kane, S. Roy, and J. M. Scanlon.
ATLAS - An Automated Software Testing System.
Proceedings of COMPCON 76, September 7-10, 1976, Washington, DC, pages 337-40.

D. Matuszek.
The Case for the Assert Statement.
SIGPLAN Notices, Vol.11(8):36-7, August 1976.

G. J. Myers.
A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections.
Communications of the ACM, Vol.21(9):760-88, 1978.

G. J. Myers.
Program Design Validation System.
IBM Technical Disclosure Bulletin, Vol.19(10):3806-8, March 1977.

NASA Software Specification and Evaluation System, Final Report.
Science Applications, Huntsville, AL, 1977 (NTIS N77-26828).

A New Generation of Cobol Testing Aids.
Software 73, July 9-11, 1973, Loughborough, Leics, England, pages 139-44.

K. Ochimizu, J. Toyoda, and K. Tanaka.
On a Construction Method of Systems for Detecting Logical Errors in Programs.
System Computer Control, Vol.5(2):88-96, March-April 1974.

L. J. Osterweil and L. D. Fosdick.
DAVE: A Validation Error Detection and Documentation System for Fortran Programs.
Software Practice and Experience, Vol.6(4):473-86, October-December 1976.

J. T. Panttaja.
Detecting Errors in Basic For/Next Statements.
IBM Technical Disclosure Bulletin, Vol.20(7):2352, November 1977.

M. P. Perriens.
An Application of Formal Inspections to Top-Down Structured Program Development.
RADC-TR-77-212, IBM Federal Systems Division, Gaithersburg, MD, 1977, (NTIS AD/A-041645).

S. Pimont and J. C. Rault.
A Software Reliability Assessment Based on a Structural and Behavioral Analysis of Programs.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 486-91.

C. V. Ramamoorthy and S-B. F. Ho.
Testing Large Software with Automated Software Evaluation Systems.
IEEE Transactions on Software Engineering, Vol.SE-1(1):46-58, March 1975.

H. Schneider and M. Schulz.
Debug Program KESP for Robotron 4000.
Rechentech. Datenverarb. (Germany), Vol.14(2):28-9, February 1979.

H. Sneed.
Softdoc - A System for Automated Software Static Analysis and Documentation.
Performance Evaluation Review, Vol.10(1):173-7, Spring 1981.

J. E. Stockenberg and A. Van Dam.
Structured Programming Analysis System.
Proceedings of the 1st National Conference on Software Engineering, September 11-12, 1975, Washington, DC, pages 27-36.

L. G. Stucki.
Software Development Tools - Acquisition Considerations - A Position Paper.
National Computer Conference, AFIPS Proceedings, Vol.46:267-8, 1977.

R. N. Taylor and L. J. Osterweil.
Anomaly Detection in Concurrent Software by Static Data Flow Analysis.
IEEE Transactions on Software Engineering, Vol.SE-6(3):265-76, May 1980.

D. Teichrow and E. A. Hershey, III.
PSL/PSA: A Computer-aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering, Vol.SE-3(1):41-8, January 1977.

K. Ushijima and T. Shibahara.
Test Programs for Detecting Optimization Techniques in Fortran.
Technol. Rep. Kyushu Univ. (Japan). Vol.47(3):313-7, June 1974.

M. Vierling and P. Ernst.
A Software Aid for Program Testing.
Elektronik (Germany), Vol.28(1):52-6, January 11, 1979.

D. Vitas.
On the Automatic Analysis of the Structure of Fortran Programs.
Informatica 78 XIII Yugoslav International Symposium on Information Processing, October 2-7, 1978, Bled, Yugoslavia.

I. K. Wendel and R. L. Kleir.
Fortran Error Detection through Static Analysis.
Software Engineering Notes, Vol.2(3):22-8, March 1977.

T. Yamada, Y. Honda, N. Shigematsu, and M. Tomura.
ESS Program Test Methods Using Program Route Tracer.
Electronics Communication Laboratory Technical Journal (Japan), Vol.30(1):235-49, 1981.

3.3. DYNAMIC ANALYSIS TOOLS

Books

A. T. Acree.
On Mutation.
Ph.D. Thesis, Georgia Institute of Technology, 1980.

T. A. Budd.
Mutation Analysis of Program Test Data.
Ph.D. Thesis, Yale University, 1980.

R. A. DeMillo.
Program Mutation: An Approach to Software Testing.
Report GIT/ICS-83-03, Georgia Institute of Technology.
January 1983.

Infotech State of the Art Report, Software Reliability, Volume 2: Invited Papers.
Infotech International, 1977.

Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography.
Infotech International, 1979.

G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

Proceedings of the Specifications of Reliable Software Conference.
IEEE Catalog No. CH1401-9c, IEEE, New York, 1979.

Software Engineering Automated Tools Index.
Software Research Associates, P. O. Box 2432, San Francisco, CA 94126.

Survey Articles

W. R. Adrion, M. A. Branstad, and J. C. Cherniavsky.
Validation, Verification, and Testing of Computer Software.
NBS Special Publication 500-75, National Bureau of Standards, pages 32-5.

AIAA Technical Committee on Computers.
AIAA/Grumman Survey of Software Development Tools Source Data, 1979.

J. C. Browne and D. B. Johnson.
FAST - A Second Generation Program Analysis System.
Proceedings of the 3rd International Conference on Software
Engineering, May 10-12, 1978, Atlanta, GA, pages 142-8.

T. A. Budd.
Mutation Analysis: Ideas, Examples, Problems and Prospects.
In Computer Program Testing, B. Chardrasekaran and S. Radicchi,
Editors. North-Holland, 1981.

B. Chandrasasekaran.
Test Tools: Usefulness Must Extend to Everyday Programming
Environment.
Computer, Vol.12(3):102-3, March 1979.

T. E. Cheatham, Jr., G. H. Holloway, and J. A. Townley.
Symbolic Evaluation and the Analysis of Programs.
IEEE Transactions on Software Engineering, Vol.SE-5(4):402-17,
July 1979.

L. A. Clarke.
Automatic Test Data Selection Techniques.
In Infotech State of the Art Report, Software Testing, Volume 2:
Invited Papers, pages 44-63.
Infotech International, 1979.

J. A. Darringer and J. C. King.
Application of Symbolic Execution to Program Testing.
Computer, Vol.11(4):51-60, April 1978.

R. A. DeMillo.
Mutation Analysis as a Tool For Software Quality Assurance.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Hints on Test Data Selection: Help for the Practicing Programmer.
Computer, Vol.11(4):34-41, April 1978.

J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY,
February 1980.

H. R. Downs.
Automated Tools for the Verification of Computer Programs.
Transactions of the American Nuclear Society and the European
Nuclear Society, 1980 International Conference, November 16-21,
1980, Washington, DC, pages 253-4.

R. C. Houghton, Jr.
Software Development Tools.
NBS Special Publication 500-88, National Bureau of Standards, 1982.

J. C. Huang.
Program Instrumentation.
In Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography, pages 144-50. Infotech International, 1979.

J. C. Huang.
Program Instrumentation and Software Testing.
Computer, Vol.11(4):25-31, April 1978.

R. J. Lipton and F. G. Sayward.
The Status of Research on Program Mutation.
Digest for the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, FL, 1978, pages 355-73.

G. A. Mann.
A Survey of Debug Systems.
Honeywell Computer Journal, Vol.7(3):182-98, 1973.

E. F. Miller, Jr.
Notes on Tools and Techniques of Testing.
In Tutorial: Program Testing Techniques, pages 107-11, IEEE Computer Society, Piscataway, NJ, 1977.

I. Miyamoto.
Automated Testing-Aid Tools Survey.
Information Processing Society of Japan (Joho Shori) (Japan), Vol.20(8):688-93, August 1979.

C. V. Ramamoorthy.
Techniques for Automated Test Data Generation.
Conference Record of the Ninth Asilomar Conference on Circuits, Systems, and Computers, November 1975.

D. J. Reifer and H. A. Montgomery.
SEATECS Software Tools Survey.
RCI-TR-008 (Compiled for the NOSC SEATECS Project), Reifer Consultants, Inc., March 1981.

Detailed Articles

A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward.
Mutation Analysis.
Report GIT/ICS-79-08, Georgia Institute of Technology, 1979.

J. M. Adams.
Experiments on the Utility of Assertions for Debugging.
Proceedings Eleventh Hawaii International Conference on System Science, Honolulu, HI, January 1978, pages 31-9.

D. M. Andrews.
Using Executable Assertions for Testing and Fault Tolerance.
Ninth Annual International Symposium on Fault-Tolerant Computing, June 20-22, 1979, Madison, WI, pages 102-5.

D. M. Andrews and J. P. Benson.
An Automated Program Testing Methodology and Its Implementation.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 254-61.

D. M. Andrews and J. P. Benson.
Using Executable Assertions for Testing.
Conference Record of the Thirteenth Asilomar Conference on Circuits, Systems and Computers, November 5-7, 1979, Pacific Grove, CA, pages 302-5.

J. Arthur and J. Ramanathan.
Design of Analyzers for Selective Program Analysis.
IEEE Transactions on Software Engineering, Vol.SE-7(1):39-51, January 1981.

D. Baldwin and F. Sayward.
Heuristics for Determining Equivalence of Program Mutations.
Technical Report 161, Yale University, 1979.

V. R. Basili and R. E. Noonan.
A Testing Tool for a Fire-Control Environment.
Proceedings of COMPCON 76, September 7-10, 1976, Washington, DC, pages 341-5.

J. D. Baum and J. M. Baca.
Real-Time Event Trace Monitor for Embedded Computer Systems.
Proceedings of the IEEE 1979 National Aerospace and Electronics Conference, May 15-17, 1979, Dayton, OH, pages 833-9.

A. M. Belyakin, S. F. Zan-Ko, V. I. Medvedev, and V. N. Yakhontov.
Soneya-A System for Testing Programs Based on the Use of a Limited Natural Language.
Programmirovanie (USSR), Vol.5(2):84-9, March-April 1979.

J. F. Benders.
Partitioning Procedures for Solving Mixed-Variables Programming Problems.
Numerical Math, Vol.4:238, 1962.

J. Bicevskis, J. Borzovs, U. Straujums, A. Zarins, and E. F. Miller, Jr.
SMOTL - A System to Construct Samples for Data Processing Program Debugging.
IEEE Transactions on Software Engineering, Vol.SE-5(1):60-6, January 1979.

S. Bologna.
TEVERE-1: A Software System for Program Testing and Verification.
AICA 79 Conference, October 10-13, 1979, Bari, Italy, pages 71-8.

S. Bologna and J. Taylor.
SSPTV-A Software System for Program Testing and Verification.
Danish Atomic Energy Commission, Roskilde, Denmark, March 1978.

T. L. Booth, R. Ammar, and R. Lenk.
An Instrumentation System to Measure User Performance in Interactive Systems.
Journal of Systems and Software, Vol.2(2):139-46, June 1981.

Y. V. Borzov.
Program Testing Using Symbolic Execution.
Programming and Computer Software, Vol.16:39-45, 1980.

R. S. Boyer, B. Elspas, and K. N. Levitt.
SELECT - A Formal System for Testing and Debugging Programs by Symbolic Execution.
SIGPLAN Notices, Vol.10(6):234-45, June 1975.

T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
The Design of a Prototype Mutation System for Program Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:623-7, 1978. Also reprinted in Tutorial: Automated Tools for Software Engineering, E. F. Miller, Editor. IEEE Computer Society, 1979.

T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Program.
7th ACM Symposium on Principles of Programming Languages, January 1980.

T. A. Budd, R. Hess, and F. G. Sayward.
EXPER Implementor's Guide.
Department of Computer Science, Yale Univeristy.

J. C. Cherniavsky.
On Finding Test Data Sets for Loop Free Programs.
Information Processing Letters (Netherlands), Vol.8(2):106-7, February 15, 1979.

T. S. Chow.
A Generalized Assertion Language.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 392-9.

J. Cohen and N. Carpenter.
A Language for Inquiring about the Run-Time Behaviour of Programs.
Software Practice and Experience, Vol.7(4):445-60, July-August 1977.

L. Clarke.
Test Data Generation and Symbolic Execution as an Aid in Software Validation.
Computer Science Conference, February 18-20, 1975, Washington, DC, page 41.

L. A. Clarke.
A System to Generate Test Data and Symbolically Execute Programs.
IEEE Transactions on Software Engineering, Vol.SE-2(3):215-22, September 1976.

L. A. Clarke.
Testing - Achievements and Frustrations.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 310-4.

L. A. Clarke, J. Hassell, and D. J. Richardson.
A Close Look at Domain Testing.
IEEE Transactions on Software Engineering, Vol.SE-8(4):380-90, July 1982.

J. A. Darringer.
EFFIGY: A System for Symbolic Execution.
Computer Science Conference, February 18-20, 1975, Washington, DC, page 41.

J. A. Darringer.
The Use of Symbolic Execution in Program Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 67-85.
Infotech International, 1979.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward.
Program Mutation: A New Approach to Program Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 107-26.
Infotech International, 1979.

Diagnostic and Debugging Aids for Reliable Software.
In Computer Systems Reliability: International State of the Art Report, W. A. Sampson and C. J. Bunyan (Editors), pages 273-89, 1974.

E. G. Dupnick.
A Zero-One Integer Programming Solution for Determining the Minimum Number of Test Cases Required for Fortran Program Checkout.
Bulletin of the Operations Research Society of America, Vol.21(2):B12, 1973.

R. E. Fairley.
An Experimental Program Testing Facility.
Proceedings of the 1st National Conference on Software Engineering, September 11-12, 1975, Washington, DC, pages 47-55.

D. W. Fife.
Test Data Generation: Three Approaches Prevail.
Computer, Vol.12:103-4, March 1979.

K. F. Fischer.
A Test Case Selection Method for the Validation of Software Maintenance Modifications.
Proceedings of COMPSAC 77, November 8-11, 1977, Chicago, IL, pages 421-6.

H. W. Flanagan.
Program Debugging System.
IBM Technical Disclosure Bulletin, Vol.16(7):2322-9, December 1973.

B. Flavigny.
A Program for the Detection of Logical Errors in Programs.
Rev. Fr. Autom. Inf. Rech. Oper. (France), Vol.9(B.2):43-59, July 1975.

K. A. Foster.
Error Sensitive Test Cases Analysis (ESTCA).
IEEE Transactions on Software Engineering, Vol.SE-6(3):258-64, May 1980.

H. E. Frye and R. L. Hoffman.
Program Event Monitoring Instruction.
IBM Technical Disclosure Bulletin, Vol.18(5):1557-9, October 1975.

R. E. Fryer.
The User Interface for a Real-Time Software Debugging System.
Conference Record of the Fourteenth Asilomar Conference on Circuits, Systems and Computers, November 17-19, 1980, Pacific Grove, CA, pages 462-9.

N. Fujimura and K. Ushijima.
Experience with a Cobol Analyzer.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 640-5.

C. Gannon.
JAVS - A Jovial Automated Verification System.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 539-44.

W. Geiger.
Experience with an Automated Test and Documentation System for Fortran Programs.
Practice in Software Adaption and Maintenance. Proceedings for the SAM Workshop, pages 143-56, April 5-6, 1979, Berlin, Germany.

M. Gerisch.
Universal Programming and Test Aids for OS/ES.
Rechentech. Datenverarb. (Germany), Vol.15(2):5-7, February 1978.

C. Ghezzi and M. Jazayeri.
Syntax Directed Symbolic Execution.
Proceedings of COMPSAC 80, October 27-31, 1980, Chicago, IL, pages 539-45.

B. Gladstone.
Monitor/Debugger Saves Time when Checking MUP Software.
EDN, Vol.21(17):69-80, September 1976.

L. Gmeiner.
Dynamic Analysis and Test Data Generation in an Automatic Test System.
Workshop on Reliable Software, September 22-23, 1978, Bonn, Germany, pages 31-48.

R. E. Gomory.
An Algorithm for Integer Solutions to Linear Programs.
In Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe, Editors. McGraw-Hill, New York, 1963.

A. I. Halsema.
Demons: A Symbolic Debugging Monitor.
BYTE, Vol.6(5):326-58, May 1981.

C. Hanisch.
Experience with a Computer Operating System Test Environment.
Rechentech. Datenverarb. (Germany), Vol.17(5):11-3, May 1980.

J. M. Hanks.
Testing Cobol Programs by Mutation: Volume I - Introduction to the CMS.1 System, Volume II - CMS.1 System Documentation.
Report GIT/ICS-80-04, Georgia Institute of Technology, 1980.

B. C. Hedfors and B. J. E. Nilsson.
The APA System, An Integrated Software Test System.
International Conference on Software Engineering for Telecommunication Switching Systems, February 18-20, 1976, Salzburg, Austria, pages 104-7.

M. A. Hennell.
An Experimental Testbed for Numerical Software.
Computer Journal (GB), Vol.21(4):333-6, November 1978.

F. J. Hill and B. Huey.
A Design Language Approach to Test Sequence Generation.
Computer, Vol.10(6):28-34, June 1977.

W. E. Howden.
An Evaluation of the Effectiveness of Symbolic Testing.
Software Practice and Experience, Vol.8(4):381-97, July - August 1978.

W. E. Howden.
DISSECT - A Symbolic Evaluation and Program Testing System.
IEEE Transactions on Software Engineering, Vol.SE-4(1):70-3, January 1978.

W. E. Howden.
Experiments with a Symbolic Evaluation System.
National Computer Conference, AFIPS Proceedings, June 1976, pages 899-908.

W. E. Howden.
Methodology for the Generation of Program Test Data.
IEEE Transactions on Computers, Vol.C-24(5):554-9, May 1975.

W. E. Howden.
Symbolic Testing and the DISSECT Symbolic Evaluation System.
IEEE Transactions on Software Engineering, Vol.SE-3(4):276-8, July 1977.

W. E. Howden.
Weak Mutation Testing and Completeness of Test Sets.
IEEE Transactions on Software Engineering, Vol.SE-8(4):371-9, July 1982.

J. C. Huang.
Detection of Data Flow Anomaly Through Program Instrumentation.
IEEE Transactions on Software Engineering, Vol.SE-5(3):226-36, May 1979.

J. C. Huang.
Instrumenting Programs for Data Flow Analysis.
Technical Report TR-UH-CS-77-4, University of Houston, May 1977.

J. C. Huang.
Program Instrumentation: A Tool for Software Testing.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 147-59. Infotech International, 1979.

J. C. Huang and R. T. Yeh.
A Method for Test-Case Generation.
2nd USA-Japan Computer Conference Proceedings, August 26-28, 1975, Tokyo, Japan, pages 585-9.

C. E. Hughes and C. P. Pfleeger.
Assist-V - An Environment Simulator for IBM 360 Systems Software Development.
IEEE Transactions on Software Engineering, Vol.SE-4(6):526-30, November 1978.

E. Husu.
PROSIT - A Tool for the Testing of Proteo Central Control Software (Electronic Switching Systems).
Telecomunicazioni (Italy), No.71-72:63-8, July-October 1979.

J. C. King.
Symbolic Execution and Program Testing.
Communications of the ACM, Vol.19(7):385-94, July 1976.

D. Konig and A. Rink.
PIPS - A Programming System for Interactive Program Testing.
Workshop on Reliable Software, September 22-23, 1978, Bonn, Germany, pages 89-100.

S. Kundu.
A New Program Analysis Technique with Applications to Test Case Generation.
3rd USA-Japan Computer Conference Proceedings, October 10-12, 1978, San Francisco, CA, pages 482-6.

S. Kundu.
SETAR - New Approach to Test Case Generation.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 163-87. Infotech International, 1979.

S. F. Lundstrom.
Adaptive Random Data Generation for Computer Software Testing.
National Computer Conference, AFIPS Proceedings, Vol.47:505-12, 1978.

N. R. Lyons.
An Automatic Data Generation System for Data Base Simulation and Testing.
Data Base, Vol.8(4):10-3, 1977.

K. Malecki.
Using Software Monitor for Evaluation and Measurement of Computer Systems.
Informatyka (Poland), Vol.12(4):15-8, April 1976.

M. A. Malik.
An Assertion Language for the Annotation of Program Modules.
Australian Computer Science Community (Australia), Vol.2(2):217-38, March 1980.

H. Marks.
Probe/1, A Tool for Verifying Software.
1979 Summer Computer Simulation Conference, July 16-18, 1979, Toronto, Canada, pages 635-8.

Y. Matsumoto and T. Torii.
HMTS Intended to Realize High-Efficiency Program Test.
Hitachi Review (Japan), Vol.25(7):263, July 1976.

O. E. Melkonyan.
The Sekont Program Testing System.
Programmirovanie (USSR), Vol.3(3):83-7, May-June 1977.

A. R. Miller.
ZSID, Z-80 Debugger for CP/M.
Interface Age, Vol.5(8):88-90, August 1980.

W. Miller and D. L. Spooner.
Automatic Generation of Floating-Point Test Data.
IEEE Transactions on Software Engineering, Vol.SE-2(3):223-6, September 1976.

G. J. Myers and D. G. Hocker.
The Use of Software Simulators in the Testing and Debugging of Microprogram Logic.
IEEE Transactions on Computers, Vol.C-30(7):519-23, July 1981.

A New Generation of Cobol Testing Aids.
Software 73, July 9-11, 1973, Loughborough, Leics, England, pages 139-44.

E. S. Novikov and Ya. A. Khetagurov.
Reliability from a Method of Monitoring and Correcting Computations.
Programmirovanie (USSR), Vol.5(6):75-82, November-December 1979.

NSW Feasibility Study, Final Technical Report, RADC-TR-78-23, February 1978.

L. J. Osterweil and L. D. Fosdick.
DAVE -- A Validation and Error Detection System for Fortran Programs.
Software Practice and Experience, Vol.6:473-86, 1976.

L. J. Osterweil and L. D. Fosdick.
Simulated Program Execution as a Strategy for Error Detection and Validation.
Proceedings of the 1976 Summer Computer Simulation Conference, July 12-14, 1976, Washington, DC, pages 704-7.

M. R. Paige and J. P. Benson.
The Use of Software Probes in Testing Fortran Programs.
Computer, Vol.7(7):40-7, July 1974.

M. R. Paige and A. H. Tindell.
A System for Test Data Generation.
Proceedings of the IEEE 1979 National Aerospace and Electronics Conference, May 15-17, 1979, Dayton, OH, pages 826-8.

R. J. Peterson.
TESTER/1: An Abstract Model for the Automatic Synthesis of Program Test Case Specifications.
Proceedings of the Symposium of Computer Software Engineering, pages 629-35, IEEE, New York, 1976.

N. M. Petukhova and L. N. Savenko.
Simulator for De-Bugging Program.
Autom. Sist. Upr. (USSR), No.1:76-81, 1974.

P. A. Pravil-Shchikov and V. S. Shchepin.
Compilation of Structural Programs in a Dialogue Mode with Concurrent Test Generation.
Avtom. and Telemekh. (USSR), Vol.40(8):129-38, August 1979.

R. L. Probert.
Optimal Insertion of Software Probes in Well-Delimited Programs.
IEEE Transactions on Software Engineering, Vol.SE-8(1):34-42, January 1982.

C. V. Ramamoorthy and K. H. Kim.
Software Monitors Aiding Systematic Testing and Their Optimal Placement.
Proceedings of the 1st National Conference on Software Engineering, September 11-12, 1975, Washington, DC, pages 21-6.

C. V. Ramamoorthy, K. H. Kim, and W. T. Chen.
Optimal Placement of Software Monitors Aiding Systematic Testing.
IEEE Transactions on Software Engineering, Vol.SE-1(4):403-11, December 1975.

C. V. Ramamoorthy, F. Ho Siu-Bun, and W. T. Chen.
On the Automated Generation of Program Test Data.
IEEE Transactions on Software Engineering, Vol.SE-2(4):293-300, December 1976.

D. J. Richardson and L. A. Clarke.
A Partition Analysis Method to Increase Program Reliability.
Proceedings of the 5th International Conference on Software Engineering, March 9-12, 1981, San Diego, CA, pages 244-53.

I. J. Riddle, J. A. Hennel, M. R. Woodward, and D. Hedley.
Practical Aspects of Program Mutation.
University of Nottingham, Nottingham, UK.

G. R. Sager.
Emulation for System Measurement/Debugging.
Proceedings of the IFIP Conference on Software for Minicomputers, September 8-12, 1975, Keszthely, Hungary, pages 107-23.

S. H. Saib.
Executable Assertions - An Aid to Reliable Software.
Proceedings Eleventh Annual Asilomar Conference on Circuits, Systems, and Computers, November 1977, Pacific Grove, CA, pages 277-81.

P. Schmitz, R. Van Megen, and H. Bons.
Methods of Systematic Test Case Determination and Test Case Preparation.
Practice in Software Adaption and Maintenance. Proceedings for the SAM Workshop, April 5-6, 1979, Berlin, Germany, pages 209-21.

L. G. Stucki.
Software Development Tools - Acquisition Considerations - A Position Paper.
National Computer Conference, AFIPS Proceedings, Vol.46:267-8, 1977.

L. G. Stucki and G. L. Foshee.
New Assertion Concepts for Self-Metric Software Validation.
Proceedings of IEEE Conference on Reliable Software, April 1975, Los Angeles, CA, pages 59-65.

A. Tanaka.
Equivalence Testing for Fortran Mutation System Using Data Flow Analysis.
Department of Information and Computer Science, Georgia Institute of Technology, 1981.

R. N. Taylor.
Assertions in Programming Languages.
SIGPLAN Notices, Vol.15(1):58-65, January 1980.

D. Teichroew and F. A. Hershey, III.
PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems.
IEEE Transactions on Software Engineering, Vol.SE-3(1):41-8, 1977.

U. Voges, L. Gmeiner and A. Amschler Von Mayrhauser.
SADAT - An Automated Testing Tool.
IEEE Transactions on Software Engineering, Vol.SE-6(3):286-90, May 1980.

Chen Wen-Tsuen, Ho Jone-Ping and Wen Chia-Hsien.
Dynamic Validation of Programs Using Assertion Checking Facilities.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 533-8.

M. R. Woodward, D. Hedley, and M. A. Hennell.
Experience with Path Analysis and Testing of Programs.
IEEE Transactions on Software Engineering, Vol.SE-6(3):278-86, May 1980.

M. R. Woodward, M. A. Hennell, and D. Hedley.
A Limited Mutation Approach to Program Testing.
University of Nottingham, Nottingham, UK, 1980.

3.4. TEST SUPPORTING TOOLS

Books

Infotech State of the Art Report: Software Reliability, Volume 2: Invited Papers.
Infotech International, 1977.

Infotech State of the Art Report, Software Testing, Volume 1: Analysis and Bibliography.
Infotech International, 1979.

G. J. Myers.
The Art of Software Testing.
John Wiley & Sons, New York, 1979.

Software Engineering Automated Tools Index.
*Software Research Associates, P. O. Box 24*32, San Francisco, CA 94126.

Survey Articles

R. P. Abbott.
Towards the Audit of Computer Software.
In *Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers*, pages 1-12.
*Infotech In*ternational, 1979.

W. L. Bryan, S. G. Siegel, and G. L. Whiteleather.
Auditing Throughout the Software Life Cycle: A Primer.
Computer, Vol.15(3):57-67, March 1982.

T. E. Cheatham, Jr.
Comparing Programming Support Environments.
Proceedings of the Symposium on Software Engineering Environments, June 16-20, 1980, Lahnstein, Germany, pages 11-25.

J. D. Donahoo and D. Swearingen.
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center, NY, February 1980.

H. R. Downs.
Automated Tools for the Verification of Computer Programs.
Transactions of the American Nuclear Society and the European Nuclear Society, 1980 International Conference, November 16-21, 1980, Washington, DC, pages 253-4.

R. C. Houghton, Jr.
Software Development Tools.
NBS Special Publication 500-88, National Bureau of Standards, 1982.

E. F. Miller, Jr.
Notes on Tools and Techniques of Testing.
In Tutorial: Program Testing Techniques, pages 107-11, IEEE Computer Society, Piscataway, NJ, 1977.

I. Miyamoto.
Automated Testing-Aid Tools Survey.
Information Processing Society of Japan (Joho Shori) (Japan), Vol.20(8):688-93, August 1979.

D. J. Panzl.
Automatic Software Test Drivers.
Computer, Vol.11(4):44-50, April 1978.

D. J. Reifer and H. A. Montgomery.
SEATECS Software Tools Survey.
RCI-TR-008 (Compiled for the NOSC SEATECS Project), Reifer Consultants, Inc., March 1981.

Detailed Articles

S. R. Alpert.
Minicomputer Peripheral Aids Program Debugging.
Computer Design, Vol.15(9):104-8, September 1976.

D. L. Baggi and M. L. Shooman.
An Automatic Driver for Pseudo-Exhaustive Software Testing.
Proceedings of Spring COMPCON 78, February 28 - March 3, 1978, San Francisco, CA, pages 278-82.

D. G. Bate.
Design and Implementation of an Interactive Test Bed.
Software Practice and Experience, Vol.4(1):91-109, January-March 1974.

J. P. Benson and R. A. Melton.
A Laboratory for the Development and Evaluation of BMD Software Quality Enhancements Techniques.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 106-9.

J. P. Bielski and W. H. Blankertz.
The General Acceptance Test System (GATS).
Proceedings of COMPCON 77, February 28 - March 3, 1977, San Francisco, CA, pages 207-10.

S. H. Caine and E. K. Gordon.
PDL -- A Tool for Software Development.
National Computer Conference, AFIPS Proceedings, 1975.

L. M. Culpepper.
A System for Reliable Engineering Software.
IEEE Transactions on Software Engineering, Vol.SE-1(2):174-8,
June 1975.

D. M. Garmer and J. W. Fish.
Automatic Test Equipment Software Testing.
Proceedings of Autotestcon '78, November 28-30, 1978, San Diego,
CA, pages 71-5.

G. J. Guitonneau, J. L. Lemonnier, and J. J. Soulet.
A Simulator for Debugging and Evaluating the E11 System.
International Conference on Software Engineering for
Telecommunication Switching Systems, February 18-20, 1976,
Salzburg, Austria, pages 108-11.

D. C. Hart, R. G. Kurkjian, M. Myles, and A. Greenspan.
A Computer-Independent Arinc Atlas Syntax Comparator.
Proceedings of Autotestcon '76, November 10-12, 1976, Arlington,
TX, pages 10-8.

C. A. Heuermann, G. J. Myers, and J. H. Winterton.
Automated Test and Verification.
IBM Technical Disclosure Bulletin, Vol.17(7):2030-5,
December 1974.

C. A. Heuermann, G. J. Myers, and J. H. Winterton.
Verification of Test Case Output.
IBM Technical Disclosure Bulletin, Vol.17(7):2034-5, December
1974.

P. P. Howley, Jr. and G. R. Sadow.
Software Verification for Large-Scale ATE Systems.
Proceedings of Autotestcon '78, November 28-30, 1978, San Diego,
CA, pages 76-83.

J. G. Kretzschmar.
Microprocessor-Based Testing of Microprocessor Programs.
Neue Technical Buero (Germany), Vol.22(1):2-5, January-February
1978.

L. M. Lemon.
Hardware System for Developing and Validating Software.
Conference Record of the Thirteenth Asilomar Conference on
Circuits, Systems and Computers, November 5-7, 1979, Pacific
Grove, CA, pages 455-9.

A. R. Miller.
ZSID, Z-80 Debugger for CP/M.
Interface Age, Vol.5(8):88-90, August 1980.

J. B. Norris.
Component Test Aid.
IBM Technical Disclosure Bulletin, Vol.20(5):1870-2, October 1977.

S. M. Nugent.
Automatic Comparison of Numerical Results Files.
Practice in Software Adaption and Maintenance, Proceedings for the SAM Workshop, April 5-6, 1979, Berlin, Germany, pages 107-20.

D. J. Panzl.
A Language for Specifying Software Tests.
National Computer Conference, AFIPS Proceedings, Vol.47:609-19, 1978.

D. J. Panzl.
Test Procedures: A New Approach to Software Verification.
Proceedings of the 2nd International Conference on Software Engineering, October 13-15, 1976, San Francisco, CA, pages 477-85.

R. Press and R. K. McClean.
The Flexible Analysis, Simulation, and Test Facility: A Practical Software-First Capability.
Proceedings of the IEEE 1976 National Aerospace and Electronics Conference, May 18-20, 1976, Dayton, OH, pages 264-8.

N. S. Prywes, A. Pnueli, and A. Shastry.
Use of a Nonprocedural Specification Language and Associated Program Generator in Software Development.
Transactions on Programming Languages and Systems, Vol.1(2):196-217, October 1979.

B. G. Ryder.
The PFORT Verifier.
Software Practice and Experience, Vol.4(4):359-77, October-December 1974.

A. J. Shils.
Creating, Evolving, and Using an Online Large Software System Test Base.
IBM Technical Disclosure Bulletin, Vol.17(11):3292-7, April 1975.

H. M. Sneed.
Prufstand - A Testbed for Systems Software Components.
In Infotech State of the Art Report, Software Testing, Volume 2: Invited Papers, pages 246-70. Infotech International, 1979.

R. A. Stasko.
Test Case Monitor.
IBM Technical Disclosure Bulletin, Vol.17(10):2976, March 1975.

R. N. Taylor and L. J. Osterweil.
A Facility for Verification, Testing and Documentation of Concurrent Process Software.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 36-41.

D. E. Wright and B. D. Carroll.
An Automated Data Collection System for the Study of Software Reliability.
Proceedings of COMPSAC 78, November 13-16, 1978, Chicago, IL, pages 571-6.

A. D. Zakrevskii.
Dialog System for Program Debugging and Editing in the Lyapas-M Language.
Kibernetika (USSR), Vol.10(6):984-5, November-December 1974.

APPENDIX A

INFORMATION SOURCES FOR TESTING TOOLS

1. Raymond C. Houghton, Jr.
   Software Development Tools.
   NBS Special Publication 500-88, National Bureau of Standards, 1982.

2. John D. Donahoo and Dorothy Swearingen.
   A Review of Software Maintenance Technology.
   RADC-TR-80-13, Interim Report, February 1980, Rome Air Development Center.

3. Data Analysis Center for Software (DACS).
   RADC/ISISI, Griffiss Air Force Base, NY 13441.
   Telephone 315-336-0937, AUTOVON 587-3395.

4. Frank S. Lamonica.
   RADC/COEE, Griffiss Air Force Base, NY 13441.
   Telephone 315-330-3977.

5. M. Finfer, et al.
   Software Debugging Methodology, Final Technical Report.
   RADC-TR-79-57 (three volumes), April 1979.

6. AIAA Technical Committee on Computers.
   AIAA/Grumman Survey of Software Development Tools Source Data, 1979.

7. Applied Systems Design Section, Software Tools: Catalog and Recommendations.
   TRW Defense and Space Systems Group, January 1979.

8. D. J. Reiffer and H. A. Montgomery.
   SEATECS Software Tools Survey.
   RCI-TR-008 (Compiled for the NOSC SEATECS Project), Reiffer Consultants, Inc., March 1981.

9. L. G. Stucki, et al.
   Methodology for Producing Reliable Software.
   McDonnel-Douglas Astronautics Company, NASA CR 144769 (two volumes), March 1976.

10. Software Engineering Automated Tool Index.
    Software Research Associates, P. O. Box 2432, San Francisco, CA 94126, Tel. 415-957-1441, Telex 340-235, March 1982.

11. Max Schindler.
Software Practice -- A Scarce Art Struggles to Become a Science.
Electronic Design, July 22, 1982, pp. 85-102.

12. Herbert Hecht.
Final Report: A Survey of Software Tools Usage.
NBS Special Publication 500-82, Computer Science and Technology.
National Bureau of Standards, November 1981.

APPENDIX B

ALPHABETICAL LISTING OF CATALOGED TOOLS
(Tools described by data sheets marked with *)

1.	ADF	
2.	ADS	
3.	ADS/CERL	
4.	AFFIRM	
5.	AISIM	
6.	AMPIC	*
7.	ARGUS/MICRO	
8.	ARTS	
9.	ASSETT	
10.	ASSIST-I	
11.	ATA-FASP	
12.	ATA-SAI	
13.	ATDG	*
14.	ATTEST	*
15.	AUDIT	
16.	AUDITOR	
17.	AUTO-DBO	
18.	AUTOFLOW	
19.	AUTORETEST	*
20.	BEST/1	
21.	CADA	
22.	CADSAT	
23.	CALLREF	
24.	CAPTURE/MVS	
25.	CARA	
26.	CASEGEN	*
27.	CAVS	
28.	CCA	
29.	CCREF	
30.	CCS	
31.	CGJA	
32.	CICS DUMPN ANALY	
33.	COBOL/ADE	
34.	COBOL/DV	*
35.	COBOL OPTIMIZER	*
36.	COBOL/QDM	
37.	COBOL STRUCT	
38.	COBOL/SP	
39.	COBOL TESTING	*
40.	COBOL TRACING	*
41.	COMGEN	
42.	COMGEN/TRW	
43.	COMLIST	
44.	COMLIST/TRW	

ALPHABETICAL LISTING OF CATALOGED TOOLS
(Tools described by data sheets marked with *)

45.	COMMAP	*
46.	COMPARE DBCOMP	
47.	COMPARISON	
48.	COMSCAN	
49.	COMSORT	
50.	CONFIG	
51.	CONFIGURATOR	
52.	CORE	
43.	COTUNE-II	
54.	CPA-ADR	
55.	CROREF	
56.	CRYSTAL	
57.	CUE	
58.	DA	
59.	DATAMACS	*
60.	DAS	
61.	DARTS	
62.	DAVE	*
63.	DCD	
64.	DDPM	
65.	DECKBOY COMPARE	
66.	DPECHT	
67.	DICTANL/OCATE	
68.	DIFFS	*
69.	DISSECT	*
70.	DOCUTOOL	*
71.	DPAD	
72.	DPNDCY	
73.	DRIVER	
74.	DYNA	*
75.	EAVS	
76.	ECA AUTOMATION	
77.	EFFIGY	*
78.	ENFORCE	
79.	EXPEDITER	*
80.	FACES	*
81.	FADEBUG-I	*
82.	FASP	
83.	FAST	*
84.	FAVS	*
85.	FCA	*
86.	FLOBOL	
87.	FORAN	*
88.	FORREF	
89.	FORTRAN AUDITOR	*
90.	FORTRAN OPTIMIZER	*

ALPHABETICAL LISTING OF CATALOGED TOOLS
(Tools described by data sheets marked with *)

91.	FORTRAN TESTING	*
92.	FORTRAN TRACING	*
93.	FORTREF	
94.	FTN ANALYZER	
95.	FTN-77 ANALYZER	
96.	FTNXREF	
97.	GAVS	
98.	GENTESTS	
99.	GENTEXTS	*
100.	GIRAFF	
101.	GOTO-ANALYZER	
102.	HARDWARE SIMULA	
103.	HAWKEYE	
104.	IFTRAN	
105.	INFORM/REFORM	
106.	INSERT	
107.	INSTRU	
108.	INTERFACE DOCUM	*
109.	IPDS	
110.	ISUS	
111.	ITB	
112.	JAVS	*
113.	JIGSAW	
114.	JOVIAL TCA	
115.	JOVIAL/VS	
116.	JOYCE	*
117.	LEXICON	
118.	LIBREF	
119.	LOGIC	
120.	LOGICFLOW	
121.	LOGOS	
122.	LOOK	
123.	MED-SYS	
124.	MEDL-P	
125.	MEDL-R	
126.	MONITOR	
127.	MSEF	
128.	NASA-VATS	
129.	NODAL	
130.	OCM	
131.	PACD	*
132.	PACD-C	
133.	PBASIC	
134.	PDL	
135.	PDL/PSA	

ALPHABETICAL LISTING OF CATALOGED TOOLS
(Tools described by data sheets marked with *)

136.	PDS	
137.	PERCAM	
138.	PET	*
139.	PFORT	*
140.	POD	
141.	PORTABLE FORTRAN MUTATION	*
142.	PPE	
143.	PREF HDR GEN	
144.	PROGCOMPANAL	
145.	PROGLOOK	
146.	PRONET	
147.	PRUFSTAND	
148.	PSL	
149.	PWB FOR VAX/VMS	
150.	QCM	
151.	QUICK-DRAW	
152.	RA	
153.	RADC/FCA	
154.	REFER	
155.	REFLECT II	
156.	REFTRAN	
157.	RISOS TOOLS	
158.	RTT	
159.	RXVP80	*
160.	SADAT	*
161.	SALSIM	
162.	SARA	
163.	SARA-H	
164.	SARA-U	
165.	SARA-III	
166.	SARA-IV	
167.	SCAN/370	
168.	SCERT	
169.	SCG/DQM	
170.	SDL	
171.	SDP/MAYDA	
172.	SDVS	
173.	SEF	
174.	SELECT	*
175.	SETAR	
176.	SIGS	
177.	SLIM	
178.	SMOTL	*
179.	SMT	
180.	SNOOP	
181.	SOFTOOL80	

ALPHABETICAL LISTING OF CATALOGED TOOLS
(Tools described by data sheets marked with *)

182.	SPECL/DARS	
183.	SPELL	
184.	SPRINT	
185.	SREM	
186.	SREP	
187.	SRIMP	
188.	SSA	
189.	STAG/TEMS	
190.	STRUCT	
191.	STRUCTURE(S)	
192.	SUBCRS	
193.	SURVAYOR	
194.	SYDIM	
195.	SYDOC	
196.	SYMCRS	
197.	SYSTEM MONITOR	
198.	SYSXREF	
199.	TAFIRM	
200.	TPAS/AM	
201.	TATTLE	
202.	TDBCOMP	*
203.	TCAT	
204.	TDEM	
205.	TEC/1	*
206.	TEST MANAGER	
207.	TEST PREDICTOR	
208.	TEVERE-1	*
209.	TFA	
210.	THE ENGINE	
211.	TIMECS	
212.	TOOLPAK	
213.	TPT	
214.	TRAILBLAZER	
215.	TSA/PPE	
216.	UCA	
217.	VIRTUAL OS	
218.	XPEDITER	

Note: The following data sheets were provided by the developers of the indicated tool. Although these responses were solicited by the contractor, no attempt has been made to assess the accuracy of the information.

TESTING TOOL DATA SHEET

Date:
Acronym: AMPIC
Title: AMPIC
Classification (all applicable categories): Symbolic Evaluator
Features: Symbolic execution, path predicate calculation, global cross-reference, structured and unstructured flow charts.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): SNOBOL
Implementation Hardware: IBM 360/370
OS (other software required):
Target Languages (of the tested module): WSC FORTRAN, ASSEMBLY (WSC, LITTON L4516D, SKC-2070)
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
[illegible] ($):
Developer: LOGICON, INC.
Contact (name, address, and telephone no.):

Mike Ikezawa
Logicon, Inc.
P. O. Box 471
San Pedro, CA 90733
(213) 831-0611

Tool summary: AMPIC is a program that structures, translates and symbolically executes other programs, written in higher order language (currently implemented for WSC Fortran) or assembly language (WSC, Litton L4516D, SKC-2070, etc.). A complete AMPIC run on a given input module consists of four AMPIC phases: 1) the module is segmented into code groups that can be treated as individual elements (nodes) of a flowchart; 2) a "structured" flowchart is created; 3) the input module is translated into methematical-type statements; 4) the input/output functional expressions for the entire paths through the input module are provided and symbolically executed.
Performance and limitations: Primarily an experimental facility
Documentation (type of available documentation): User's guide
References:

Applied Systems Design Section, TRW Defense and Space Systems Group
Software Tools: Catalogue and Recommendations
TRW Automated Software Tools Series, January 1979, U.S. Army TB 22-18

STEP - State-of-the-Art Overview

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Software Research Associates
Software Engineering Automated Tools Index
P. O. Box 2432, San Francisco, CA 94126

M. A. Ikezawa.
AMPIC for the Non-Programmer.
Logicon, Inc., Report R-CSS-77004, May 9, 1977.

TESTING TOOL DATA SHEET

Date:
Acronym: ATDG
Title: AUTOMATED TEST DATA GENERATOR
Classification (all applicable categories): Test Data Generator
Features: Test data generation, path structure analysis, anomaly detection, variable analysis.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Fortran
Implementation Hardware: UNIVAC 1110
OS (other software required):
Target Languages (of the tested module):
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: TRW for NASA in Houston
Contact (name, address, and telephone no.):

Dr. Barry Boehm
TRW Systems, Inc.
1 Space Park
Redondo Beach, CA 90278
(213)535-2184

Tool summary: The ATDG is an experimental interactive tool with two different functions: The test data generation (TDG) function provides automated support to program testing at the unit level (i.e., a single sub-route, function or main program) by identification of effective test case paths and the data constraints which must be satisfied to execute these paths; the static error analysis (SEA) function provides a diagnostic capability to supplement the error detection functions of conventional Fortran compilers by identification of path-dependent errors (e.g., uninitialized variables, infinite loops, unreachable code). These two functions are performed by analyzing a logic network of the software element using the principles of directed graph theory and dynamic programming. A network is constructed by defining a software element in terms of segments (logic blocks of Fortran statements that can be addressed), and by identifying the transfers and connective properties between these segments.
Performance and limitations: Experimental tool
Documentation (type of available documentation): User information note

References:

L. G. Stucki, et. al.
Methodology for Producing Reliable Software.
McDonnell douglas Astronautics Company, NASA CR 144769, March 76,
Two Volumes

TRW (Catalog)
Software Tools Catalogue and Recommendations
TRW, Defense and Space Systems Group, January 1979.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Software Research Associates
Software Engineering Automated Tools Index
P. O. Box 2432, San Francisco, CA 94126

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
February 1980.

TESTING TOOL DATA SHEET

Date: 10/11/82
Acronym: ATTEST
Title: AUTOMATIC TEST ENHANCEMENT SYSTEM
Classification (all applicable categories): Test Data Generator,
Symbolic Evaluator,
Data Flow Analyzer
Features: Symbolic evaluation, test data generation, data flow analysis, automatic path selection, constraint simplification
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Fortran 77
Implementation Hardware: VAX
OS (other software required): VMS
Target Languages (of the tested module): Fortran 66
Tool portable (yes, no): Yes (requires VM)
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no): Yes
Restrictions (copy rights, licenses, etc.): None
Tool supported (yes, no): No
Cost ($): None
Developer: Software Development Laboratory, U. of MA.
Contact (name, address, and telephone no.):
Lori A. Clarke
Dept. of Computer & Information Science
University of Massachusetts
Amherst, MA 01003
Tool summary: ATTEST is a test data generation system that analyzes programs written in FORTRAN 66. It contains the following components:

Data Flow Analyzer (DAVE) - Preprocesses the source statements into internal tables of information used by the other components. Produces a data flow report.
Path Selector - Path selection can be done manually, either interactively or statically, or automatically. The automatic path selection feature selects paths to satisfy user specified criteria.
Symbolic Evaluator - Creates symbolic representation of each selected path's domain (path condition) and computations.
Simplifier - Simplifies each constraint in the path condition.
Constraint Manager - Performs simple reductions on the constraints, handles complex constraints such as OR conditions, and when necessary calls a linear programming system to solve the path condition.

Performance and limitations: Not of production quality; unsupported
Documentation (type of available documentation): Limited

References:

W. Miller and D. L. Spooner.
Automatic Generation of Floating-Point Test Data.
IEEE Transaction on Softwaare Engineering, Vol. SE-2, March 1976, pp 223-226.

Lori A. Clarke and N. R. Ogden.
Top-Down Testing with Symbolic Execution.
DIGEST Workshop on Software Testing and Test Documentation, Ft. Lauderdale, Florida, Dec. 1978, pp. 191-196.

Automatic Test Data Selection Techniques.
Infotech State of the Art Report, Software Testing, Vol. 2, Infotech Internationl Ltd., September 1978, pp. 43-65.

Lori A. Clarke and J. Woods.
Program Testing Using Symbolic Execution.
Proceedings of the Software Specification and Testing Technology Conference, Washington, DC, April 1978, pp. 124-144.

Lori A. Clarke and Paul Abrahams.
Compile-Time Analysis of Data List-Format List Correspondences.
IEEE Transactions on Software Engineering, Vol. SE-5,6, November 1979, pp. 612-617.

Lori A. Clarke and D. J. Richardson.
Symbolic Evaluation Methods - Implementations and Applications.
Computer Program Testing. (B. Chandrasekaran and S. Radicchi, editors).
North Holland Publishing Company, 1981, pp. 65-102.

TESTING TOOL DATA SHEET

Date:
Acronym: AUT
Title: AUTOMATED UNIT TEST
Classification (all applicable categories): Test Driver
Features: Regression testing, simulation of test environment
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool):
Implementation Hardware: IBM 360/370
OS (other software required):
Target Languages (of the tested module): BAL, PL/q, COBOL
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($): 1,200
Developer: IBM
Contact (name, address, and telephone no.):

IBM Corporation
Data Processing Division
1133 Westchester Avenue
White Plains, NY 10604

Tool summary: AUT is a test harness system for BAL, PL/1 or COBOL environments. AUT is a productivity aid used to drive test cases through a unit of code for internal interface testing, monitor execution of the test cases, verify the performance of the test cases, and provide diagnostic information about discrepancies. Also provides capability to simulate uncoded or unfinished units of code or entire modules while driving test cases. IBM is publically silent on level of use internally, externally.
Performance and limitations:
Documentation (type of available documentation):
References:

IBM Documentation: Installed User Program, Automated Unit Test (AUT), For TSO and BATCH OS/VS, Program Description/Operations Manual, Program Number 5796-PEC, Manual SH20-1663-0, August 1975.

TESTING TOOL DATA SHEET

Date:
Acronym: AUTORETEST
Title: AUTORETEST
Classification (all applicable categories): Test Driver, Comparator
Features: Test data management, regression testing, automated comparison between selected old and new test parameters
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN IV, ASSEMBLY
Implementation Hardware: IBM 360/370
OS (other software required):
Target Languages (of the tested module):
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: TRW, Defense systems Software Department
Contact (name, address, and telephone no.):

Clarke Lucas
TRW, Defense Systems Dept.
One Space Park
Redondo Beach, CA 90278
(213)535-0426

Tool summary: The principal application of the AUTORETEST program is the automation of user software test results revalidation. The system provides an automated comparison between selected old and new test parameters, thereby allowing invaluable documentation of the test cases. This system also provides a flexibility in that a tolerance criterion may be assigned to each comparison and thereby suppress insignificant differences. This is similar to the driver tool available for the CDC computers.

Performance and limitations:
Documentation (type of available documentation): Development Specification, Programmer's Guide
References:

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Applied System Design Section, TRW Defense and Space Systems Group.
Software Tools: Catalogue and Recommendations.
TRW Automated Software Tools Series, January 1979.
U.S. Army TB22-18.

TESTING TOOL DATA SHEET

Date:
Acronym: CASEGEN
Title: CASEGEN
Classification (all applicable categories): Test Data Generator, Symbolic Evaluator
Features: Path generation, path constraint generation, automatic test data generation
Stage of Development (concept, design, implemented): Implemented as a prototype system part of FACES
Implementation Language (used to write the tool): Fortran
Implementation Hardware:
OS (other software required):
Target Languages (of the tested module): Fortran
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer:
Contact (name, address, and telephone no.):
Tool summary: Capture/MVS processes SMF/RMF records produced by OS/MVS to develop information for determining capture ratios in CPU sizing studies, for identifying high-overhead areas for tuning applications, and for developing representative baselines for capacity planning studies using either benchmarks or models. The package separates total processing activity during any user-selected interval into distinct workloads representing batch, TSO, IMS CICS, and other categories. Reports produced by capture/MVS contain such information as (1) overhead loads on CPU's, I/O devices, and channels for such system functions as paging, swapping and I/O interrupt handling; (2) per-work-load breakdowns of overhead time for batch, TSO, and other work-load categories; (3) capture ratios for each workload with options for including telecommunications overhead, spooling, and thelike; (4) activity profiles for each workload that indicate the total service time per transaction at each device and processor.
Performance and limitations: The system has problems in determining the number of loop iterations and user assistance is sometimes necessary
Documentation (type of available documentation):
References:

C. V. Ramamoorthy.
Techniques for Automated Test Data Generation.
Proceedings Ninth Asilomar Conference on Circuits, Systems, and Computers, November 1975.

C. V. Ramamoorthy, S. F. Ho, and W. T. Chen.
On the Automated Generation of Program Test Data.
IEEE Transaction on Software Engineering, SE-2(3), pp. 215-222, 1976.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

TESTING TOOL DATA SHEET

Date: 1982
Acronym: COBOL/DV
Title: COBOL/DV
Classification (all applicable categories): Test Data Generator
Features: Documentation aid, test data generation, run-time debugging aid, I/O specification analysis, tracing
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): COBOL
Implementation Hardware:
OS (other software required):
Target Languages (of the tested module): COBOL
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no): Yes (Applied Data Research)
Cost ($):
Developer:
Contact (name, address, and telephone no.):

Applied Data Research
Route 206 and Orchard Road, CN-8
Princeton, NJ 08540
(201)874-9100

Tool summary: Programmings aids: High-level functional verbs - 1) data manipulation 2) file handling, 3) table handling, 4) report writing, 5) documentation; COBOL short forms-reserved words, phases and clauses; data-name prefixer: Readable alternative to dataname qualification. Test data generator: Uses Cobol file and data descriptions - generates test data in parallel with program development; complete flexibility over data generation - 1) fields generated as constants, computed, random printable, 2) volume of test data under program control; regeneration of test data following maintenance - data generation parameters remain in the program as commentary. Run-time debugging aid: - abnormal termination analysis and reporting - multiple abends can be trapped, analyzed, located and reported during a single test; program activity display - input, intermediate results and output contents displayed in order of test execution.
Performance and limitations:
Documentation (type of available documentation):
References:

W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, Verification, and Testing of Computer Software, NBS Special Publication 500-75
National Bureau of Standards, pp. 32-35.

Applied Data Research Product Description, "COBOL/DV,"
Route 206 and Orchard Road, CN-8
Princeton, NJ 08540
(201) 874-9100

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

TESTING TOOL DATA SHEET

Date: 7/29/82
Acronym:
Title: COBOL OPTIMIZATION INSTRUMENTERS
Classification (all applicable categories): Instrumenter
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.
OS (other software required):
Target Languages (of the tested module): COBOL
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): Depends on system
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Licensing agreement
Tool supported (yes, no): Yes
Cost($): $7,000, includes one year maintenance; maintenance cost: 1/6th of original price
Developer: Softool Corporation
Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: There are three tools that automatically generate program, paragraph, and statement-level execution-time profiles (i.e., reports) of programs. The profiles quantify optimization efforts in detail. They show absolute and relative execution times for pro- grams, paragraphs, and statements, as well as frequency counts and optimization indices. INSTRUMENTER I operates at the program level. INSTRUMENTER II operates at the paragraph level, and INSTRUMENTER III operates at the statement level. These tools require no modification of any compiler or application program. They simply accept as input, source programs and test data, and output clear profiles. These tools permit top-down optimization in a natural manner. They possess a strong management orientation and can have much impact in properly focusing optimization efforts. They serve as an excellent quality assurance facility which allows management to set, facilitate, and enforce optimiza- tion standards. The OPTIMIZATION INSTRUMENTERS are members of SOFTOOL, an integrated set of tools marketed by Softool Corp.

Performance and limitations:

Documentation (type of available documentation): Interactive tutorials, manuals

References: Product description from Softool Corp.

TESTING TOOL DATA SHEET

Date: 7/29/82

Acronym:

Title: COBOL TESTING INSTRUMENTERS

Classification (all applicable categories): Instrumenters

Features: There are three tools that automatically generate program, paragraph, and statement-level execution-time profiles (i.e., reports) of programs. The profiles quantify test coverage and test effectiveness in detail. INSTRUMENTER I operates at the program level. INSTRUMENTER II operates at the paragraph level, and INSTRUMENTER III operates at the statement level. These tools require no modification of any compiler or application program. They simply accept as input source programs and test data, and output clear profiles. They permit top-down testing in a natural manner. These products possess a strong management orientation and can have much impact on minimizing the cost of testing. They serve as an excellent quality assurance facility which allows management to set, facilitate, and enforce testing standards. The TESTING INSTRUMENTERS are members of SOFTOOL, an integrated set of tools marketed by Softool Corporation.

Stage of Development (concept, design, implemented): Implemented

Implementation Language (used to write the tool): FORTRAN

Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.

OS (other software required):

Target Languages (of the tested module): COBOL

Tool portable (yes, no): Yes

Size (no. of source statements, memory size, etc.): Depends on system

Tool available (yes, no): Yes   Public domain (yes, no): No

Restrictions (copy rights, licenses, etc.): Licensing agreement

Tool supported (yes, no): Yes

Cost($): $7,000, includes one year maintenance; maintenance cost: 1/6th of original price

Developer: Softool Corporation

Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: SAME AS "FEATURES" LISTED ABOVE

Performance and limitations:

Documentation (type of available documentation): Interactive tutorials, manuals

References: Product description from Softool Corp.

TESTING TOOL DATA SHEET

Date: 7/29/82
Acronym:
Title: COBOL TRACING INSTRUMENTERS
Classification (all applicable categories): Instrumenter
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.
OS (other software required):
Target Languages (of the tested module): COBOL
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): Depends on system
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Licensing agreement
Tool supported (yes, no): Yes
Cost($): $7,000, includes one year maintenance; maintenance cost: 1/6th of original price
Developer: Softool Corporation
Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: There are three tools that automatically document the path of program, paragraph, statement control flow from program, paragraph, statement to program, paragraph, statement. INSTRUMENTER I operates at the program level. INSTRUMENTER II operates at the paragraph level, and INSTRUMENTER III operates at the statement level. These products offer the software professional a flexible, consistent and easy to use tracing facility. These tools require no modification of any compiler or application program. They simply accept as input source programs and test data, and output clear trace documentation (i.e., profiles) which is formatted and indented to facilitate understanding. They permit top-down tracing in a natural manner. The TRACING INSTRUMENTERS are members of SOFTOOL, an integrated set of tools marketed by Softool Corporation.

Performance and limitations:
Documentation (type of available documentation): Interactive tutorials, manuals
References: Product description from Softool Corp.

TESTING TOOL DATA SHEET

Date: 7/28/82
Acronym: COMMAP
Title: COMMON BLOCK MAP
Classification (all applicable categories): Static Analyzer
Features: A set of four reports containing different information about the static use of variables within the COMMON blocks. The user selects which of the reports to display or print.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN-77
Implementation Hardware: CDC
OS (other software required):
Target Languages (of the tested module): FORTRAN-66 or FORTRAN-77
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 3,000 source lines
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Non-exclusive license
Tool supported (yes, no): Yes
Cost ($):
Developer: Boeing Computer Services
Contact (name, address, and telephone no.):

Dr. Leon Stucki
P. O. Box 24346 - Mail Stop 9C-71
Seattle, WA 98124
(206)575-5118

Tool summary: COMMAP is a static analyzer for FORTRAN programs. Operating on existing source code, it produces a matrix cross-referencing variables in common blocks versus the subroutines that use them. The matrix specifies whether a variable is referenced or defined within a subroutine. It also analyzes the information in the matrix and reports on potential errors in the use of the variable (for example, variables which are referenced, but never defined).
Performance and limitations: COMMAP is currently available only on CDC (EKS).
Documentation (type of available documentation): Using ARGUS on EKSII Module 5
References:

TESTING TOOL DATA SHEET

Date:
Acronym: DATAMACS
Title: DATAMACS
Classification (all applicable categories): A Software Management, Control and Maintenance Tool, Test Data Generator, I/O Specification Analyzer
Features: Test file generation, I/O specification analysis, regression testing, file structure testing
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): BAL
Implementation Hardware: IBM 360/370 environment and compatible main frames
OS (other software required):
Target Languages (of the tested module): COBOL
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($): 16,000
Developer:
Contact (name, address, and telephone no.):

Technical/Marketing Representative
Management and Computer Services, Inc.
Great Valley Corporate Center
Valley Forge, PA 19482

Tool summary: DATAMACS is a highly flexible test data generator for COBOL programs. It generates all types of files (including VSAM and databases), creates hierarchical record structures, and changes field values automatically. It can be used in a load-and-go or stand-alone environment. DATAMACS input basically consists of a group of instructions inserted in a complete or partial COBOL program deck or library module. Data is created using both the test data prarmeters and information from the select statement and file definition. Options are available for retrieving data from live files, assigning check digits to selected fields, and interfacing with IMS, TOTAL, and IDMS database management systems. IDMS allows users of Cullinane Corporation's IDMS database management system to modify, unlead, or reload existing databases. It also provides archival storage and allows IDSMS users to utilize selective extractions of files rather than live databases. Maintenance included in lease/purchase price for first year; thereafter 15% of existing purchase price at time of renewal. One-half day's training free with purchase.
Performance and limitations:

Documentation (type of available documentation):
References:

Applied Systems Design Section, TRW Defense and Space Systems Group
Software Tools: Catalogue and Recommendations
TRW Automated Software Tools Series, January 1979, U.S. Army TB 22-18

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Software Research Associates
Software Engineering Automated Tools Index
P. O. Box 2432, San Francisco, CA 94126

TESTING TOOL DATA SHEET

Date:
Acronym: DAVE
Title: DOCUMENTATION, ANALYSIS, VALIDATION, and ERROR DETECTION
Classification (all applicable categories): Static Analyzer
Features: Diagnostics, Data Flow Analysis, Interface Analysis, Cross Reference, Standard Enforcer, Documentation Aid.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN-66
Implementation Hardware: Available for CDC 6400, IBM, UNIVAC, DEC 11/780 and Others.
OS (other software required):
Target Languages (of the tested module): ANSI FORTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 20,000 Source Statements; 50,000 words of CDC 6400
Tool available (yes, no): Yes   Public domain (yes, no): Yes
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($): 250
Developer: University of Colorado at Boulder
Contact (name, address, and telephone no.):

Leon Osterweil
University of Colorado
Dept. of Computer Science
Boulder, CO 80309
(303)492-7514

Tool summary: DAVE is one of the earlier static analyzers. The system provides good documentation, reliability, and ease of use at reasonable cost of operation. It uses data flow analysis to detect program anomolies such as references to undefined variables, unreferenced variable definitions, uninitialized variables (local or common), and interface checking.
Performance and limitations:
Documentation (type of available documentation):
References:

J. C. Browne and David B. Johnson.
FAST: A Second Generation Program Analysis.
Proceedings of the 3rd International Conference of Software Engineering, March 10-11, 1978, Atlanta, IEEE, pp 142-148.

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Leon J. Osterweil and Lloyd D. Fosdick.
DAVE - A Validation Error Detection and Documentation System for Fortran Programs.
Software Practice and Experience, Oct.-Dec. 1976, pp 473-486.

Software Research Associates.
Software Engineering Automated Tools Index, P. O. Box 2432, San Francisco, CA 94126, Tel. (415) 957-1441.

TESTING TOOL DATA SHEET

Date:
Acronym: DIFFS(TM)
Title: DIFFS(TM)
Classification (all applicable categories): File Comparator
Features: File comparison
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): COBOL, SCOBOL
Implementation Hardware:
OS (other software required):
Target Languages (of the tested module):
Tool portable (yes, no): Yes (distributed in source code form)
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Marketed Product
Tool supported (yes, no): Yes (Software Consulting Services)
Cost ($): 500 (perpetual lease)
Developer:
Contact (name, address, and telephone no.):

Ms. Martha J. Cichelli
Software Consulting Services
901 Whittier Drive
Allentown, PA 18103
(215)797-9690

Tool summary: DIFFS is a software productivity aid for programmers and auditors which compares two files and shows their differences. Should either file contain extra records, DIFFS searches for the point where the files match again, displays the extra records, and continues comparing. DIFFS' user selectable options simplify difficult comparison problems. For program files, sequence number fields and leading and trailing blanks can be ignored. For report and data files, selected column ranges can be compared or ignored. Multiple blanks can be treated as one blank for text file comparisons. Differences in files can be printed in either character format or in hexadecimal.

Performance and limitations:
Documentation (type of available documentation): User's manual
References:

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Software Research Associates
Software Engineering Automated Tools Index
P. O. Box 2432, San Francisco, CA 94126

D. J. Reifer and H. A. Montgomery.
SEATECS Software Tool Survey, RCI-TR-008
Reifer Consultants, Inc. 810330.

TESTING TOOL DATA SHEET

Date:
Acronym: DISSECT
Title:
Classification (all applicable categories): Symbolic Evaluator
Features: Symbolic execution, static analysis, assertion checking, path structure analysis (paths are selected by the user), documentation
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): LISP
Implementation Hardware: PDP-10 LISP SYSTEM
OS (other software required):
Target Languages (of the tested module): ANSI FORTRAN
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.): 4100 lines of LISP source code, requires at least 70K 36-bit words
Tool available (yes, no): Yes Public domain (yes, no): Yes
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer:
Contact (name, address, and telephone no.):
Tool summary: The DISSECT system is a symbolic evaluation tool which provides a command language to specify the path selection, input values, and set of output to be generated. Complex programs can be divided into segments and analyzed using separate cases. DISSECT analyzes an ANSI Fortran program to determine computations carried out along the selected paths, predicate constraints of each executed, and the symbolic values of the output variables. Since the symbolically evaluated predicate for a path describes the set of all input values which cause the paths to be executed, the output can be used as a guideline for the manual preparation of test data.
Performance and limitations: Howden found that the interactive path selection process was not satisfying because the choices must be made more carefully and systematically than as usually possible in interactive mode. DISSECT when combined with other techniques found 3-4% more error than the combined use of all techniques without DISSECT. It was 10-20% more effective than structured alone and the automatic test data generator was not useful in any of the six programs tested. The evaluation indicated that no single program analysis technique should be used to the exclusion of all others. Stucki rated the tool to be costly to run and the ease of use was on the average. Automatic test data generation was not planned for inclusion in DISSECT because of the unsuccessful development.
Documentation (type of available documentation):

References:

W. E. Howden
DISSECT - A Symbolic Evaluation and Program Testing System.
IEEE Transaction on Software Engineering, Vol. SE-4(1), January 1978, pp. 70-73.

L. G. Stucki, et al.
Methodology for Producing Reliable Software.
McDonnell Douglas Astronautics Company, March 1976, NASA CR144769, Two Volumes

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center Feburary 1980.

TESTING TOOL DATA SHEET

Date: 7/25/82
Acronym: DOCUTOOL
Title: DOCUTOOL
Classification (all applicable categories): Static Analyzer, Automatic Code Documentor
Features: The DOCUTOOL produces a file with the preface inserted in the source as comments. This file may then be edited to add additional information about each variable. DOCUTOOL can be used in conjunction with the CDC UPDATE function.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Pascal
Implementation Hardware: CDC(EKS)
OS (other software required):
Target Languages (of the tested module): FORTRAN-66 or FORTRAN-77
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 3,800 lines source
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Non-exclusive license
Tool supported (yes, no): Yes
Cost ($):
Developer: Boeing Computer Services - SAMA Division
Contact (name, address, and telephone no.):

Dr. Leon Stucki
P. O. Box 24346 - Mail Stop 9C-71
Seattle, WA 98124
(206)575-5118

Tool summary: Docutool is a tool which produces a preface for each module in a FORTRAN program. The preface is produced directly from the source code. The preface contains information about the subroutines, parameters, global variables and local variables used by the module. The information is formatted according to a predefined template.
Performance and limitations: Docutool is currently available only on CDC (EKS).
Documentation (type of available documentation): Using ARGUS on EKSII Module 8
References:

TESTING TOOL DATA SHEET

Date: 7/25/82
Acronym: DYNA
Title: DYNAMIC ANALYZER FOR FORTRAN
Classification (all applicable categories): Dynamic Analyzer
Features: The reports include an Entry Summary, a Program Summary, Module Summaries and an Annotated Source Listing. The Entry Summary documents the number of times each module was called while executing the test data. The Program Summary and Module Summaries categorize the number of different types of statements in the program or module and the percentage of each which were executed. The Annotated Source Listing provides detailed information about the number of times each statement or branch was executed.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN 77
Implementation Hardware: CDC (EKS II), VAX (UNIX & VMS), IBM (MVS)
OS (other software required): A FORTRAN-77 Compiler
Target Languages (of the tested module): FORTRAN-66 or FORTRAN-77
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 5,600 source lines
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Non-exclusive licenses
Tool supported (yes, no): Yes
Cost ($):
Developer: Boeing Computer Services - SAMA Division
Contact (name, address, and telephone no.):

Dr. Leon Stucki
P. O. Box 24346 - Mail Stop 9C-71
Seattle, WA 98124
(206)575-5118

Tool summary: DYNA is a tool for FORTRAN Programs which allows the user to see the dynamic behavior of a module while it is executing on the users test data. No modifications to either the program or its test data are required. DYNA operates in three steps. During the preprocessing step, probes (Additional FORTRAN statements) are automatically inserted in the source code and the "instrumented source" code is compiled. During the execution step, counts are made of the number of times each statement is executed by the test data. The counts may be accumulated with the counts from previous executions, if desired. During the post processing step, the execution data is formatted into reports.
Performance and limitations: Instrumented version of source code increase by 10-25% depending on branching amount in original source (size and execution time).
Documentation (type of available documentation): Using Argua on EKSII Module 4. Dyna User's Manual.

TESTING TOOL DATA SHEET

Date:
Acronym: EFFIGY
Title: EFFIGY
Classification (all applicable categories): Symbolic Evaluator
Features: Interactive symbolic execution (with normal execution as a special case), assertion checking, proof of correctness, standard interactive debug tools (including trace, break points, and state saving)
Stage of Development (concept, design, implemented): Implemented 1973
Implementation Language (used to write the tool): PL/1
Implementation Hardware: Runs on CMS under VM/370 on IBM/370,Model 168
OS (other software required):
Target Languages (of the tested module): PL/1 (restricted to integer valued variables and one dimensional arrays)
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: IBM
Contact (name, address, and telephone no.):
Tool summary: The EFFIGY system is an interactive symbolic execution tool incorporating standard debug tools and expanded to include assertion checking, a simple program testing manager and a program verifier. Normal program execution is provided as a special case. EFFIGY accepts one statement at a time, building a symbolic execution tree that defines the paths through the program. A test manager is available for systematically exploring the alternatives presented in the symbolic execution tree. The program verifier generates verification conditions from user supplied assertions in conjunction with the symbolic execution.
Performance and limitations: The system is a research tool and is limited in practical use.
Documentation (type of available documentation):
References:

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

J. C. King.
A New Approach to Program Testing.
Proceedings of International Conference on Reliability Software,
Los Angeles, CA, April 1975, pp. 228-233.

TESTING TOOL DATA SHEET

Date:
Acronym: EXPEDITER
Title: EXPEDITER
Classification (all applicable categories): Test Driver
Features: Tracing, regression testing
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): BAL
Implementation Hardware:
OS (other software required):
Target Languages (of the tested module):
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 10K-40K core memory
Tool available (yes, no): Yes Public domain (yes, no):
Restrictions (copy rights, licenses, etc.): Availability according to Air Force Manual (AFM) 300-6, Paragraph 11-7A.
Tool supported (yes, no): Yes (RADC/ISIS)
Cost ($):
Developer: Application Development System, Inc.
Contact (name, address, and telephone no.):

Edward F. Harris
Application Development Systems, Inc.
1530 Meridian Avenue
San Jose, CA 95125
(408)264-2272

Tool summary: EXPEDITER provides facilities for unit testing of modules and program component testing in the development environment. It also includes features for problem isolation and verification of fixes in the maintenance context. No changes to source programs are required. It is responsible for improving productivity in a Cobol environment from 10 lines of procedures division code pr programmer pr day to 45 lines.

Performance and limitations:
Documentation (type of available documentation): User's Guide (125) Reference Card (4), SPF Tutorial, TSO Help, Programmed Instruction Course (200).
References:

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

TESTING TOOL DATA SHEET

Date:
Acronym: FACES
Title: FORTRAN AUTOMATED CODE EVALUATION SYSTEM
Classification (all applicable categories): Static Analyzer
Features: Data flow analysis, Diagnostics, Variables analyzer (intermodule), Interface checker, Standards enforcer, Reachability analyzer.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: Configuration available UNIVAC 1108, CDC 6400, IBM 360/65
OS (other software required):
Target Languages (of the tested module): ANSI Fortran
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): The front-end consists of 6,000 source statements, and the diagnotic routines consists of 1,500 source statements.
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: COSMIC, University of Georgia
Contact (name, address, and telephone no.):

Rex Walker
COSMIC
University of Georgia
Suite 112, Barrow Hall
Athens, GA 30602
(404)542-3265

Tool summary: FACES is developed to detect error prone constructs such as call to subroutine with constants as parameters, check type and dimension of variables in COMMON and subroutine parameters, redundant and unreachable code, loop construction and termination, code standards, and uninitialized local variables. The system is comprised of a preprocessor, a processor, and a report generator. Either unit modules or interrelated modules can be run as a data set of FACES. FACES is organized into a driver section with three subsystem components. The main driver is responsible for file manipulations and interpreting user commands. One of the components is called the Automatic Interrogation Routine (AIR). Its purpose is to examine tables generated by a front-end portion of FACES, and look for types of coding constructions selected by the user. If the specified constructions are found, diagnostic messages are recorded on the flag file. A report generator generates user reports. Area of coding that cannot be effectively evaluated are also reported to the user.

Performance and limitations: FACES provides only intra module initialization checking and variable traces.

Documentation (type of available documentation):

References:

C. V. Ramamoorthy.
Testing Large Software with Automated Software Evaluation Systems
IEEE Transaction on Software Engineering, March 1975, pp 46-58.

J. C. Browne and David B. Johnson.
FAST: A Second Generation Program Analysis.
Proceedings of the 3rd International Conference of Software Engineering, March 10-11, 1978, Atlanta, IEEE, pp 142-148.

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

TESTING TOOL DATA SHEET

Date:
Acronym: FADEBUG-I
Title: FACOM AUTOMATIC DEBUG
Classification (all applicable categories): Output Comparator,
Anomoly Detector
Features: Comparison, I/O specification analysis, debug aid
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Assembly
Implementation Hardware: FACOM 230-60
OS (other software required):
Target Languages (of the tested module):
Tcol portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no): Yes (Fugitsu Ltd.)
Cost ($):
Developer: Fujitsu Ltd.
Contact (name, address, and telephone no.):
Tool summary: FADEBUG-I has two primary functions: comparing the set of output data produced by a program with user-specified output data is identified as its most important function, and automatic isolation and definition of all possible execution paths from entry to exit in a program module. These capabilities aid in finding and removing program bugs. In the module test stage of program development the following difficulties are identified: (1) Examination and verification of output data from module test execution. (2) Examination of module processing paths for logical errors. (3) Evaluation of module logic paths for omissions. FADEBUG-I is designed to reduce or eliminate these difficulties through its test function or route definition function.
Performance and limitations:
Documentation (type of available documentation):
References:

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Itoh.
Fade-Bug-I, A new Tool for Program Debugging.
Proc. IEEE Symposium, Computer Softw. Reliability, pp. 38-43, 1977.

TESTING TOOL DATA SHEET

Date:
Acronym: FAST
Title: FORTRAN ANALYSIS SYSTEM
Classification (all applicable categories): Static Analyzer
Features: Interaction operation, data flow analysis, interface checking, consistency checking, error checking, command/query language.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: The system uses the commercially available data management system, System 2000 (MRI System Corp.) as its data handler and data correlator along with the FACES source program parser and the POBSW parser generator (Univ. of Texas, Austin).
OS (other software required):
Target Languages (of the tested module): Fortran
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer:
Contact (name, address, and telephone no.):
Prof. Jim Browne
Information Research Associates (IRA)
911 West 29th Street
Austin, TX
Tool summary: The fast system creates a data base of the attributes of modules, statements and names in a Fortran program and interactively processes a wide range of queries concerning these attributes. The fast data base is generated from the Fortran source program by using: (1) the FACES parser (2) a program to map the output of the parser onto system 2000 load string (3) the system 2000 data management system. The fast command/query language, which is used to query the data base, defines approximately 10 attributes of Fortran names and statements. These attributes can be combined in logical expressions to qualify or isolate very broad or very narrow program contexts. The command language interpreter was implemented through the use of the BOBSW parser generator.
Performance and limitations:
Documentation (type of available documentation):
References:
J. C. Browne and David B. Johnson.
FAST: A Second Generation Program Analysis.
Proceedings of the 3rd International Conference of Software Engineering, March 10-11, 1978, Atlanta, IEEE, pp 142-148.

STEP - State-of-the-Art Overview

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

TESTING TOOL DATA SHEET

Date: August 1982
Acronym: FAVS
Title: FORTRAN AUTOMATED VERIFICATION SYSTEM
Classification (all applicable categories): Source Program Analysis & Testing Static Analyzer, Self-metric Instrumenter, Coverage Analyzer, Documenter
Features: Run time analysis, subject, code input, FORTRAN, DMATRAN, transformation, translation, structure preprocessing, restructuring, instrumentation, formatting, machine output, source code output, FORTRAN, user output, diagnostics, user-oriented text, documentation, tables, static analysis, cross reference, type analysis, structure checking, dynamic analysis, coverage analysis, tuning.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): DMATRAN (a structured FORTRAN)
Implementation Hardware: Honeywell H6180; Univac 1100 Computer Systems
OS (other software required): ECOS and 05110
Target Languages (of the tested module): DMATRAN or FORTRAN V
Tool portable (yes, no): Yes, with minor modifications
Size (no. of source statements, memory size, etc.): 35,000 source statements; 52K core
Tool available (yes, no): Yes Public domain (yes, no): Yes, with approval
Restrictions (copy rights, licenses, etc.): Air force approval required for code release
Tool supported (yes, no): No
Cost ($): DMATRAN Precompiler: $650; FAVS: $850
Developer: General Research Corp., Santa Barbara CA
Contact (name, address, and telephone no.):
Frank S. LaMonica
RADC/COEE
Griffiss, AFB NY 13441
(315)330-3977

Tool summary: FAVS, an integrated collection of computer programs, was developed for the purpose of assuring that software systems written in FORTRAN are comprehensively tested. FAVS provides (1) static detection of unreachable statements, set/use errors, mode-conversion errors, and external reference errors, (2) a means of measuring the effectiveness of test cases by source code instrumentation, (3) assistance in the construction of test data that will thoroughly exercise the software, and (4) automated documentation. In order to aid in the production of application software that adheres to modern programming techniques, FAVS also provides for the translation from DMATRAN (a structured extension of FORTRAN) to FORTRAN and from FORTRAN to DMATRAN.

Performance and limitations: The FAVS itself is written in the DMATRAN Programming Language (a structured FORTRAN V). The DMATRAN Precompiler (in addition to the FORTRAN V Compiler) is needed to compile the FAVS source code.

Documentation (type of available documentation): DMARTRAN User's Guide; FAVS User's Manual; FAVS-Generated Documentation (eg. Module Hierarchy Inter-Relationshops, Set/Use Tables, Program Listings)

References: Documentation available through Federal Software Exchange Center, 5285 Port Royal Road, Springfield VA, 22161, (703) 487-4655. Order Number: DMATRAN Precompiler FSWEC-81/0002-1 cost: PC $22.50 MF $8.50 FAVS FSWEC-81/0003-1 cost: PC $22.50 MF $8.50.

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

M. Finfer, et. al.
Software Debugging Methodology, Final Technical Report,
NADC-TR-79-57, Three Volumes, April 1979.

TESTING TOOL DATA SHEET

Date:
Acronym: FCA
Title: FORTRAN CODE AUDITOR
Classification (all applicable categories): Source Program Analysis and Testing
Features: Subject, Code Input: FORTRAN Y Source Code
User Output: Diagnostics, Program Listings
Static Analysis: Auditing, Structure Checking
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN IV
Implementation Hardware: Honeywell H6180
OS (other software required): GCOS Operating System
Target Languages (of the tested module): FORTRAN Y (Honeywell Extended Compiler)
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 6300 Source Statements: 40K Core
Tool available (yes, no): Yes Public domain (yes, no): Yes - with approval
Restrictions (copy rights, licenses, etc.): Air Force approval required for Code Release
Tool supported (yes, no): No
Cost ($): Contact Federal Software Exchange Control, (703)756-6140
Developer: TRW
Contact (name, address, and telephone no.):

Frank S. Lamonica
RADC/COEE
Griffiss AFB, NY 13441
(314)330-3977

Tool summary: The FORTRAN Code Auditor, an automated test tool, is used for the cost effective enforcement of FORTRAN programming standards and conventions appropriate to the Air Force software environment. It does not modify code. Using predefined coding standards and conventions, it simply advises the user where these standards and conventions have not been adhered to. The major advantage of favoring an automated auditor over manual methods, besides cost effectiveness, is complete objectivity and unambiguity. The standards can be viewed as being coding enforcements in four areas: (1) Documentation Standards - Standards defining quantity and placement of commentary thus enhancing program readability and comprehension. (2) Format Standards - Standards identifying physical placement and grouping of code elements on the source code listing. (3) Design Standards - Standards limiting module size and placing restrictions on the use of certain instructions with the end of providing an optimization of code relative to execution time. (4) Structural Standards - Standards requiring the use of strict rules for the top-down design and implementation of a system of programs and the requirement that the components adhere to a heirarchical form as much as possible.

Performance and limitations: Target module, to be analyzed by Code Auditor, must be written in FORTRAN Y Programming Language.

Documentation (type of available documentation): User's Manual; Program Maintenance Manual

References:

Manuals in National Technical Information Service Inventory

User's Manual: Reference RADC-TR-76-395, Volume I; Accession #AD A035-778

Maintenance Manual: Reference RADC-TR-76-395, Volume II, Accession #AD A035-914

TESTING TOOL DATA SHEET

Date:
Acronym: FORAN
Title: FORTRAN ANALYZER PROGRAM
Classification (all applicable categories): Static Analyzer
Features: Interface analysis, cross reference, consistency checking, error checking, variable analysis.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Fortran
Implementation Hardware: CDC 6X00/7X00
OS (other software required):
Target Languages (of the tested module): Any Fortran dialect
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no): Yes
Cost ($):
Developer:
Contact (name, address, and telephone no.):

U.S. Army Advanced Research Center
Huntsville, Alabama

Tool summary: FORAN performs static analysis on source code written in any dialect of Fortran. Usage of program labels, tags, data variables, constants, subroutines, and other program elements are analyzed for a main program and its related subroutine components. Each item name is listed, showing the statement numbers where the item is referenced and how it is referenced (assigned, used, input, output, subroutine call, etc.). FORAN also identifies symbols defined but not used, discrepancies in variable type and dimension, and number and type of parameters in functions and subroutines. Syntax errors are flagged during the analysis. FORAN's primary use is to determine possible computation of logic errors from the static analysis of data usage. It is also valuable in analyzing the effect of a program modification on data usage.

Performance and limitations: The FORAN analysis is limited to 4095 data items and a total of 24,000 unique references for all named items. Finfer, et al report that it is easy to use and its output contains more information and is easier to read than a compiler's symbolic reference map.

Documentation (type of available documentation):

References:

M. Finfer.
Software Debugging Methodology.
Final Technical Report, RADC-TR-79-57, April 79, Three Volumes.

STEP - State-of-the-Art Overview

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

TESTING TOOL DATA SHEET

Date: 7/29/82
Acronym:
Title: FORTRAN AUDITOR
Classification (all applicable categories): Code Auditor, Static Analyzer
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.
OS (other software required):
Target Languages (of the tested module): FORTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): Depends on system
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Licensing agreement
Tool supported (yes, no): Yes
Cost($): $16,000, includes one year maintenance; maintenance cost: 1/6th of original price
Developer: Softool Corporation
Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: The AUDITOR automatically audits FORTRAN programs for compliance with user programming standards, poor programming practices, nonportable code and deviations from the American National Standards Institute (ANSI) definition of the FORTRAN language. This tool also generates automatic program documentation. In addition, this product is a powerful error detector which typically detects many errors that escape commercial compilers. This tool requires no modification of any compiler or application program. It simply accepts as input FORTRAN source programs and outputs various reports. An option is available that allows FORTRAN programs for 16-bit word machines to be checked on machines with 32-bit words. This product possesses a strong management orientation and serves as an excellent quality assurance tool since it presents simple summaries at the end of its clear and detailed output. This tool is a member of SOFTOOL, an integrated set of tools marketed by Softool Corporation.

Performance and limitations:
Documentation (type of available documentation): Interactive tutorials, manuals
References: Product description from Softool Corp.

TESTING TOOL DATA SHEET

Date: 7/29/82
Acronym:
Title: FORTRAN OPTIMIZATION INSTRUMENTERS
Classification (all applicable categories): Instrumenter
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.
OS (other software required):
Target Languages (of the tested module): FORTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): Depends on system
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Licensing agreement
Tool supported (yes, no): Yes
Cost($): $7,000, includes one year maintenance; maintenance cost: 1/6th of original price
Developer: Softool Corporation
Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: There are two tools that automatically generate module and statement level execution-time profiles (i.e., reports) of program. The profiles quantify optimization efforts in detail. They show absolute and relative execution times for subsystems, modules and statements as well as frequency counts and optimization indices. INSTRUMENTER I operates at the routine level. INSTRUMENTER II operates at the statement level. These tools require no modification of any compiler or application program. They simply accept as input source programs and test data, and output clear profiles. These tools permit top-down optimization in a natural manner. They possess a strong management orientation and can have much impact in properly focusing optimization efforts. They serve as an excellent quality assurance facility which allows management to set, facilitate and enforce optimization standards. The OPTIMIZATION INSTRUMENTERS are members of SOFTOOL, an integrated set of tools marketed by Softool Corp.
Performance and limitations:
Documentation (type of available documentation): Interactive tutorials, manuals
References: Product description from Softool Corp.

TESTING TOOL DATA SHEET

Date: 7/29/82
Acronym:
Title: FORTRAN TESTING INSTRUMENTERS
Classification (all applicable categories): Instrumenter
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.
OS (other software required):
Target Languages (of the tested module): FORTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): Depends on system
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Licensing agreement
Tool supported (yes, no): Yes
Cost($): $7,000, includes one year maintenance; maintenance cost: 1/6th of original price
Developer: Softool Corporation
Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: There are two tools that automatically generate module and statement level execution-time profiles (i.e., reports) of programs. The profiles quantify test coverage and test effectiveness in detail. INSTRUMENTER I operates at the routine level. INSTRUMENTER II operates at the statement level. These tools require no modification of any compiler or application program. They simply accept as input source programs and test data, and output clear profiles. They permit top-down testing in a natural manner. These products possess a strong management orientation and can have much impact on minimizing the cost of testing. They serve as an excellent quality assurance facility which allows management to set, facilitate and enforce testing standards. The TESTING INSTRUMENTERS are members of SOFTOOL, an integrated set of tools marketed by Softool Corporation.

Performance and limitations:
Documentation (type of available documentation): Interactive tutorials, manuals
References: Product description from Softool Corp.

TESTING TOOL DATA SHEET

Date: 7/29/82
Acronym:
Title: FORTRAN TRACING INSTRUMENTERS
Classification (all applicable categories): Instrumenter
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.
OS (other software required):
Target Languages (of the tested module): FORTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): Depends on system
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Licensing agreement
Tool supported (yes, no): Yes
Cost($): $7,000, includes one year maintenance; maintenance cost: 1/6th of original price
Developer: Softool Corporation
Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: There are two tools that automatically document the path of program control flow from module (statement) to module (statement). INSTRUMENTER I operates at the routine level. INSTRUMENTER II operates at the statement level. These products offer the software professional a flexible, consistent and easy to use tracing facility. These tools require no modification of any compiler or application program. They simply accept as input source programs and test data, and output clear trace documentation (i.e., profiles) which is formatted and indented to facilitate understanding. They permit top-down tracing in a natural manner. The TRACING INSTRUMENTERS are members of SOFTOOL, an integrated set of tools marketed by Softool Corporation.
Performance and limitations:
Documentation (type of available documentation): Interactive tutorials, manuals
References: Product description from Softool Corp.

TESTING TOOL DATA SHEET

Date: 8/20/82
Acronym: GENTEXTS
Title: GENTEXTS
Classification (all applicable categories): Test Data Generator for Compilers
Features: A grammar, in affix form, is used as input to produce a program. The output program is systematically generated to test some particular aspects of a compiler.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): PL/1, PASCAL, SIMULA 67
Implementation Hardware: CIT-HB
OS (other software required): SIRIS
Target Languages (of the tested module):
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 800 P1/1 source lines
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no): Yes
Cost ($): 2,000
Developer: IRISA, Unoversity of Rennes, France
Contact (name, address, and telephone no.):

B. Houssais
IRISA, Campus de Beaulier
35042 Rennes
Cedex, France
Tel. 99.36.20.00
or
Jeff Rees
Intermetrics Inc.
Boston, MA, USA

Tool summary: Preparing tests for a compiler entails writing a large number of test programs of a relatively fixed structure. Such a test program can be described by a particular type of grammar, which is learned fairly quickly. These grammars, called "command grammars", are submitted to the generator, which derives the corresponding test program. the generator automatically transforms the command grammar into a test generator program. This program (in simula 67) is then compiled and executed, and produces the test cases described by the grammar. This output can then be used (after possible modifications) to test the target compiler.

Performance and limitations: A test set of 40,000 lines of ALGOL 68 programs have been produced for ALGOL 68 compilers. Input was about 50 grammars of total length of 3,000 lines. All programs were compiled and run. The number of errors discovered was significant.

STEP - State-of-the-Art Overview

Documentation (type of available documentation): Users's manual (French and English); Technical paper

References:

J. Andre, J. Duclov, P. Laforgue, H. Massie, and J. C. Rault.
Catalogue 1980 De Prototypes De Recherche En Logiciel
ADI (AGence De L'Informatique), CNRS, France, 801100

B. Houssais
Production Systematique De Tests.
(Thesis in French), Microfiche from INRIA, BP105, F78153,
Le Chesnay, France

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

TESTING TOOL DATA SHEET

Date: 7/29/82
Acronym:
Title: INTERFACE DOCUMENTER
Classification (all applicable categories): Static Analyzer
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: DEC, DG, IBM, GOULD-S.E.L.
OS (other software required):
Target Languages (of the tested module): FORTRAN, COBOL, any object Code
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): Depends on system
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Licensing agreement
Tool supported (yes, no): Yes
Cost($): $7,000, includes one year maintenance; maintenance cost: 1/6th of original price
Developer: Softool Corporation
Contact (name, address, and telephone no.):

Krisse Specht
349 S. Kellogg
Goleta, CA 93717
(805) 964-0560

Tool summary: This software product accepts as input a collection of object modules and automatically generates clear information indicating all interfaces between the object modules. It produces, for each module, an annotated list of the modules it references as well as a list of all the modules that reference it. External data items are also documented. This product is very easy to use. It accepts as inputs the same object modules that are normally presented to your linker (binder). Thus in order to generate the interface documentation you simply submit your inputs to the INTERFACE DOCUMENTER instead of the linker (binder). This tool is independent of the language in which the programs being documented are written. It will generate interface documentation for FORTRAN, COBOL, ASSEMBLER, etc. This produce is a member of SOFTOOL, an integrated set of tools marketed by Softool Corporation.
Performance and limitations:
Documentation (type of available documentation): Interactive tutorials, manuals

TESTING TOOL DATA SHEET

Date:
Acronym: JAVS
Title: JOVIAL AUTOMATED VERIFICATION SYSTEM
Classification (all applicable categories): Static Analyzer,
Instrumenter,
Coverage Analyzer,
Assertion Checker,
Automatic Documenter
Features: Test completion analysis, test data generation aid, path flow analysis, path structure analysis, reachability analysis, interface checking, assertion checking, automatic documentation, debug tools (graphic output, trace, cross-reference, dump, breakpoint)
Stage of Development (concept, design, implemented): Implemented 1975
Implementation Language (used to write the tool): JOVIAL J3
Implementation Hardware: HIS 6180, CDC 6400
OS (other software required): GCOS, WWMCCS, GOLETA operating systems
Target Languages (of the tested module): JOVIAL
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.): 53,000 words of primary memory on the HIS 6180 system
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: General Research Corp.
Contact (name, address, and telephone no.):

N. B. Brooks
General Research Corporation
5383 Hollister Avenue
Santa Barbara, CA 93111
(805)964-7724

Tool summary: JAVS is a workable, field-tested system for systematically and comprehensively software testing. It is a series of tools which provide a means of measuring the effectiveness of both individual and cumulative software test cases, a capability to facilitate the construction of test data that will thoroughly exercise the soft- ware, and an analysis of retesting requirements following software modification. The system performs static analysis on sucessfully compiled JOVIAL (J3) source modules. Up to 250 invokable modules and an unlimited number of JOVIAL statements can be analyzed in a single run. The analysis includes determination of program paths, inter- and intra- module relationships, unreachable modules, extensive cross reference of symbols and usage of program variables. In dynamic analysis, the system determines test coverage and identifies the unexercises paths. Execution analysis indicates which modules, decision paths, and statements have been exercised, including the frequency and the execution time spent in each module. JAVS also provides tracing capability, regression testing, assertion checking, and automatic program documentation.

Performance and limitations: Static analysis can be performed on JOVIAL (J3) source modules after a successful error-free compilation. Up to 250 invokalbe modules and an unlimited number of JOVIAL statements can be analyzed in a single process job. In general, the execution of a JAVS - instrumented program requires 1.5 times the execution time of an uninstrumented program and approximately twice the load core size. Finfer et. al. reports that JAVS is a powerful tool that provides the user a good deal of control over the amount and type of debugging information produced, but it does require the user to master a rich command language. Gannon recommends adding identification of uninitialized variables and physical-units consistency checking in static analysis, and automatic generation of certain type of assertions and coverage measurement of program functions in dynamic analysis. TRW researchers point out some of the JAVS system disadvantages: 1) The output is difficult for users to analyze because of its D-D path orientation. Manual correlation is required to interpret the results at any other working level, such as statement level, which may be more familiar to users. 2) The overhead caused by recording execution monitoring data on a mass storage trace file would be unacceptable for the instrumentation of an entire medium to large scale programs.

Documentation (type of available documentation): User's guide, reference manual

References:

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

M. Finfer, et. al.
Software Debugging Methodology, Final Technical Report,
NADC-TR-79-57, Three Volumes, April 1979.

Compendium of ADS Project Management Tools and Techniques.
Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

TRW Systems and Space Group.
NSW FEASIBILITY STUDY, Final Technical Report, RADC-TR-78-23,
Feburary 1978.

TESTING TOOL DATA SHEET

Date:
Acronym: JOYCE
Title: JOYCE
Classification (all applicable categories): Static Analyzer
Features: Path structure analysis, symbol cross reference, variable analyzer/interface checking, documentation aid.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Fortran
Implementation Hardware: CDC 6X00/7X00
OS (other software required):
Target Languages (of the tested module): Fortran
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes   Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no): Yes
Cost ($):
Developer: McDonnell
Contact (name, address, and telephone no.):

McDonnell Douglas Automation Company
P. O. Box 516
St. Louis, MO 63166

Tool summary: JOYCE is an automatic static analysis tool for Fortran programs. It accepts as primary input Fortran source decks in the form of card decks or CDC compile files. The source decks are edited and the edited information is combined to produce several combinations of descriptive reports. JOYCE compiles tables of symbols and cross references of symbol usage within each routine of a program. These symbols include Fortran variable names, the names of any reference function or module, any entry points, and all I/O file references. Flowlists are provided in the form of microfilm Fortran listings with all transfers indicated by arrows to the right of the statement text and all dd loops indicated by brackets to the left.

Performance and limitations: JOYCE was evaluated under contract to NASA Goddard Space Flight Center (GSFC) and was found to have low operating cost and quite easy to use. But only the configuration for CDC is available.

Documentation (type of available documentation):

References:

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Software Research Associates.
Software Engineering Automated Tools Index.
P. O. Box 2432, San Francisco, CA 94126, (415)957-1441.

L. G. Stucki.
Methodology for Producing Reliable Software.
McDonnell Douglas Astronautics Company, March 76, NASA CR 144769,
Two Volumes

TESTING TOOL DATA SHEET

Date:
Acronym: PACE
Title: PRODUCT ASSURANCE CONFIDENCE EVALUATOR
Classification (all applicable categories): Static Analyzer, Instrumenter, Test Completion Analyzer, Path Structure Analyzer, Coverage Analyzer
Features: Path flow analysis, instrumention, optimization aid, test case selection aid, regression testing, coverage anal ysis
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Fortran
Implementation Hardware: CDC 6500/7600, UNIVAC 1108
OS (other software required):
Target Languages (of the tested module):
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: TRW, SEID Software Product Assurance
Contact (name, address, and telephone no.):

Frank Ingrassia
TRW, SEID Software Product Assurance
One Space Park
Redondo Beach, CA 90278
(213)536-3140

Tool summary: PACE is a unique quality assurance tool to aid program developers and testers in the planning, execution and evaluation of both routine level and program level tests. The object of pace is to quantitatively assess how thoroughly and rigorously a program has been tested. PACE is the tool that is used to assure that every logical and arithmetic instruction of every branch be subjected to an execution test. Versions of the PACE system have been developed by TRW which provide special options dictated by a given user. These PACE versions are Nodal, Anode, and AVS/TDEM. A highly modular design approach was taken to reduce and isolate hardware/software dependent characteristics and assure easy implementation on a variety of computers. Input to PACE consists of the user's Fortran source code, and a PACE option care. Output from PACE can be varied by using the option card, but nominally includes: 1) a listing of the user's source code annotated with segment numbers, 2) a program structure summary.
Performance and limitations:
Documentation (type of available documentation): User's manual

References:

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Software Tools: Catalogue and Recommendations
TRW Automated Software Tools Series, Applied Systems Design
Section, TRW Defense and Space Systems Group, January 1979

TESTING TOOL DATA SHEET

Date:
Acronym: PET
Title: PROGRAM EVALUATOR AND TESTER
Classification (all applicable categories): Instrumenter,
Dynamic Assertion Processor,
Coverage Analyzer
Features: Instrumentation, diagnostics, static analysis, statistical analysis, profile generation, coverage analysis, assertion checking
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Fortran
Implementation Hardware: IBM, CDC, HONEYWELL, UNIVAC, CDC 6000
OS (other software required):
Target Languages (of the tested module): Fortran
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): For sale
Tool supported (yes, no): Yes (McDonnell-Douglas Corp.)
Cost ($):
Developer: McDonnell-Douglas Corp.
Contact (name, address, and telephone no.):

J. B. Churchwell
McDonnell-Douglas Corporation
5301 Bolsa Avenue
Huntington Beach, CA 92647
(714)896-4155

Tool summary: PET accepts Fortran programs as inputs and gathers and analyzes data in two general areas: 1) the syntactic profile of the source program showing the number of executable, nonexecutable, and comment statements, the number of call statements and total program branches, and the number of coding standard's violations, and 2) actual program performance statistics produced by the PET include: the number and percentage of those executable source statements actually executed; the number and percentage of those branches and call's actually taken or executed; the following specific data associated with each executable source statement: a detailed execution counts, detailed branch counts on all if and goto statements, and min/max data range values on assignment.

Performance and limitations: PET is an early self-metric tool. It was evaluated to be easy to use and the operating cost was not high. The system was recommended for use in situation where operating cost is not the major factor in selection. Gilb reports that PET was effective in coverage analysis and the increase in execution time resulting from the PET instrumentation varies from 25% to 150% depending on the options used.

Documentation (type of available documentation): User's manual , system description

References:

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Software Tools: Catalogue and Recommendations
TRW Automated Software Tools Series, Applied Systems Design
Section, TRW Defense and Space Systems Group, January 1979

L. G. Stucki.
A Prototype Automatic Program Testing Tool.
AFIPS Fall Joint Computer Conference, 721205.

T. Gilb.
Software Metrics.
Winthrop Publishers, Inc., Cambridge, MA, 1977, page 282.

L. G. Stucki, et.al.
Methodology for Producing Reliable Software, NASA CR 144769
McDonnell-Douglas Astronautics Company, March 1976, Two Volumes

L. G. Stucki and G. L. Foshee.
New Assertion Concepts for Self-Metric Software Validation.
Proceedings of IEEE Conference on Reliable Software, Los Angeles,
CA, April 1975, pages 59-65.

TESTING TOOL DATA SHEET

Date:
Acronym: PFORT
Title: PFORT VERIFIER
Classification (all applicable categories): Standard Enforcer
Features: Standard enforcer, documentation aid, interface checking, cross reference.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): Fortran
Implementation Hardware:
OS (other software required):
Target Languages (of the tested module): Fortran
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes Public domain (yes, no): Yes
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: Jet Propulsion Laboratory
Contact (name, address, and telephone no.):

Irma B. Biren
Bell Lab.
600 Mountain Avenue
Murray Hill, NJ 07974
(201)582-3000

Tool summary: PFORT is a portability checker for Fortran. It analyzes a Fortran Program and notes the occurrences of programming practices that are likely to be impediments to portability.
Performance and limitations:
Documentation (type of available documentation): User's Manual
References:

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
February 1980.

TESTING TOOL DATA SHEET

Date: 1/1/83
Acronym:
Title: PORTABLE FORTRAN MUTATION SYSTEM
Classification (all applicable categories): Automatic Mutation System
Features: Test harness and driver, computes mutation scores
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool):
Implementation Hardware:
OS (other software required):
Target Languages (of the tested module): FORTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes Public domain (yes, no): Yes
Restrictions (copy rights, licenses, etc.): Copyright
Tool supported (yes, no): No
Cost ($):
Developer: Dr. T. A. Budd
Contact (name, address, and telephone no.):

Prof. T. A. Budd
University of Arizona
Tucson, AZ 85721
(602)626-0111

Tool summary:
Performance and limitations:
Documentation (type of available documentation):

STEP - State-of-the-Art Overview

TESTING TOOL DATA SHEET

Date:

Acronym: RXVP-80

Title: RXVP-80

Classification (all applicable categories): Static Analyzer,
Test Driver,
Assertion Processor,
Instrumenter,
Coverage Analyzer

Features: Static analysis, coverage analysis, assertion checking, symbolic execution, instrumentation, interface analysis, cross reference, complexity measurement, ability to analyze very large source programs.

Stage of Development (concept, design, implemented): Implemented

Implementation Language (used to write the tool):

Implementation Hardware: CDC, IBM, UNIVAC

OS (other software required): Fortran compiler that is compatible with ANSI X 3.9-1966.

Target Languages (of the tested module): Fortran, IFTRAN (TM), or V-IFTRAN (TM)

Tool portable (yes, no): Yes

Size (no. of source statements, memory size, etc.): 50,000 32-bit memory words

Tool available (yes, no): Yes   Public domain (yes, no): No

Restrictions (copy rights, licenses, etc.): License

Tool supported (yes, no): Yes

Cost ($):

Control and input component: $8,000
Static analysis component: $6,000
Execution coverage analysis component: $8,000
Program documentation component: $4,000
Entire system: $26,000

Developer: General Research Corp.

Contact (name, address, and telephone no.):

William R. Dehaan
General Research Corp.
5383 Hollister Avenue, P. O. Box 6770
Santa Barbara, CA 93111
(805)964-7724

Tool summary: The heart of the RXVP80 is a large library capable of storing the result of analysis of very large programs (10,000 source lines). The system performs much of its analysis on an internal representation of the program as a directed graph. One of the primary features of RXVP80 is its ability to analyze only the new or changed modules of a program, using the stored library to check interfaces. Its static analysis component performs mode and type checking, CALL checking, set/use checking, and graph structure checking. The execution coverage analysis component performs the C1 coverage checking. The report generated includes COMMON matrix, input/output report, invocation report, calling tree, and cross references.

Performance and limitations: Ramamoorthy and Ho report that extensive man-machine interactions are required for the testing of programs and the test data preparation.

Documentation (type of available documentation): User manual

References:

T. Gilb.
Software Metrics.
Wintrhop Publishers, Inc., Cambridge, MA, 1977, pp 282.

C. V. Ramamoorthy.
Testing Large Software with Automated Software Evaluation Systems.
IEEE Transaction on Software Engineering, March 1975, pp. 46-58.

E. F. Miller and R. A. Melton.
Automated generation of Test Case Data Sets
Proceedings of International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 51-58.

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Software Research Associates.
Software Engineering Automated Tools Index.
P. O. Box 2432, San Francisco, CA 94126, (415)957-1441.

TRW Catalogue.
Software Tools Catalogue and Recommendations.
TRW, Defense and Space Systems Group, January 1979.

TESTING TOOL DATA SHEET

Date:
Acronym: SADAT
Title: STATIC AND DYNAMIC ANALYSIS AND TEST
Classification (all applicable categories): Static Analyzer,
Instrumenter,
Test Data Generator,
Symbolic Evaluator
Features: Instrumentation, statistical analysis, profile generation, coverage analysis, symbolic execution, tuning, tracing, path flow tracing, auditing, data flow analysis
Stage of Development (concept, design, implemented): Implemented 1978
Implementation Language (used to write the tool): PL/1
Implementation Hardware: IBM 370/168, IBM3033
OS (other software required):
Target Languages (of the tested module): Fortran
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.): 8,000 statements, requires 1M byte to run programs of some 100 statements
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: KERNFORSCHUNGSZENTRUM, KARLSRUHE GMBH
Contact (name, address, and telephone no.):
Udo Voges, Lothar Gmeiner and Anneliese Amschler von Mayrhauser
Tool summary: SADAT consists of six main components: command processor, static analysis module, test case generation module, path predicate calculation module, dynamic analysis module, and a data base which is the means of communication among the modules. The SADAT is controlled by a set of user commands which activate different modules to perform appropriate analysis on a user program. The tested single Fortran modules are assumed to be compiler error-free. The static analysis consists of lexical analysis which mainly creates necessary tables and structural analysis which creates a graph representation of the program. The errors detected in this analysis include unreachable statements, labels not declared or not used, variables not declared, not initialized or not used, and jumps into/out of loops. The test case generation automatically generates test data to exercise each d-d path at least once. The user can also specify his own paths to be exercised. In path predicate calculation, symbolic execution is used to compute the path predicate of every path in the selected set. The dynamic analysis performs instrumentation to determine useful information for identification of dynamical dead code, control of the correct number of loop iterations, and optimication of the most frequently executed parts.

Performance and limitations: The system has been in experimental usage in different projects since 1978. The static and dynamic analysis modules were found to be valuable and steady tools. The test data generation (symbolic execution) has some deficiencies such as requiring too much time and storage for large and complex programs, problems with loops and subroutine calls.

Documentation (type available): User manual, installation instructions

References:

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

U. Voges, L. Gmeiner, and A. Amschler von Mayrhauser.
SADAT - An Automated Testing Tool.
IEEE Transaction on Software Engineering, Vol. SE-6(3), May 1980.

A. Amschler and L. Gmeiner.
SADAT - A System for Automated Generation, Execution and Eval. of Tests for FORTRAN Programs, (in German), FKF-EXT. 13/77-2.

A. Amschler.
Test Data Generation as an Integrated Part of a System for the Automatic Execution and Evaluation of Tests, (in German), Kiplomarbeit, Universit at Karlsruhe, Germany, July 1976.

A. Amschler, L. Gmeiner, and U. Voges.
SADAT - AN Automated Testing.
Presented at the Workshop on Software Testing and Test Documentation, Ft. Lauderdale, FL, December 18-20, 1978.

L. Gmeiner.
Dynamic Analysis and Test Data Generation in an Automatic Test System.
Workshop on Reliable Software, September 22, 1978.

M. Seifert.
SADAT-EIN System Zur Automatischen Durchfuhrung and Answertung von Tests, KFK-EXT, 13/75-05, May 1975.

L. Gmeiner
Installation Instructions, KFK-IDT, 1978, unpublished.

H. Trauboth, W. Geiger, L. Gmeiner, U. Voges.
Program Testing Techniques for Nuclear Protection Systems.
Infotech State of the Art Report - Software Testing, Vol. 2: Invited Papers.
Infotech International, 1979, pp. 283-305.

TESTING TOOL DATA SHEET

Date:
Acronym: SELECT
Title: SYMBOLIC EXECUTION LANGUAGE TO ENABLE COMPREHENSIVE TESTING
Classification (all applicable categories): Symoblic Evaluator,
Test Data Generator,
Assertion Processor
Features: Symbolic execution, static analysis, path structure analysis, assertion checking, test data generation
Stage of Development (concept, design, implemented): Implemented 1974
Implementation Language (used to write the tool): LISP
Implementation Hardware: PDP-11
OS (other software required):
Target Languages (of the tested module): LISP SUBSET
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no): Yes (Stanford Research Institute)
Cost ($):
Developer:
Contact (name, address, and telephone no.):

SRI International
Advanced Computer Systems Dept.
333 Ravenswood Avenue
Menlo Park, CA 94025
(415)326-6200

Tool summary: SELECT is a symbolic execution tool which is intended to be a compromise between an automated program proving system and ad hoc debugging practice experimentally. SELECT includes: 1) semantic analysis of programs, 2) construction of input data constraints to cover selected program path, 3) identification of (some) unfeasible program paths, 4) automatic determination of actual (real number) input data to drive the test program through selected paths, 5) execution (actual or symbolic) of the test program with optimal intermediate assertions and output assertions, 6) generation of simplified expressions for the values of all program variables, in terms of symbolic input values, and 7) path analysis for each potentially executable path or for a user-selected subset of paths. Multiple executions of a loop with a path are defined as separate paths, producing a potentially infinite number of distinct paths. The number of loop traversals may be constrained by the user.
Performance and limitations: An experimental system
Documentation (type of available documentation):

References:

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

John D. Donahoo and Dorothy Swearingen
A Review of Software Maintenance Technology.
RADC-TR-80-13, Interim Report, Rome Air Development Center
Feburary 1980.

Software Research Associates
Software Engineering Automated Tools Index
P. O. Box 2432, San Francisco, CA 94126

R. S. Boyer, R. Elspas, and K. N. Levitt.
SELECT - A Formal System for Testing and Debugging Programs by
Symbolic Execution.
Proceedings of 1975 International conference on Reliable Software,
1975. pp. 234-245.

TESTING TOOL DATA SHEET

Date:
Acronym: SMOTL
Title:
Classification (all applicable categories): Test Data Generator
Features: Test data generator, regression testing, run-time error detection, coverage analysis, batch operation
Stage of Development (concept, design, implemented): Implemented 1976
Implementation Language (used to write the tool): SMOD (a COBOL-like language without direct access to storage devices)
Implementation Hardware: MINSK-32 (Soviet computer) 180K bytes main storage, CPU speed approximately 50,000 operations/sec.
OS (other software required):
Target Languages (of the tested module):
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.): 30,000 computer instructions
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer:
Contact (name, address, and telephone no.):
Tool summary: The operation of the system consists of six phases: directed graph construction, static analysis, covering set of paths con- struction, minimization of the covering set, test data generation, and report generation. The system uses a concept of state and standard optimization methods to generata a manageable number of test data sets for practical programs. In testing with the con- structed test data, the system demonstrates the function of all program parts that can be executed without abnormal termination and also gives diagnostic messages explaining the reasons of infeasibility of all the remaining untested program parts.
Performance and limitations: The system is effective for small programs (fewer than 300 statements containing about 11 conditional statements). But for the larger programs, the system takes very long processing time and doesn't generate useful results. The concept of 'state' used was proved to be insufficient for these programs.
Documentation (type of available documentation):
References:
J.Bicevskis, J.Borozovs, U.Straujums, A.Zarins, and E.F.Miller. SMOTL - A System to Construct Samples for Data Processing Programming Debugging, Technical Note RN-415 Software Research Associates, San Francisco, CA, April 1978.

TESTING TOOL DATA SHEET

Date:
Acronym: TDBCOMP
Title: TDBCOMP PROGRAM
Classification (all applicable categories): Maintenance Tool,
Comparator
Features:
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): JOVIAL J4
Implementation Hardware: CDC 3XXX
OS (other software required):
Target Languages (of the tested module):
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Public domain (yes, no):
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($):
Developer: TRW, Operational Software Operations M
Contact (name, address, and telephone no.):

David E. Heine
TRW, Operational Software Operations M
One Space Park
Redondo Beach, CA 90278
(213)535-3480

Tool summary: TDBCOMP compares and summarizes the differences between two data bases, where one data base is on tape and the other is active on disk. The importance of automatic data comparison in evaluation of the effect of changes (both coding changes and parametric changes) is that it saves many engineering man-hours otherwise wasted on manual data comparisons. It provides more accurate comparison than possible manually, and enables the engineer to focus his time and attention on analysis of the differences reported. This capability is needed on practically a daily basis during a software test process.

Performance and limitations:
Documentation (type of available documentation): User's manual
References:

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

Applied System Design Section, TRW Defense and Space Systems Group.
Software Tools: Catalogue and Recommendations.
TRW Automated Software Tools Series, January 1979.
U.S. Army TB22-18.

TESTING TOOL DATA SHEET

Date: 1/1/83
Acronym: TEC/1
Title: FORTRAN MUTATION SYSTEM
Classification (all applicable categories): Automatic Mutation System
Features: Test harness and driver, generates mutants, computes mutation scores, archives test files, produces test reports and statistics
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): FORTRAN
Implementation Hardware: PRIME 550, PRIME 450, VAX 11/780
OS (other software required): PRIMOS, UNIX
Target Languages (of the tested module): FORTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): 25,000 lines
Tool available (yes, no): Yes Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): License required
Tool supported (yes, no): Yes
Cost ($):
Developer:
Contact (name, address, and telephone no.):

Richard A. DeMillo
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, GA 30332
(404) 894-3180

Tool summary: TEC/1 allows entry of Fortran (ANSI 74) programs, interactive or batch entry of test data and interactive monitoring of program execution. Multiple modules may be processed. TEC/1 allows selection of specified mutation operators and testing strategies (e.g. statement coverage, arithmetic errors). Reports are produced to document the testing process.
Performance and limitations: In unit test of modules up to 2,000 lines, complete mutation testing can be carried out with modest resources. For larger modules, random sampling of mutants must be carried out.
Documentation (type of available documentation): Users guide, Internal maintenance, Installation manual, Specification

TESTING TOOL DATA SHEET

Date:
Acronym: TESTMANAGER
Title: TESTMANAGER
Classification (all applicable categories): Test Driver
Features: Regression testing, operation in batch or interactive mode
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool):
Implementation Hardware: IBM 360, 370, 30XX,43XX, ICL1900 Series, BS1000 and BS2000
OS (other software required):
Target Languages (of the tested module): Assembly, COBOL, CORAL, FORTRAN, PLAN, PL/I
Tool portable (yes, no):
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.):
Tool supported (yes, no):
Cost ($): 9,000 to 14,000
Developer:
Contact (name, address, and telephone no.):

Marketing Administrator
MSP Incorporated
21 Worthen Road
Lexington, MA 02173
(617)861-6130

Tool summary: TESTMANAGER is an interactive program and structure testing system for the development of reliable systems. TESTMANAGER enables a programmer to test individual modules in either a batch or interactive environment. Supplied in a version suitable for the operating system employed, TESTMANAGER supports modules written in Assembler, COBOL, CORAL, FORTRAN, PLAN and PL/I. Using a user supplied input stream of command statements to control the run and provide data to the user's module, TESTMANAGER executes the module with the data supplied and then displays the results of the test. Operation of TESTMANAGER involves four stages: defining the environment; formatting the data; executing the test run; and displaying the results. Defining the environment identifies the areas of storage to be used in the subsequent stages. It requires the specification of names and associated lengths for each storage area to be used. Formatting the data places values into the areas of storage defined in the first stage. Execution of the modules submitted for testing makes available, as directed, the areas of storage that have been defined and formatted. This state also provides for the simulation of test module interaction with non-present modules and the trapping and diagnosing of hardware program interrupts. The final stage allows the examination of the contents of areas of storage after the execution of test modules and permits predefined expected results to be compared with the actual results. During a single run, the user can test a single module or a series of logically associated or unassociated modules.

Performance and limitations:

Documentation (type of available documentation): TESTMANAGER Fact Book (available from MSP)

References:

Software Research Associates
Software Engineering Automated Tools Index
P. O. Box 2432, San Francisco, CA 94126

TESTING TOOL DATA SHEET

Date: 7/27/82
Acronym: TEVERE-1
Title: A SOFTWARE SYSTEM FOR PROGRAMS TESTING AND VERIFICATION
Classification (all applicable categories): Test Data Generator, Symbolic Evaluator
Features: The program is executed symbolically and the weakest precondition theory is applied to derive path predicates for a class of well-structured programs. The tool can be used both to derive automatically path predicates starting from program post-condition 'true', or to execute symbolically a program starting from a given post-condition.
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool): LISP 1.4
Implementation Hardware: DEC PDP-11
OS (other software required): RSZ 11M
Target Languages (of the tested module): IFTRAN
Tool portable (yes, no): Yes
Size (no. of source statements, memory size, etc.): At least 66K core memory
Tool available (yes, no): Yes Public domain (yes, no): Yes
Restrictions (copy rights, licenses, etc.): Licenses
Tool supported (yes, no): No
Cost ($): Not defined
Developer: S. Bologna - J. R. Taylor
Contact (name, address, and telephone no.):

S. Bologna
ENEA CRE-CASACCIA
S. P. ANGUILLARESE IM1.300
00060 Roma - Italy
Tel. Int (06) 69683708

Tool summary: TEVERE-1 is a software system intended to be used for validation of well-structured programs written in an ALGOL-like language which allows the use of only the three basic constructs of structured programming (assignment, while-do, if-then-else) plus input-output statements. Both the program used to derive test cases and the program used to aid program proof are based on the 'weakest precondition' theory applied to well-structured programs in order to derive 'path-predicates', logical requirements which must be fulfilled by test data if the program execution is to follow a particular path. The system is intended for use as part of the testing and verification of the software for a reactor safety system.
Performance and limitations: The system becomes very slow when applied to modules with more than a few hundred paths.
Documentation (type of available documentation): TEVERE-1: A Software System for Programs Testing and Verification

References:

S. Bologna.
TEVERE-1: A Software system for Programs Testing and Verification
CONGRESSO AICA '79, October 1979.

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

TESTING TOOL DATA SHEET

Date:
Acronym: XPEDITER
Title: THE PROGRAMMER PRODUCTIVITY TOOL FOR THE 80's
Classification (all applicable categories): Test Driver, Test Bed
Features: Regression testing, test environment preparation, operates in batch or interactive mode
Stage of Development (concept, design, implemented): Implemented
Implementation Language (used to write the tool):
Implementation Hardware: IBM 360/370
OS (other software required): OS or OS/VS operating system
Target Languages (of the tested module):
Tool portable (yes, no): No
Size (no. of source statements, memory size, etc.):
Tool available (yes, no): Yes   Public domain (yes, no): No
Restrictions (copy rights, licenses, etc.): Marketed Product
Tool supported (yes, no): Yes (Application Development Systems, Inc.)
Cost ($): Basic system $25,000, with all options $45,000
Developer:
Contact (name, address, and telephone no.):

Mr. Ronald D. Sleiter
Application Development Systems, Inc.
7420 Unity Avenue North
Minneapolis, MN 55443
(612)560-8633

Tool summary: XPEDITER is a testing and debugging tool for COBOL programmers. It responds to user commands, in batch or interactively under TSO, SPF or IMS, a) to create a test environment for any sequence of code from a few instructions to an entire program; b) to control test execution including interruption of execution, simulation of uncoded or coded logic and interception of abnormal terminations; c) to capture relevant information such as parameter values, working storage contents, or data that has changed. No change is made to source or object code. XPEDITER supports code compiled with the CAPEX OPTIMIZER. It requires no modification to the operating system. On-site training included with price.
Performance and limitations:
Documentation (type of available documentation): Application Description Manual (48)
References:

Software Research Associates
<u>Software Engineering Automated Tools Index</u>
P. O. Box 2432, San Francisco, CA 94126

Raymond C. Houghton, Jr.
Software Development Tools, NBS Special Publication 500-88
National Bureau of Standards

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

VOLUME 3
SOFTWARE TEST AND EVALUATION:
Current Defense Practices Overview

SUBMITTED TO
THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION
AND
THE OFFICE OF NAVAL RESEARCH
ONR CONTRACT NUMBER N00014-79-C-0231

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

Volume 3
Software Test and Evaluation:
Current Defense Practices Overview

SUBMITTED BY
GEORGIA INSTITUTE OF TECHNOLOGY

TO

THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION

AND

THE OFFICE OF NAVAL RESEARCH

FOR

ONR CONTRACT NO. N00014-79-C-0231
Subcontract 2G36661

June, 1983

Volume 3

Software Test and Evaluation:

Current Defense Practices Overview

Table of Contents

Page

1. Overview and Data Gathering Procedure 1

2. Findings of the Current Defense Practices Interviews 3

2.1. HQ and Development Command Interviews 4

2.2. Project Interviews 9

2.3. OT&E Agency Interviews 37

2.4. Development Organization Interviews 48
2.4.1. Applications Software 49
2.4.2. Support Software 76

2.5. IV&V Organization Interviews 88

3. Military Standards and Guidance 101

3.1. Department of Defense Directives and Instructions 102
3.2. Military Standards 111
3.3. Air Force Regulations 130
3.4. Army Regulations 139
3.5. Navy Regulations and Standards 154
3.6. Miscellaneous Documents 163

Appendix A: Data Gathering Guides 214

Appendix B: Bibliography 255

FOREWORD

This volume is one of a set of reports on Software Test and Evaluation prepared by the Georgia Institute of Technology for The Office of the Secretary of Defense/Director Defense Test and Evaluation under Office of Naval Research Contract N00014-79-C-0231.

Comments should be directed to: Director Defense Test and Evaluation (Strategic, Naval, and C^3I Systems), OSD/OUSDRE, The Pentagon, Washington, D.C. 20301.

Volumes in this set include:

Volume 1: Final Report and Recommendations
Volume 2: Software Test and Evaluation: State-of-the-Art Overview
Volume 3: Software Test and Evaluation: Current Defense Practices Overview
Volume 4: Transcript of STEP Workshop, March 1982
Volume 5: Report of Expert Panel on Software Test and Evaluation
Volume 6: Tactical Computer System Applicability Study

CHAPTER 1

OVERVIEW AND DATA GATHERING PROCEDURE

The testing and evaluation performed on software developed for DoD applications is influenced by a variety of organizations and guidance documents. The primary guidance which exists with respect to software T&E resides in DoDD 5000.3. The Services implement this directive in regulations which provide further guidance to their activities. In addition, the Development Commands of the Services may supplement the Headquarters' guidance in regulations, instructions, or pamphlets with which their subordinate Commands must comply. The final responsibility for adherence to the guidance rests with the project offices which monitor the activities of the Defense contractors. The Services' independent test and evaluation organizations are responsible for the operational test and evaluation of the systems produced. In order to assess the current Defense practices, the functional groups mentioned above were surveyed on subjects related to software test and evaluation.

The survey methodology consisted of conducting interviews with selected representatives of the military and industrial sectors. These groups included the HQ and Development Commands for the Army, Navy, and Air Force, project offices for selected programs, OT&E agencies, and Defense contractors. The subjects discussed during the interviews spanned the areas of military regulations and standards, reviews and inspections, testing techniques, tools, quality assurance, independent verification and validation, and risk assessment. Although the interviews covered a variety of topics, all were related to the software development process, and therefore, the quality of the final software product.

The survey was not a random sampling of Defense organizations and no attempt has been made to give statistical interpretations to the results. Rather, the study team was guided to selected project offices by the HQ and Development Commands and by OSD. Defense contractors were selected by the study team in consultation with NSIA. Several considerations helped to determine the mix of organizations selected for interviews. These considerations included the size of the organization and the type of software activity. The overall goal of the interview selection process was to give the most representative picture possible of current contractor practices. The interview results showed a high degree of similarity. The lack of significant deviation in the responses of these organizations is evidence that if, in some cases, current practices do differ significantly from what is described, those differences are most likely unique to the specific circumstances of the program or contractor involved rather than representative of the norm in the testing and evaluation being performed on military software systems today.

To aid in the data gathering effort, a set of data gathering guides was developed, consisting of one guide for each functional group being interviewed. The guides ensured that the same basic information was gathered during interviews with representatives of each functional group. The use of personal interviews rather than the mass mailing of questionnaires helped circumvent the problems of differing terminologies and low response rates.

As experience was gained with the use of the data gathering guides, they were modified and reorganized to increase their effectiveness. The final versions appear in Appendix A. Discussions with representatives of the Operational Test and Evaluation Agencies were held prior to the development of the data gathering guides; therefore, no guide is included for use when interviewing that functional group.

CHAPTER 2

FINDINGS OF THE CURRENT DEFENSE PRACTICES INTERVIEWS

This chapter reports the findings of the interviews which were conducted with selected military and industrial personnel. These results are organized according to the function of the respective organizations: HQ and Development Commands, Project Offices, OT&E Agencies, Development Organizations (contractors), and IV&V Organizations (contractors). Each section describes the organizations interviewed, the information requested, the purpose of the interviews, and the data gathered.

2.1 HQ AND DEVELOPMENT COMMAND INTERVIEWS

Interviews were conducted with representatives of the Headquarters and Development Commands for the Army, Navy, and Air Force. In addition, since the Air Force Systems Command is organized according to product divisions, each of which has an Embedded Computer Resources (ECR) Focal Point, results of interviews with four AF ECR Focal Points are also reported here.

The primary purpose of these interviews was to determine what guidance the Headquarters received from the Department of Defense with respect to software test and evaluation, what guidance they passed on to the Development Commands, and how the Development Commands were assisting the individual project offices. Comments on the effectiveness of the current regulations, etc., and suggestions for improvement were also solicited. Another area of interest was the view these offices have of the future of mission critical or embedded computer resources.

The primary vehicles used to provide the Military Services with guidance from the Department of Defense are DoD Directives (DoDD) and DoD Instructions (DoDI). Guidance documents which are relevant to the acquisition, development, and testing of mission critical or embedded computer resources include:

DoDD 5000.1 - "Major System Acquisition"

DoDI 5000.2 - "Major System Acquisition Procedures"

DoDD 5000.3 - "Test and Evaluation"

DoDD 5000.29 - "Management of Computer Resources in Major Defense Systems"

In addition, since Program Managers for major systems must report to the Defense Systems Acquisition Review Council (DSARC), the document entitled "Embedded Computer Resources and the DSARC Process" provides further guidance.

The DoD guidance has been implemented in Army Regulations (AR's), the Army's Development and Readiness Command (DARCOM) Regulations, Navy Tactical Digital Standards (TADSTAND's), and Air Force Regulations (AFR's). Those regulations and standards of interest to us are primarily:

AR 70-1 - "Army Research, Development, and Acquisition"

AR 70-10 - "Test and Evaluation During Development and Acquisition of Materiel"

AR 1000-1 -	"Basic Policies for Systems Acquisition"
DARCOM Reg. 70-16 -	"Management of Computer Resources in Battlefield Automated Systems"
TADSTAND E -	"Software Development, Documentation, and Testing Policy for Navy Mission Critical Systems"
AFR 80-14 -	"Test and Evaluation"
AFR 800-14 - Vol.I:	"Management of Computer Resources in Systems"
AFR 800-14- Vol.II:	"Acquisition and Support Procedures for Computer Resources in Systems"

For a description of the contents of these guidance documents, see Chapter 3.

For multiservice programs, the regulations of the lead service are usually followed. Procedures for coordinating multiservice test and evaluation are developed by Joint Program Offices and/or Computer Resource Working Groups. Agreements are documented via memorandums of understanding.

Though the documents referred to above outline general policy, few conclusions can be drawn regarding the actual restrictions imposed on a specific program. This is due to the latitude which Program Managers have when tailoring or interpreting the applicable regulations or standards. In addition, for extreme cases, a program may be granted a waiver which allows a regulation, standard, or parts thereof, to be completely ignored.

The strengths of the regulations are perceived to be that they fill a void where nothing else exists, that they have flexibility, and that they facilitate early planning for systems. The basic policy of AFR 80-14 is seen to be completely adequate.

The weaknesses of the regulations are seen in different lights: AR 1000-1 is too long; DARCOM 70-16 is untested, few Computer Resource Management Plans (CRMP's) have been submitted, and DARCOM approval of CRMP's is not required. AFR 800-14 has a good philosophy of testing but may conflict with the top down development strategy of Ada[1]. Other perceived weaknesses include the feelings that the regulations have no "meat" or impact, they are not uniformly or even necessarily good, and there is no good mechanism to keep them current.

[1] Ada is a registered trademark of the Ada Joint Program Office - US Government

Known enhancement efforts with respect to the regulations include the following:

- AR 1000-1 is being updated to reflect the new Carlucci acquisition policy. It may also be shortened in the process.

- AFR 800-14 is being revised to include Air Force policy regarding Independent Verification and Validation (IV&V), as well as to reflect recent changes to DoD acquisition policy.

The control which the Headquarters and Development Commands have over the software test and evaluation process involves the review of the Test and Evaluation Master Plans (TEMP's) which are developed for major programs. It should be noted, however, that the testing described in a TEMP usually refers to system testing rather than software testing. Thus, the actual control exerted by organizations outside of the Program Offices may in many instances be minimal. One exception to this involves the Air Force ECR Focal Points. These individuals may, in some cases, review proposals and statements of work to ensure that the software issues are properly addressed. In addition, they sometimes act as consultants to Program Offices and are members of Computer Resources Working Groups.

Another area of interest which the Headquarters and Development Command representatives were questioned about involves the source selection process and the amount of importance given to a potential contractor's internal policies and past performance regarding software test and evaluation. Though no formal guidelines exist, past performance does have an indirect influence on the source selection process since it affects how comfortable the source selection board is with the idea of working with potential contractors. In addition, one office is in the process of building a database concerning past performance, with the awareness that there may be legal problems with using this information.

As an alternative to evaluations based on past performance, Development Command personnel do evaluate the potential contractor's employee base, internal software development standards and tools, and facilities. This is felt to be more beneficial due to changes which may have occurred within the potential contractor's organization since the completion of previous government work.

Military standards and specifications have been developed to ensure that certain minimum requirements are met by the contractors when developing mission critical or embedded systems for the government. Those which the Headquarters and Development Command interviewees were cognizant of include:

MIL-STD-483 (USAF) -	"Configuration Management Practices for Systems, Equipment, Munitions, and Computer Software"
MIL-STD-490 -	"Specification Practices"
MIL-STD-1521A (USAF) -	"Technical Reviews and Audits for Systems, Equipments, and Computer Programs"
MIL-STD-1679 (NAVY) -	"Weapons System Software Development"
MIL-S-52779A -	"Software Quality Assurance Program Requirements"

These military standards and specifications are described in Section 3.2.

Criticisms of the military standards and specifications include the feeling that MIL-STD-1679 is too detailed and inappropriate in places, and MIL-S-52779A cannot be used to tell a contractor how to do a good job. It is also believed that it will require at least 2-3 years to implement recommended revisions to MIL-STD's 483, 490, and 1521A.

New technology trends relating to embedded computer resources which were felt to be potentially beneficial include the use of:

- Independent Verification and Validation (IV&V)
- Software Metrics
- Ada and the Ada Programming Support Environment (APSE)
- Reusable Software
- Hardware Facilities to simulate and develop repeatable tests
- Very High Speed Integrated Circuits (VHSIC)

Other new technologies mentioned include program design languages (PDL's), asynchronous software, the Military Computer Family, redundant circuit design, built-in testing, and pin electronics for testing. The use of software metrics for the determination of award fees on a major program will be described in Section 2.2.

The standardization of ISA's, busses, and interfaces were other efforts which were discussed that may reduce the significance of some of the problems we currently face with software test and evaluation. In contrast to this, VHSIC is seen to greatly complicate problems encountered during testing.

Although one interviewee feels that it is not the government's job to push new technologies, preparations are being made for their application. In particular, efforts which are underway with respect to Ada include the allocation of resources to allow its development and use, the search for candidate programs, the teaching of courses, and the use of Ada as a PDL for systems which are currently under development. Though much support exists for Ada, there is a strong feeling that, in order to effectively take advantage of the benefits it may provide, a gradual introduction of the language is important, readiness criteria need to be established for its use, and it should not be required prior to the availability of supporting tools.

Finally, the following general comments were made by the interviewees.

- When good requirements are provided, systems can be developed ahead of schedule and under budget.

- Personnel in the Project Offices who are responsible for the acquisition of software often do not have a software background.

- There needs to be a definition of adequacy for Program Reviews.

- The military doesn't know what reasonable contractual terms with respect to testing are.

- There is a need for the documentation of "lessons learned" so that old mistakes can be avoided.

2.2. PROJECT INTERVIEWS

OVERVIEW AND BACKGROUND INFORMATION

This section reports the findings which resulted from interviews with military representatives involved with specific projects. Information was gathered on the project status and history, military regulations and standards applied, reviews conducted, development testing and evaluation, acceptance testing, quality assurance programs, independent verification and validation activities, operational testing and evaluation, and risk assessment. Projects were chosen to achieve a breadth of knowledge concerning the involvement of the military Services in the software development and procurement process. Specific projects examined included:

Cruise Missile - This is a joint project being conducted by the Navy and the Air Force. Personnel from the Joint Cruise Missile Project Office (JCMPO) and other organizations involved in the system's development were interviewed. The Cruise Missile Project includes the development of the following:

Air Launched Cruise Missile (ALCM) - Air Force - Includes nuclear land attack missiles. ALCM was developed prior to the establishment of JCMPO.

Ground Launched Cruise Missile (GLCM) - Air Force - Also includes nuclear land attack missiles.

Sea Launched Cruise Missile (SLCM) - Navy - Includes conventional and nuclear land attack missiles and conventional anti-ship missiles.

The primary software systems which must be developed for the Cruise Missile include the software which resides in the missiles, the Weapon Control System, and the Mission Planning System.

AEGIS Weapon System, Mark 7 - Navy - AEGIS is an advanced shipboard weapon system. The software provides functions for display groups, radar systems, command and decision systems, fire control systems, and weapon control systems as well as a training control system. The AEGIS Weapon System can be tailored to a specific ship's needs by using subsets of the total system.

Combat System Engineering Development Site (CSEDS) - In conjunction with our AEGIS-related visits, we were given the opportunity to visit CSEDS. CSEDS provides a land-based engineering facility to design, develop, integrate and test the AEGIS ship combat system including the AEGIS Weapon System. Our examination of CSEDS related specifically to its use in the testing of the AEGIS Weapon System.

Remotely Piloted Vehicle (RPV) - Army - The RPV system is being developed to fill the requirement for unmanned aerial target acquisition, target identification and location/laser designation for laser seeking weapons, artillery adjustment and battlefield reconnaissance. Software is used in the air vehicle, the ground control station, and the maintenance shelter to provide functions for air vehicle flight, tracking, fault isolation, displays, and interfacing.

Data was also gathered on the following Army systems: TACFIRE, the Battery Computer System (BCS), and Firefinder. Finally, discussions were held concerning the Naval Tactical Data System (NTDS).

The programs surveyed were in various phases of the software development life cycle at the time the interviews were conducted. ALCM was to obtain initial operating capability in December 1982. The GLCM Weapon Control System was in the coding and testing stage, although some design modifications were also being performed. The Mission Planning System was installed at five sites in Europe, two in the continental U.S., and one in Hawaii. SLCM's conventional missiles (both anti-ship and land attack) were undergoing operational evaluation. The AEGIS Weapon System was deployed on its first ship. The RPV contract was let in 1975 with production planned for 1985 and the Critical Design Review occuring in the last months of 1982. TACFIRE was in the maintenance phase with updates scheduled for release in July, October, and December 1982. BCS had completed its operational testing and follow-on evaluation, and was to be fielded in October or November 1982. Firefinder was in production and fielded in Europe.

The programming languages used for these projects were as follows:

PROJECT	LANGUAGE
Cruise Missile:	
Missile Software	Assembly Language
Weapon Control System	Fortran & Assembly Language
Mission Planning System	Fortran & Cobol
Automatic Test Software	ATLAS
AEGIS	CMS2 & Assembly Language
RPV	Fortran, PLM, & Assembly Language
TACFIRE	TACPOLE (derivative of PL/1)
FIREFINDER	Assembly Language
BCS	SIRR
NTDS	CMS2 & Assembly Language

Due to timing and size constraints, 100% of the ALCM missile software was written in Assembly Language. AEGIS claimed less than 1% Assembly Language code.

The only opportunity for the reuse of software occurred in the Cruise Missile Project. Eighteen out of twenty-five software modules were common between the various versions of ALCM, GLCM, and SLCM. In addition, the Mission Planning System was able to reuse two ALCM modules.

A variety of hardware and operating systems were employed for these projects. The target machines for weapon systems tend to be microprocessors or minicomputers. Larger mainframes may be used for development or mission planning purposes. The Navy's hardware is proscribed in TADSTAND B (see Section 3.5) to be a member of the family of standard embedded computers and computer peripheral equipment including the AN/UYK-7, -14, -20, -43, -44 and upgrades to those systems. Examples of the hardware and operating systems used were:

PROJECT	HARDWARE AND OPERATING SYSTEMS
Cruise Missile:	
ALCM	LC 4516 C micropr
ossor Litton Inertial Navigation Element (INE) IBM 370 (host)
GLCM	ROLM 1666D with RMX-RDOS ROLM 1602 with RTOS
SLCM	
anti-ship missiles	16K IBM machine
land attack missiles	64K Litton machine
Mission Planning System	VAX 11/780 with VMS
AEGIS	AN/UYK-7 & AN/UYK-20 with standard CMS2 compilers, etc.
RPV	Intel 8085A, Norden 11-34M & SKC-312
TACFIRE	VAX 11/780
FIREFINDER	UYK-15 (Interdata)
BCS	4 bit microprocessors which emulate an Elliot 901 or 908
NTDS	AN/UYK-7 with Standard CMS2 compilers, etc.

Some special hardware devices used in the testing process were identified. ALCM uses an Instruction Level Simulator for module testing, as well as a subsystem simulator and miscellaneous black boxes for simulating such data as radar input. CSEDS, which was described earlier, is a major installation used as an interface simulator system (wrap around simulator) for AEGIS. For FIREFINDER, the Radar Environmental Simulator (RES) is used, with test tapes provided by the contractor. Other simulators will be identified in the subsection entitled "Testing and Evaluation Tools".

MILITARY REGULATIONS AND STANDARDS

Military regulations and standards invoked on these projects include the following:

PROJECT	REGULATIONS & STANDARDS
Cruise Missile	
ALCM Missile Software	AFR 800-14 MIL-STD's 483 (USAF), 490, 1521A & 52779A
GLCM Missile Software	AFR's 800-14, 122-9, & 122-10 SECNAV Instruction 3560.1 TADSTAND's A-E MIL-STD-1679 (Navy)
GLCM Weapon Control System	AFR 800-14 MIL-STD's 490, 1679 (Navy), & 52779A MIL-HDBK-255
SLCM	MIL-STD-1679 (Navy)
Mission Planning System	MIL-STD-1679 (Navy)
AEGIS	SECNAV Instruction 3560.1 TADSTAND's A-E MIL-STD's 490 & 1679 (Navy)
TACFIRE	AR's 18-1 & 380-38 DARCOM Reg's 70-10 & 70-16
BCS	DARCOM Reg's 70-10, 70-16, & 700-34 MIL-STD's 483 (USAF) & 52779A
NTDS	SECNAV Instruction 3560.1 TADSTAND's A-E MIL-STD-1679 (Navy)

The contents of these military regulations and standards are summarized in Chapter 3 of this document.

For the multiservice software testing and evaluation effort conducted for the Cruise Missile, final decisions as to which regulations and procedures would be followed (Air Force vs. Navy) were made by the director of the Project Office.

The only waiver granted for any surveyed projects' software was one for the SLCM anti-ship missile when it was unable to meet the reserve capacity requirements for memory space.

The only specific positive input received on current military regulations and standards concerned MIL-STD-1679 (Navy), "Weapons System Software Development". Its strength is seen to lie in the approach it takes to documentation and milestones. One office is hopeful about proposed MIL-STD-SDS, "Defense System Software Development". It was felt that this standard will allow the government to have more control over the software development which it pays for.

Specific negative remarks concerning the regulations and standards were more numerous. They included:

- Much of the required documentation is useless.

- Too much emphasis is placed on MIL-STD-1679.

- The requirement for flow charts should be replaced with one for the use of a PDL.

- Regulations should require the monitoring of various statuses; a definition of what a computer is and is not is needed.

- There is a need for a "good" testing regulation; every requirement must be tested. Guidance is needed relative to testing bad inputs, extreme cases, and overload conditions.

- Less specific regulations and more policies are needed; firmware must be recognized and regulated as software.

- Human engineering standards are needed.

- The 20% reserve capacity requirement should only be applied when the system is fielded for the first time.

- Guidelines should be just that, guidance - not instruction; the percentage of required reserve capacity should be project specific.

- Standard ISA's are an unrealistic requirement and will never be obeyed.

- In general, the regulations and standards are too vague. The quality of the final product is dependent on the contractors involved and not the regulations or standards imposed.

More general comments on regulations and standards included:

- An effort is currently underway to rewrite MIL-STD-1679 and TADSTAND 9 for inclusion in Army SOW's.
- Good software development practices are needed in the first place; you cannot depend upon MIL-STD-1679 as an instructional tool.
- In spite of the standards, etc., performance specifications are not consistent in either form or content. However, personnel have learned to live with this as a fact of life.
- Navy standards are completely different from Air Force standards.

THE SOURCE SELECTION PROCESS

The source selection process used by the project offices and the related activities of other organizations interviewed varied. Reviews were performed on almost any combination of the following when rating potential contractors: the proposal, the software development approach, the quality assurance program, documentation standards, and configuration management procedures. Some potential contractors may be required to agree to the submission of quality reports on module and integration testing. Finally, in some cases, the evaluators also verified that IV&V organizations would be allowed to witness and/or dictate testing. The evaluation of potential IV&V contractors may include all of the above plus the requirement that the candidate write technical procedures for review when determining competence. Descriptions of the source selection process used for some of the projects examined follow.

For ALCM, one function of the source selection team was to look at code for structure of design and modularity. They also evaluated potential contractors based on the existance of an independent test group, the quality of the test program, and whether the development methodology follow a requirements - design - code - test pattern. Additionally, in order to select a contractor for the ALCM guidance system, a "fly-off" was conducted between two potential contractors' preliminary versions of the guidance system. The winner of that fly-off has been the sole source for the guidance system ever since. One of the primary factors contributing to the decision was the immaturity of one of the contractor's software.

For the GLCM Weapon Control System, criteria were determined, an RFP was written, and recommendations were made based on the responses. For the Cruise Missile Mission Planning System, the contractor was chosen as a result of proposal analysis. Finally, for NTDS, one of the Navy's Fleet Combat Directions Systems Support Activities (FCDSSA's) served a technical selection role by ranking or grading responders to the RFP on the technical competence demonstrated, thereby providing approximately 40% of the weight in the selection decision.

Internal policies and past performance of potential contractors may be a consideration when letting a contract, if only on an informal basis. One of the organizations interviewed looks at Contracting Officers' Reports for information about previous contractors. Another organization considers this to be a major weakness of the source selection process because "it does not carry as much weight as it should". Yet another group, however, does not look at past performance, and believes that since the government is not known for providing quality requirements it is not fair to judge previous contractors on past performance. One specific internal policy which would have a positive influence on one of the groups interviewed is to require the use of a PDL.

For the GLCM Weapon Control System, the source selection team determines the amount of weight to be given past history, and can request a review of internal policies. The team also reviews the Computer Program Development Plan. For the Cruise Missile Mission Planning System, part of the justification for choosing the contractor was previous experience in the area. For NTDS, there is no direct examination of past performance, but it is believed that "experience helps"; the attitude taken is that proposals, including test plans, are to be believed completely.

SOFTWARE DEVELOPMENT LIFE CYCLE

The acquisition process and, therefore, the software development life cycles for the programs studied differ depending upon time and/or budget constraints, requirements that were poorly defined or changed in mid-program, etc. As a result of these influences, the acquisition process frequently deviates from the "textbook" approach. Another reason for deviation from the standard approach involves new systems which are very similar to systems which have already been developed. As was mentioned previously, this was the case with the GLCM and SLCM missile software. The ALCM missile software provided the base system which was then modified to meet the unique requirements of GLCM and SLCM.

Unfortunately, the GLCM Weapon Control System has encountered various problems as a result of the use of a "non-standard" approach to software development. That approach involves coding being completed from Program Performance Specifications with the design being completed after the fact. In addition, some integration testing has been performed prior to module testing. When this interview was conducted, the contractor was being required to perform the neglected module tests on any modules which were modified. The Cruise Missile Mission Planning System enjoyed a more typical life cycle, wherein the government developed requirements and specifications, issued a RFP, and let the contractor produce the software. No major problems were reported in this case.

The AEGIS Weapons System's development process included a competitive systems design procurement, the prototyping of major software systems, and the issuing of a contract to produce the full system, followed by the "normal" development cycle including the Preliminary and Critical Design Reviews and an OPEVAL (operational evaluation). For NTDS, the FCDSSA developed the top level requirements and the Interface Design Specification; the Program Performance Specification was also developed in-house; the Program Design Specification was developed by the contractor who implemented and debugged the software. Upon the completion of the functional testing, the system was delivered to the Navy. The acceptance testing, performed by a Navy test group onboard ship, was system integration testing rather than just software testing. There has been no OPTEVFOR (Operational Test and Evaluation Force) involvement to date.

Firefinder was an accelerated program and did not follow the normal acquisition process due to time constraints. TACFIRE, as was stated previously, is currently in the maintenance phase of the life cycle. Three new versions of the software (one being tested prior to release, one being coded for the next release, and one being designed for the release after that) are under development at any one point in time with deliveries to the field being accomplished every six months.

One of the interviewees defined a good development plan to be one which "includes a description of the tasks, the methodology, and the controls necessary for all phases of the development effort".

Various reviews and reports may be required to attempt to monitor and control the software development process. Requirements of the projects of interest included:

- Reviews of test plans, procedures and reports for validation and acceptance tests. (ALCM)

- Reviews of test specifications, plans, and procedures. In addition, on-site military personnel witness system level tests. (GLCM Weapon Control System)

- Test reports from in-plant acceptance tests, as well as quarterly reports from IV&V organizations. (Cruise Missile Mission Planning System)

- Monthly test status reports; reviews of programmer notebooks; element, integration, and operational test plans, procedures, and reports. (AEGIS)

- Military personnel performing the management function at field test sites. (TACFIRE)

- Monthly status reports; reviews of test plans, procedures, and reports. Validation test matrix of requirements versus test cases is required, as well as specifications reviews and requirements analysis reports. (BCS)

- Weekly status reports describing the number of test steps completed and the number of Program Trouble Reports produced; approval of test plans and procedures for functional and acceptance tests. (NTDS)

DOCUMENTATION ITEMS

The variety of possible documentation items which may be produced as a system proceeds from concept to design to the final operational system is evidenced by the following list of those items produced for the projects examined.

- System Specifications including Part I (the Computer Program Development Specification), and Part II (the Computer Program Product Specification); the Interface Control Document; Test Plans, Procedures, and Reports; and the Version Description Document. (ALCM)

- Prime Item Development Specifications, Program Performance Specifications, Program Design Specifications, Interface Specifications, and Program Package and Version Description Document. (GLCM)

- Program Performance Specifications, Program Design Specifications, a Data Base Design Document, and a Program Description Document. (SLCM)

- Functional Descriptions, System and Program Specifications, Design Specifications, a Program Package, User Manuals, Maintenance Manuals, and Interface Manuals. (Cruise Missile Mission Planning System)
- A-Level System Specifications, B5 Specifications (Computer Program Development Specifications), C5 Specifications (Computer Program Product Specifications), Data Base Description Document, Program Description Document, etc. (AEGIS)
- B5 and C5 Specifications, Version Description Documents, and a Fielding Package consisting of training manuals and self-paced training. (TACFIRE)
- Functional Specifications, and B5 and C5 Specifications. Note: B5 and C5 specifications were rewritten after coding was complete. (Firefinder)

In the examples above, the Part I, B5, and Program Performance Specification are essentially the same document. This is also true for the Part II, C5, and the Program Design Specification.

REQUIREMENTS ANALYSIS

Requirements analysis is the validation of the software requirements/specificatons prior to implementaton. Analysis activities may consist of engineers' and analysts' reviews of the requirements to make sure that they make sense, are needed, and are not either already implemented or in conflict with something that is. Simulators may be used to further examine the requirements or a performance analysis document may be developed. In some cases, the users may also review the requirements.

The requirements analysis performed for AEGIS included modeling, simulation of algorithms, and a sensitivity analysis. That conducted on the GLCM Weapon Control System, the Cruise Missile Planning System, and BCS benefited from the participation of IV&V organizations. In addition, the BCS requirements were analyzed specifically for testability.

DESIGN ANALYSIS

The design for ALCM, the Cruise Missile Mission Planning System, and BCS were all developed using a PDL. In addition, ALCM and the Cruise Missile Mission Planning System were the only systems employing a top-down software development methodology, though this is a requirement for any new software for GLCM and SLCM.

Activities relating to the validation of software design prior to implementation include informal comparisons with the requirements; design reviews (including Preliminary and Critical Design Reviews); checks for logical sequence and structure, and the accomodation of growth; and traces of the design into the requirements.

Some design reviews conducted for a program may consist of informal reviews where "people walk across the hall and talk to each other" or the hopeful reliance on highly qualified people. However, formal government design reviews are inevitably required. These formal reviews are the Preliminary and Critical Design Reviews (PDR's and CDR's) and they comprise critical milestones in almost all development efforts. PDR's and CDR's may be conducted by the project office, the development contractor, or both. MIL-STD-1521A contains the procedures to be followed during PDR's and CDR's. Only the Cruise Missile Mission Planning System required strict adherence to this document. Other Cruise Missile development organizations used it, at best, as a guideline. Participants in the formal reviews may include representatives of the Project Office, development contractors, any IV&V contractors, DT&E agencies, OT&E agencies, and users. It is not unusual for the attendance to exceed 100 persons. It is difficult not to question the usefulness of these reviews for the software with such a large number of persons in attendance.

No specific design-to-test procedures or procedures for the quantitative assessment of software design maturity and supportability were found in use on the projects examined. The general opinion appears to be that if structured design and programming are used, then, design-to-test procedures are unnecessary.

One purpose of software prototypes is to refine system requirements/specifications and designs prior to implementation. In some cases, prototypes may result indirectly from building many versions of the same system, as with the Cruise Missile, or from the requirements changing as the government refines its idea of what exactly is needed from the system. For the Cruise Missile Mission Planning System, the contractor built a prototype prior to the receiving the contract. GLCM and AEGIS also used prototypes for the purpose stated above.

The software design is usually baselined upon successful completion of the Critical Design Review. In some instances, however, it is not baselined until after in-plant acceptance has occurred, the system has been deployed, or never. In cases where the design is not baselined, the requirements/specifications are.

CODE ANALYSIS

Structured programming was used for the ALCM Missile Software, the GLCM Weapon Control System, the Cruise Missile Mission Planning System, and AEGIS. For ALCM, this was accomplished through the use of code macros (the ALCM missile code was 100% Assembly Language). For AEGIS, preprocessors were used to enforce programming standards.

Program analysis is the verification that code conforms to the original software design. Techniques used to accomplish this task included:

- peer walkthroughs. (ALCM)

- reviews by supervisors, and a detailed code analysis by IV&V organizations on a very limited number of modules. (GLCM)

- internal contractor code reviews with the participation of the government. (Cruise Missile Mission Planning System)

- code walkthroughs with peers, engineers, and the government. (AEGIS)

- code walkthroughs and code reviews by the more experienced programmers. (NTDS)

- comparison of the code with the specifications. (Firefinder)

It should be noted that the activities described above are usually internal to the development organizations. In many cases, the interviewees did not have an extreme amount of confidence in the information they were providing.

COMMENTS ON ANALYSIS ACTIVITIES

Major likes and dislikes of the analysis techniques discussed thus far, as well as perceived strengths and weaknesses, and other comments included the following:

- Competitive fly-off's can damage the requirements "flow down" process.

- The structure of the requirements document usually dictates the structure of the design and the code.

- Technical design reviews are good. Care should be taken to ensure that they do not become political management reviews.

- It is good to have users present at PDR's so that they can see what they will be receiving and provide input; however, they can hinder progress at CDR's.

- A better review process including structured walkthroughs is needed.

TEST PREPARATIONS

It is generally recognized that preparations for software testing should begin as early as possible in the software development life cycle. Unfortunately, the realities of time and money constraints may, in many cases, interfere with good intentions in this area.

For the GLCM Weapon Control System, preparations for integration testing begin during the coding phase. (Remember that in this case, module testing has been neglected in the past.) The test group develops the test cases which are then validated during the actual testing process. The software system level tests are written by the same test group, and then reviewed and modified by JCMPO personnel. Due to the problems that have been encountered, a group of JCMPO personnel now resides on-site with the contractor to monitor the software development process.

The Test and Implementation Plans for the Cruise Missile Mission Planning System were developed at the Preliminary Design Review. For AEGIS, it was reported that, although the preparations for testing began "very early, they fell by the wayside". In this case, all test plans and procedures are written by the contractor.

Preparations for testing NTDS begin once the performance specification is baselined. The test plans developed trace each requirement into at least one test case. The goal is "reasonableness" in terms of testing both nominal and out-of-range inputs - not exhaustivity. TACFIRE testers interface with the software designers and coders when preparing for testing.

DEVELOPMENT TESTING AND EVALUATION

The project offices, in general, have very little visibility into the development testing and evaluation process employed by the contractors. Therefore, the following information represents the "suspicions" of the project offices rather than exact data on the development testing and evaluation conducted.

The software which resides in the ALCM missile, the GLCM Weapon Control System, the Cruise Missile Mission Planning System, and NTDS all undergo scenario type testing (as opposed to generic type testing). The bottom-up and black-box testing strategies are combined to produce the development testing and evaluation procedures for the ALCM missile software. The GLCM Weapon Control System software is exercised using requirements-based tests with nominal values.

In the ideal case, all critical software computational and decision algorithms and their timing assumptions should be validated. For ALCM, the timing assumptions were validated through the use of worst-case tests. At the time of the interviews, no timing tests had been performed for the GLCM Weapon Control System. Hardware and software timing statistics are available for AEGIS. No distinction was made between testing critical and non-critical computational and decision algorithms.

In addition to the testing and evaluation conducted by the development organizations, in some cases, an independent assessment of the software is also performed. This task may be accomplished through the use of independent test or verification and validation groups within the contractor's organization, military personnel, or IV&V organizations. This topic will be discussed in more detail later.

INTEGRATION TESTING

The purposes of integration testing are twofold. The first is to verify that software modules, subsystems, etc., interface with each other as required. The second is to verify the interfaces of the software with the target hardware and system. Whether this testing consists of multiple distinct phases or just one phase is a function of the availability of the hardware and other components of the system under development. In many cases, simulators play a very important role during integration testing.

For some elements of the Cruise Missile Project, JCMPO personnel witness the integration testing performed in the contractor's facility. A six degrees of freedom loop simulator is used during at least part of the integration testing phase. When available, actual flight test data is used to update the data bases which reside in the simulator. Most test cases executed during this testing are based on nominal inputs. For ALCM, the initial module interface testing was accomplished by adding one module at a time to the software package being tested during integration testing.

For the GLCM Weapon Control System software, the initial integration tests were performed on the developer's build. The government supplied pre-engineering equipment for use during testing, as well as system level tests for execution (five no-fault tests; twelve fault tests). Currently, all builds are performed by a configuration management organization who also prepares the version description document. Test procedures are updated as required by any modifications incorporated into the new versions being tested.

For AEGIS, testing begins at the module level and proceeds successively through more complex stages. Once a module's capabilities have been successfully demonstrated during module level testing, it is integrated into a subprogram build which is an operational subset of the total software system. The final build comprises the total software system which, upon completion of its integration testing, is integrated with the appropriate equipment to form an operational element in the multi-element environment of the test site. Operational elements are eventually integrated into an operational AEGIS Weapons System configured as required to satisfy a particular ship's mission.

The decision as to how much integration testing is necessary and sufficient is usually a subjective one. For the integration testing that is performed, however, the decision that this phase of testing is complete, in most cases, occurs after portions of the test set have been executed numerous times to verify the correction of any errors which were documented via Program Trouble Reports. One method of measuring progress during integration testing is to compare the core storage requirements of that portion of the software system which has satisfactorily completed integration testing with the estimated size of the total software system.

The perceived strength of integration testing depends upon the effectiveness of any simulators used. For one program, the lack of government influence over the testing performed was viewed as a weakness of integration testing.

ACCEPTANCE TESTING

Acceptance testing is software system level testing. In most cases, the government exercises approval power over the contents of the acceptance test plans and procedures. A summary of the acceptance testing performed for some of the projects investigated follows:

- Scenario driven tests based upon the requirements including limited stress testing. (GLCM missile software)

- System level tests written by the contractor's test team. The tests are reviewed by the on-site military personnel and JCMPO for adequacy. Also includes government quality testing which consists, in part, of stress and duration testing. (GLCM Weapon Control System)

- In-plant testing, demonstrations, and documentation reviews. (Cruise Missile Mission Planning)

- Results from in-plant development and operational tests using a simulator are evaluated. (Firefinder)

- Formal Qualification Testing. (BCS)

In order to establish the proper execution of the software, all functions are demonstrated during acceptance testing for the GLCM Weapon Control System and the Cruise Missile Mission Planning System. In order to demonstrate that the software will operate correctly in the user environment, simulators (sometimes in combination with other system equipment) are often used. This is the case with the GLCM Weapon Control System. In addition, this project uses acceptance testing to demonstrate the compliance of the software with any timing requirements which may exist.

We were not able to identify any procedures for the quantification of requirements to allow the establishment of threshold values for acceptance. The name, acceptance testing, can be misleading since this implies the possibility of non-acceptance. In actuality, very few rejections occur. In most cases, the software is accepted with the understanding that any deficiencies identified prior to acceptance will be corrected. One organization, however, exressed the opinion that the software should be rejected until the contractor "gets it right".

The primary strength of the acceptance testing process is seen to be the variety of personnel involved. In addition, when multiple testing sites are used, the confidence in the proper performance of the software increases.

Weaknesses identified relate to the large number of "implicit tests" involved when demonstrating functional correctness, the large number of tests executed previously which are reused for acceptance testing, the small number of errors found during acceptance testing, and the assumption that sufficient low-level testing has been completed in advance. Another problem which may arise, if military personnel on TDY are involved in acceptance testing, is the loss of concentration if the testing forces them to be away from home for an extended period of time.

TESTING AND EVALUATION TOOLS

The primary testing and evaluation tools used on the projects examined were simulators. The importance of the use of simulators during the testing process is evident from the number which were identified. Simulators used for the Cruise Missile Project include the following:

- Navigation analysis simulator.
- Six degrees of freedom closed loop simulator.
- Wrap around simulator program (WASP).
- GLCM Lab (GLAB) located at development contractor's site.
- Launch Control Center using a Launch Integration Platform Simulator which simulates the transporter-erector-launcher (TEL) and a Digital Weapons System Simulator.
- GLCM system integration lab (GSIL) at integration contractor's site, where the actual hardware can hook up to an actual TEL or a simulated TEL, and a simulated missile;
- MSDEF simulator which is located onboard the airplane and simulates missiles.

In addition, AEGIS uses CSEDS, the land-based test site described previously, and the AEGIS Source Code Processor, a code auditor.

METRICS

Very few instances of the use of metrics to evaluate the quality of the software produced were identified. One of interest, however, involved the RPV project. In this case, the development contractor hired another contractor to evaluate its design and code using metrics. The methodology was based on the work supported by RADC and described in RADC-TR-77-369, Volumes I and II. Quality software incentive awards were to be paid to the development contractor by the government based on the evaluation of the other contractor.

The metric score was a normalized "goodness" score directly related to the percentage of compliance with certain pre-defined standards, etc., where 1 (or 100%) was perfect and 0 (or 0%) was totally non-compliant. The design quality factors of interest were defined to be correctness, reliability, flexibility, testability, maintainability, and intraoperability. The attributes or criteria related to the quality factors were traceability, completeness, consistency, accuracy, error tolerance, simplicity, modularity, communications commonality, and data commonality. The actual measurements were based on a cross reference relating modules to requirements (traceability); a completeness checklist (completeness); procedure consistency measures and data consistency measures (consistency); the sufficiency of numerical methods (accuracy); the recovery from improper input data, computational failures, and hardware faults (error tolerance); design structure measures (simplicity/modularity); protocol standards and single module interfaces with other systems (communications commonality); and translation standards (data commonality).

Unfortunately, the use of metrics to evaluate RPV's software has been discontinued. The decision for cancellation was an economic one made by the development contractor who was paying the evaluating contractor.

COMMENTS ON SOFTWARE TESTING

Miscellaneous comments made relative to experiences with the testing and evaluation performed for the projects examined included:

- It is important to have good people forming a strong team with a good track record.
- The requirements and their interpretation are not necessarily consistent between the government and the contractor.
- The amount and quality of testing performed is deficient. Testers are not independent of the coders and often do not know how to test.
- The majority of the tests conducted are no-fault tests.
- There is very little human factors testing.
- Testing is terminated to avoid overrunning contracts.
- Configuration management is very important during testing.
- The importance of quality simulators cannot be stressed enough.
- There is an insufficient use of tools during the testing and evaluation process.
- The documentation delivered is often inadequate.

TEST DOCUMENTATION PROCEDURES AND REGRESSION TESTING

Testing performed is usually documented via the use of test plans, procedures, and reports. These documents are required for software system level testing. Lower level testing is usually internal to the contractor's organization where the documentation procedures may vary or, for module testing, be nonexistent.

For the Cruise Missile Project, the test plans and procedures are approved by JCMPO. As the tests are executed, observers check off each test step as it is completed, record any anomalies, and log any changes. In some cases, the test plans and procedures are stored on disk to facilitate changes which may be necessary due to changing requirements.

The vehicle used to report and track software errors is the Program Trouble Report. This document describes the failure conditions and the impact of the error on the requirements, design, and code.

In most cases, errors are categorized according to severity. The severity of any error increases when nuclear application are involved, as is the case for some versions of the Cruise Missile. Users may become involved in the process of assigning priorities to the errors in terms of which must be corrected immediately, where workarounds exist, etc. For the AEGIS project, graphs showing how fast new errors are being discovered are maintained to track the progress toward a quality product.

Those regression testing procedures for software that were described in the interviews include:

- Run complete tests. Also use tools to compare versions of code. All changes must have a comment including a Program Trouble Report number. (ALCM)

- Run system level functional tests. (GLCM Weapon Control System)

- The testing is scenario-based and performance oriented. Special tests for modifications are run when appropriate. (AEGIS)

- Test new functions. Use personal judgement to determine what other areas should be tested. (NTDS)

- Execute benchmark tests to check out old code using an automated system with manual input for testing the man-machine interface. This is scenario-type testing. (TACFIRE)

Many projects plan to develop standard test sets, if they do not already exist, for use during regression testing. Once this occurs, the problem to be solved involves the determination of whether the complete set should be executed each time or if a subset of the tests is sufficient. Maintenance procedures for the test set when modifications are implemented must also be developed.

QUALITY ASSURANCE PROGRAM

Software quality assurance (QA) programs vary from organization to organization. Potential elements of a QA program include software design reviews and audits; the witnessing of acceptance tests and, in some cases, limited development tests; and the final software configuration audit prior to the installation of the software in an operational environment. In addition, some QA organizations track and sign off on Program Trouble Reports. These activities may be performed by government QA organizations and/or those maintained within the corporate structures of the contractors involved. In the instances where the contractors' QA organizations were responsible for the activities described, limited information was available during the interviews conducted.

For ALCM, the software acceptance testing was witnessed by the government's QA organization; the contractor's configuration management organization performed the final software configuration audit (FCA). Both the physical configuration audit (PCA) and the final configuration audit were reviewed by the government.

In the case of GLCM, a software engineering practices manual was developed based on the requirements of MIL-STD-1679. This manual provides standards and procedures for the QA program. Compliance with the standards and procedures is ensured by reviews which are conducted by the contractor's QA organization. The software QA organization must also approve all deliverable documents.

Software design reviews and audits, as well as code walkthroughs, are "sometimes" conducted by the QA organization for AEGIS. Acceptance testing is witnessed and a final software configuration audit is performed. It should be noted that the reviews conducted are non-technical reviews.

Comments on software quality assurance programs included the following:

- The desired elements of a software QA program are not well defined. Once more experience is gained in this area, a standardized QA program should be developed.

- QA personnel are not technical enough.

- In order to maintain the technical proficiency of QA personnel, they should be rotated in and out of development groups on a yearly basis.

- In some cases, it may be necessary for government personnel to help train a contractor's QA personnel.

- For one program, it was felt that QA really had not contributed anything.

- In addition to the QA organizations, all other groups involved in the development and acquisition of a program should be involved in the PCA and FCA. This gives everyone one last chance to voice their opinions.

INDEPENDENT VERIFICATION AND VALIDATION

In the past, independent verification and validation of software was typically performed by independent contractors. Today, however, the military services are beginning to realize the benefits of having IV&V activities performed in-house by the organizations which will ultimately be responsible for maintaining the software under development. This is the case for portions of the Cruise Missile Project.

The IV&V organizations report directly to the project office. The involvement and responsibilities of the project office with respect to the IV&V of the software include managing and funding the effort, approving the IV&V plan, supplying documentation, etc. to the IV&V contractor, and forwarding any IV&V problem reports or comments to the development contractor for resolution.

Under ideal circumstances, the IV&V organization would become involved with the project no later than the development contractor. In actuality, this is not always the case. For the projects examined, the initial involvement of the IV&V organizations ranged from "Day 1" to well into the coding phase.

An analysis of the software requirements for completeness, correctness, consistency, traceability, and testability is performed to some degree by the IV&V organizations for all of the applicabe projects (those which have the support of an IV&V organization).

For ALCM, the IV&V organization, also, provided information to the source selection board. In-depth IV&V activities were only conducted for critical applications areas of the software. These activities included independent testing using an Instruction Level Simulator.

The GLCM missile software was tested independently using data from a nominal mission. When the necessary documentation is available, an analysis of the software design for correctness and satisfaction of the requirements will be conducted for the GLCM Weapon Control System. In addition, the IV&V organization performed code analyses and independent testing including both nominal and stress tests. This testing supported the development testing process.

The IV&V organization for the Cruise Missile Mission Planning System performed design analyses, supported development testing, and conducted validation testing using both nominal scenarios and those which stress the system.

AEGIS had no IV&V as such, but the engineering support contractors reviewed the performance specifications for content, completeness, and consistency. That organization also developed some test plans.

The RPV IV&V contractor was responsible for examining only flight critical functions. In addition, the IV&V contractor, for a limited time, evaluated the software using metrics as described in an earlier section. (In this case, the IV&V contractor was paid by the development contractor.)

The IV&V activities performed for BCS included reviewing documentation, witnessing contractor's tests, and conducting independent tests. The independent tests consisted of black-box tests where each requirement was tested by at least one case. A conscious decision was made to strive for repeatability of the tests rather than coverage during testing. Therefore, a system reload was performed before executing each test case.

The primary strength of IV&V is seen to lie in its independence. Other comments made included:

- IV&V is expensive.

- Many benefits are lost if IV&V begins too late.

- On complex systems, it is difficult to get IV&V personnel "up to speed" in a timely fashion.

- IV&V organizations are not responsive enough. IV&V reports lag behind the development process thus diminishing their effectiveness.

- The flow of information from the development contractor to the project office to the IV&V contractor and back causes major delays.

- In some cases, IV&V contractors exagerate problems to increase the apparent importance of IV&V.
- In one case, the only errors found to date by the IV&V organization have been documentation errors.

OPERATIONAL TESTING AND EVALUATION

The primary purpose of operational testing is to evaluate a system's capabilities in as realistic an environment as is possible. Certain limitations, however, always exist. One example is that when testing a cruise missile it cannot be fired over the North Pole (i.e., towards the USSR). OT&E also provides the opportunity for another independent assessment of the software, this time in the context of the total system.

Operational testing and evaluation is usually performed by organizations within the military services which have been formed expressly for that purpose. For details relating to these organizations and their philosophies concerning OT&E of software, see Section 2.3. In some cases, other organizations may assist the OT&E Agencies. For BCS, both TRADOC and the Field Artillery Board were involved in operational testing.

The OT&E Agencies do not report to the project offices, but to the appropriate Chief of Staff, etc. The project offices are responsible for providing data and information concerning schedules, etc., about the system under development to the OT&E Agencies. Since the OT&E Agencies are independent of the project offices, the information concerning OT&E which was available during these interviews was rather limited.

During the GLCM missile software development effort, OT&E personnel attended the Preliminary and Critical Design Reviews and forwarded action items to the project office for resolution. They were also involved in the validation of simulators used during testing. Finally, special flight tests were conducted to validate critical software functions.

Similarly, OT&E personnel attended PDR's and CDR's for the GLCM Weapon Control System and forwarded comments to the appropriate project personnel.

In the case of the Cruise Missile Mission Planning System, OT&E personnel were involved, to some extent, in the development testing and evaluation. They also attended the PDR's and CDR's.

During OT&E for AEGIS, a conscious effort was made to develop scenarios which would exercise the software throughout the total range of expected operational conditions. In this case, AEGIS project personnel witnessed the operational testing to assist OT&E personnel with problem resolution. Again, OT&E representatives attended design reviews and had limited involvement in the development testing and evaluation conducted.

The operational testers for BCS worked with the developers to identify critical issues for OT&E. They also assisted the developers during the Final Qualification Tests and took those test results into consideration during their evaluation. Efforts were directed toward trying to "break" the system during OT&E. The accuracy of the data was evaluated; the fidelity of the system operation between versions was inspected; and response times were measured. Mean time between failure rates were also calculated to estimate software reliability. In order to clearly identify deficienceis as software or hardware related, all test reports are examined. Regression testing is performed by selecting a certain number of days from a two week war scenario. The days chosen depend upon the changes incorporated in the new version, the economy of testing, and the availability of resources.

Comments made relative to the operational testing and evaluation process for software included the following:

- The strength of OT&E lies in the evaluation of human factors.
- OT&E should be automated wherever possible.
- The amount of software actually exercised during OT&E is not sufficient.
- OT&E should be conducted to assess the extent of enhancement a system provides to battlefield operations.

RISK ASSESSMENT

For the projects examined, specific software risk assessment procedures are nonexistent. The only exceptions to this occur when critical nuclear related functions are performed by the software. MIL-HDBK 255 (AS) entitled "Nuclear Weapons Systems, Safety, Design and Evaluation Criteria For" describes the assessment conducted (see Section 3.2). In general, if a risk assessment is performed, it is based on the intuitions and past experience of the personnel involved. The primary consideration during this assessment is the success of the mission.

It is perceived that the evaluation conducted for nuclear weapons forces the software to be "good". Personnel familiar with these risk assessment procedures expressed the opinion that the procedures should be applied to all projects, not just those with nuclear implications.

GENERAL COMMENTS - LESSONS LEARNED

The following comments describe "lessons learned" by the individuals interviewed:

- Systems are too complex to attempt to the concurrent development of hardware, operating systems, languages, and applications.
- The expectations of all parties involved in large system developments should not be too high; complexity must be recognized.
- A perfect system cannot be built.
- Determine the allocation of the functional responsibilities to hardware vs. software in advance.
- Improvement is needed in the area of identifying and defining requirements up front.
- Software is very expensive; do not underestimate the cost.
- Be realistic about acquiring software. Allow sufficient time and money for development and testing.
- Money spent on testing is money well spent.
- An enormous amount of unintentional "throw-away" software is produced.
- There is not enough government involvement in the software development process.
- A program's management and testing philosophy needs to take into account the high susceptibility of military systems to change.
- With respect to the potential perturbation of a project during development, either allow none or plan for it.
- The software baseline needs to be established early for changing requirements.

- Allow 25% of the memory for growth.
- Memory requirements will expand at least 50% over original estimates.
- The turnover in personnel creates problems with continuity in the knowledge of the project.
- Get good people involved early and keep them involved.
- Conduct training courses for developers.
- The software development process must be structured.
- Follow the "classical" approach to software development.
- Engineering houses do not necessarily have sophisticated software engineering procedures.
- You cannot legislate standards.
- Be careful when using MIL-STD-1679; make sure the appropriate portions are applied, especially with respect to required documentation.
- Maintain control over the documentation. Require the documentation corresponding to each phase of the software development process prior to the beginning of the next phase. Insist on good documentation.
- Documentation should be produced as the development process progresses, not after the fact.
- Developers cannot be relied upon to produce adequate documentation unless management emphasizes it.
- Documentation is important, but difficult to get on schedule.
- A good technical writer is worth his weight in gold.
- Have plenty of design reviews to allow interested parties to "speak now or forever hold your peace".
- A useful way to define a program's size is by the number of decisions.
- Testing needs to be automated as much as possible.
- Always require module testing.

- Functionally test the software (using simulators, if necessary).

- Do not underestimate the importance of simulations.

- Examine test results.

- Errors which occur most often during testing should be corrected first.

- Decisions must be made with respect to how much testing is necessary and sufficient.

- The real problems with the software are best found in the operational environment.

- Stress the importance of QA and CM.

- When schedule slippages are imminent, software quality assurance is the first effort to be cut.

- Start IV&V early with the cooperation of the prime contractor.

- IV&V contractors may try to create problems where none exist.

- Metrics and risk assessment procedures are needed.

- Even well defined metrics won't guarantee that all concerned parties will evaluate the software consistently.

- When a contractor is in serious trouble, "keep the heat off" so that some work may be accomplished. Too much "help" does not help.

2.3. OT&E AGENCY INTERVIEWS

Each of the Military Services has an organization which has been given the mission to operationally test and evaluate new and modified systems. These OT&E Agencies are the Army's Operational Test and Evaluation Agency (OTEA), the Navy's Operational Test and Evaluation Force (OPTEVFOR), and the Air Force Test and Evaluation Center (AFTEC).

Since the testing which is performed is operational testing of systems, the software is usually singled out on an exception basis only. However, due to the special section in DoDD 5000.3 on the test and evaluation of software which states, for example, that "...OT&E Agencies shall participate in software planning and development to ensure consideration of the operational environment and early development of operational test objectives", groups which specialize in software have been formulated within each organization.

The software specialists, in some cases, are involved with the development of new systems from the time of conception. They attend the Computer Resource Working Group meetings, Preliminary and Critical Design Reviews, and may even witness acceptance testing. In addition, the Software Evaluation element of AFTEC has developed a set of handbooks for use when evaluating the operational effectiveness and suitability of the software.

Informal meetings were held with software specialists from each of the OT&E agencies. The information gathered is presented here.

OTEA (Army)

OTEA is in the process of staffing a "Methodology and Software Testing Section". At the time of our meeting, the group consisted of six software specialists. Due to the limited number of personnel, it is impossible for the OTEA software specialists to examine all systems under development.

Under ideal conditions, the first involvement of the OTEA software specialists with a new program will be to supply a representative to the Computer Resources Working Group. The primary benefit to be gained from this early, and continued, involvement is a thorough familiarity with the system, its requirements, and its operation. This includes gaining insight into the development of the software system and acquiring a thorough understanding of the software functions and interface requirements.

Early involvement enables the identification of necessary software operational testing aids (i.e., simulators and stimulators, etc.) while there is still time for their development and use. Monitoring the development of a software system enables OTEA to better identify the areas that should be highlighted during OT and make appropriate recommendations as to whether or not the software is sufficiently mature to be properly evaluated in an operational environment. Finally, detailed knowledge of the software allows the operational tester to identify software-unique OT requirements and develop more effective test scenarios so that the system can be exercised at or near the limitations which are "built-in" to the software. Examples of software-unique OT requirements are software/hardware monitors, software instrumentation, and data reduction programs.

OT&E is conducted to estimate a system's operational effectiveness and suitability. In order to locate errors which were introduced during the translation of user requirements into system specifications, OTEA bases its testing on the user's needs. Software OT test objectives are normally subordinate to system operational effectiveness and suitability test objectives. In any case, however, software subobjectives and low level data requirements do appear in test planning documents.

The assessment of the system's operational effectiveness includes an assessment of the functional performance of the fielded software, i.e., how well the software assists the battlefield system in accomplishing its mission in the context of the operation and organization. Therefore, OTEA attempts to include scenarios that exercise the software throughout its typical performance envelope. When limited resources preclude stressing critical software functions, simulators may be used to enhance the real environmental testing. In addition to data gathered during OT, contractor test data and IV&V data, when available, are used by OTEA to evaluate system software performance. Questions addressed during this evaluation include:

1) Do the critical system functions implemented in the software perform as required?

2) Does the software perform satisfactorily when operated at saturation level?

3) Does the software perform satisfactorily in degraded modes of operation typically expected in tactical circumstances?

4) Do the software recovery procedures sufficiently restore the operational effectiveness of the system?

5) Are the software functions compatible with the operational concepts, tactics, and doctrine?

The system suitability assessment includes an examination of the software's suitability and supportability. Software suitability is determined by how well the fielded system interacts with the using personnel and other systems in performing the battlefield system's primary mission. Software supportability is determined by how fielded support software and equipment affect the capability of Army personnel to operate and maintain the operational software in a timely manner.

Software "test data" which results from the assessments described above is both quantifiable and non-quantifiable. OTEA consolidates all available data according to operational issues when evaluating the impact of the software on the operational effectiveness and suitability of the system.

OPTEVFOR (Navy)

OPTEVFOR's philosophy toward operational test and evaluation is that of testing the system - not the software. Consideration must be given to the software, human factors interfaces, and the hardware; but, none of these areas should be isolated during testing. Therefore, no attempt is made to ensure the execution of the software's critical paths during OT&E.

In order to gain knowledge and understanding of how the system is built, OPTEVFOR representatives attend Preliminary and Critical Design Reviews. Prior to OPTEVFOR acceptance of a software system for evaluation (OPEVAL), the developer must demonstrate that the software is, in fact, mature enough for OT&E. This demonstration consists of a TECHEVAL which requires that the software be tested as stated in MIL-STD-1679 (see Section 3.2).

The purpose of OT&E is to determine the system's effectiveness and suitability. All testing performed by OPTEVFOR is scenario based. During OT&E, software errors are identified for correction only. Software reliability is estimated when assessing system effectiveness; however, the model used is the same mean time between failure model which is applied to hardware. In addition, the definition of software maintenance, used when assessing system suitability, is the amount of time it takes, after a failure, for the system to become available for normal operation (i.e., the amount of time it takes to reload all files).

AFTEC (Air Force)

Air Force Regulation 23-36 assigns AFTEC the mission to manage the Air Force OT&E program according to Air Force policy. In addition, AFTEC plans, directs, controls, evaluates, and reports on OT&E and recommends OT&E policy to HQ USAF. One AFTEC responsibility is to observe, or take part in, the early development of major and other HQ USAF-designated systems. To accomplish this task, AFTEC personnel sometimes reside in system program offices or contractor facilities. AFTEC is also involved in the preparation of the Test and Evaluation Master Plans and attends specific acquisition program reviews.

The purpose of OT&E is to evaluate system capabilities in light of operational requirements and concepts. AFTEC's position on the OT&E of software is that, when software is present, it is an integral part of the overall system and must be evaluated in that context. However, due to the unique nature of software and the difficulty of uncovering software problems, it does require special emphasis. The primary problem when planning for the OT&E of software is the determination of the extent of special attention needed and the identification of areas which should be evaluated independent of the system.

The OT&E of software conducted by AFTEC consists of three phases: test preparation, test conduct, and test evaluation. Test preparation begins early in the system life cycle and includes the development of test plans with the assistance of the implementing, using, and supporting agencies. The entire test design/test planning function is an evolutionary, iterative process which involves the definition, evaluation, and refinement of test objectives, measures of effectiveness, and test methodology along with the associated test resources. Test conduct involves the preparation of detailed test procedures, scheduling of day-to-day activities, on-the-scene management of test events, and the preparation of the final test report. This phase includes both in-plant and on-site testing. The evaluation phase consists of test data analysis and evaluation, and report preparation. As with OTEA, development testing results are also considered during the final evaluation.

Two test teams are formed for each system evaluated by AFTEC: the Headquarters Test Team Element and the Field Test Team. The former is responsible for all phases of test design, test planning, and overall test management. When appropriate, the Headquarters Test Team Element includes a software test manager selected from the Software Branch. On the average, software test managers support three to four different test programs. The Field Test Team is responsible for actual test conduct in accordance with the approved test plan. The Field Test Team includes a Deputy for Software Evaluation who is concerned with effectiveness from the user's point of view and suitability from the supporter's point of view. The Deputy for Software Evaluation also directs the software evaluators and is responsible for ensuring that all software test objectives are completed and that test results are reported.

AFTEC's software evaluation focuses on the following areas:

(1) Software Performance - As stated previously, the software performance evaluation is conducted in the context of the overall system performance. Test scenarios are defined to stress known or suspected weak spots in the system design. When software problems arise during OT&E, they are evaluated in terms of the extent of system degradation caused.

(2) Software Operator-Machine Interface - This evaluation is concerned with the interactions between the operator and the computer. The assessment procedure involves the use of standardized questionnaires and will be discussed in detail below.

(3) Software Maintainability - This evaluation assesses the quality of the computer program code and supporting documentation in terms of the ease with which changes can be made. As with the software operator-machine interface evaluation, the assessment employs standardized questionnaires and will be discussed below.

(4) Support System Effectiveness - This evaluation is intended to determine whether or not the software support system (i.e., support software, equipment, and documentation) effectively supports the maintenance team. Since support systems are rarely available during OT, the assessment is usually based upon the evaluators' subjective opinion of the planned support system. Efforts are currently underway to develop a methodology and tools to replace the subjectivity of the evaluation with objectivity.

The primary tools available to the AFTEC software evaluators are standardized questionnaires, the event trace monitor, and Independent Verification and Validation.

AFTEC is a strong advocate of Independent Verification and Validation (IV&V). A recent Air Force Policy letter states that, "Consideration will be given to the use of IV&V in new acquisitions and for retrofit or modification of existing systems". The preferred source for the accomplishment of IV&V is the designated support organization. This arrangement would help eliminate the waste of expertise gained on a system when performing an IV&V. Criteria for the extent of IV&V necessary include safety, mission essentiality, technical risk, supportability, cost/schedule impact, and security. Possible uses of IV&V data and results by AFTEC are to identify critical paths which should be exercised during OT&E and to identify suspected weak spots in the software. To avoid problems with the IV&V contractors, OT&E requirements should be specifically identified in the IV&V contract.

The event trace monitor is a test tool which can be used during OT to monitor the processing performed by a computer and record the occurrence and time of key events. This allows a determination of the amount of reserve processing time available for future enhancements and the verification that a system failure is not about to occur due to stressed operating conditions. It can also be used to aid the process of identifying sources of failures which occur during OT&E (i.e., hardware malfunctions vs. software errors).

The Computer/Support Systems Division of the Test and Evaluation Directorate at AFTEC has prepared a set of handbooks for use when operationally testing and evaluating software. This set of handbooks consists of the following volumes:

Volume I: Software Test Manager's Handbook

Volume II: Handbook for Deputy for Software Evaluation

Volume III: Software Maintainability Evaluator's Handbook

Volume IV: Software Operator-Machine Interface Evaluator's Handbook

Volume V: Computer Support Resources Evaluator's Handbook

Much of the material presented thus far is contained in these documents. We will now discuss the use of standardized questionnaires to evaluate the software's maintainability and operator-machine interface as described in Volumes III and IV of the handbooks.

Software maintainability is defined to be the ease with which programmers/analysts can change software, whether it be to correct errors, add or delete system capabilities, or incorporate modifications made necessary by hardware changes. The evaluation of software maintainability is based on the use of closed form questionnaires which determine the degree of existence of desirable attributes of the code and documentation (a separate questionnaire exists for each). The attributes are each associated with a software characteristic which is believed to increase the maintainability of the software. These software characteristics are modularity, descriptiveness, consistency, simplicity, expandability, and instrumentation.

The software documentation (design, testing, and maintenance documents) is evaluated for content and format. The content evaluation answers the question, "Has the software been designed for maintainability?" The format evaluation answers the question, "Does the organization of the documents aid in the communication of information?" The software source listing evaluation assesses each selected module's source listing and the consistency between the source listings and related documentation. It is important to realize that not all module source listings are evaluated; the software test manager selects a reasonable number of representative modules for evaluation. The separate evaluations are then consolidated to perform an overall assessment. Since evaluations are conducted on both the software documentation and selected source code modules, potential maintainability problems can be identified as to their location (code vs. documentation), the characteristic involved, or a combination of the two.

The questionnaires used for this evaluation are contained in the "Software Maintainability Evaluator's Handbook". Only one question is presented per page along with the identification of the characteristic whose presence is being measured, any necessary explanations, examples, a glossary, and special response instructions. In actuality, the questions are not really questions at all; they are statements of the existence of desirable attributes.

The response scale for the questions consists of answers ranging from completely disagree (which is assigned the value of 1) to completely agree (which is assigned the value of 6). In this way, subjective opinions of the evaluators are quantified. All of the evaluators answers are averaged on each question, thereby providing the basis for a statistical evaluation. The averages are next grouped according to the characteristic they measure and another average score is calculated. Finally, the average scores are multiplied by preassigned relative weights and summed to arrive at an overall maintainability score for the documentation or source code, as appropriate. Any of the average scores described can be compared to preset evaluation criteria (thresholds or goals), in order to identify potential problem areas. If the thresholds are stated as contractual requirements, software may be returned to the developers for improvement.

The use of the questionnaires involves four phases: planning, calibration, assessment, and analysis. During the planning phase, the evaluator team, characteristic weights, and evaluation schedule are established. The calibration phase consists of each evaluator completing one documentation and one module source listing questionnaire. The completed questionnaires are then examined for possible areas of misinterpretation. The function of the calibration phase is to ensure that each evaluator has a clear understanding of the questions on each questionnaire and any response guidelines. The assessment phase consists of the evaluators updating their calibration phase questionnaires and completing the remainder of the questionnaires. In the analysis phase, the averages described above are computed for use in the final evaluation.

The following are sample software documentation questions taken from the "Software Maintainability Evaluator's Handbook". The software characteristics being measured are identified within the parentheses.

Major parts of the documentation are essentially self-contained (format modularity).

The program control flow is organized in a top down hierarchical tree pattern (processing modularity).

Each physically separate part of the documentation includes a useful table of contents (format descriptiveness).

Timing requirements for each major function of the program are explained in the documentation (constraints descriptiveness).

The processing done by each module is explained in the documentation (module descriptiveness).

Program initialization and termination processing is explained (external interface descriptiveness).

There is a useful set of charts which show the general program control and data flow hierarchy among all modules (internal interface descriptiveness).

The documentation on each complex mathematical model includes information such as a derivation, accuracy requirements, stability considerations and references (math model descriptiveness).

The format of the documentation reflects the organization of the program (format consistency).

It appears that programming conventions have been established for the interfacing of modules (design consistency).

The documentation is physically organized as a systematic description of the program from levels of less detail to levels of more detail (format simplicity).

The documentation indicates that each program module is designed to perform only one major function (design simplicity).

A numbering scheme has been adopted which allows for easy addition or deletion of narrative parts of the documentation (format expandability).

The program has been designed so that functional parts may be easily added or deleted (design expandability).

There is a separate part of the documentation for the description of a program test plan (format instrumentation).

The documentation describes a standardized set of program test data (input and output) that has been designed to exercise the program (design instrumentation).

Overall, it appears that the characteristics of the program documentation contribute to the maintainability of the program (general question).

The following are sample module source listing questions taken from the "Software Maintainability Evaluator's Handbook". Once again, the software characteristics being measured are identified within the parentheses.

The concepts of structured programming have been applied to the control structures in this module (data/control modularity).

This module performs only related functional tasks (processing modularity).

The purpose of this module is described in a preface block (preface block descriptiveness).

Imbedded comments describe each function (block of code) within this module (imbedded comments descriptiveness).

The module code is indented within control structures to show control flow (implementation descriptiveness).

The module's flow chart represents the logic control flow as shown in this module's source listing (external consistency).

Global variables are distinguishable from local variables by a naming convention (internal consistency).

Esoteric (clever) programming is avoided in this module (general coding simplicity).

Each physical source line in this module contains at most one executable source statement (singular coding simplicity).

The number of expressions used to control branching in this module is manageable (size simplicity).

Constants used more than once in this module are parameterized (general expandability).

It appears that functional parts could be easily inserted, deleted, or replaced within this module (processing expandability).

This module contains checks for possible out-of-bound array subscripts (processing instrumentation).

Intermediate results within this module can be selectively collected for display (control of instrumentation).

Overall, it appears that the characteristics of this module's source listing contribute to the maintainability of this module (general question).

The software operator-machine interface evaluation is performed to determine the adequacy of the attention given to the design of that part of the system which involves the interaction between the computer-driven system and its operator. The methodology employed is the same as that used for the evaluation of the software's maintainability. In this case, the characteristics being measured by the desirable attributes detailed on the questionnaire are assurability, controllability, workload reasonability, descriptiveness, consistency, and simplicity. Multiple operator-machine interfaces may be assessed using this procedure, however, the evaluators should have experience operating the system prior to the evaluation.

The following are sample questions taken from the "Software Operator-Machine Interface Evaluator's Handbook". The software characteristics being measured are again identified within the parentheses.

Operator input errors do not cause system failures (assurability).

The operator can interrupt and resume automatic processes (controllability).

The system may be operated without reference to manuals during normal operations (workload reasonability).

The machine gives the operator decision aids if tasks cannot be executed as ordered (descriptiveness).

Operator entered commands are systematically formatted (consistency).

Each new message contains only one idea to which the operator must respond (simplicity).

Overall, it appears that the operator-machine interface has been well-designed (general question).

Some limitations identified by AFTEC with respect to their evaluation of software during OT&E are that there is little assurance that critical functions are exercised; the deficiencies which are discovered are very expensive to correct due to the time in the development cycle when OT&E is performed; and, finally, there is a shortage of software engineers available to conduct the evaluations.

2.4. DEVELOPMENT ORGANIZATION INTERVIEWS

This section reports the findings of the interviews which were conducted with industrial representatives involved in the development of software for the government. These results are organized according to whether the software developed is applications software or support software. This differentiation is made due to the different environment for usage and the varying criticality of errors.

2.4.1. DEVELOPMENT ORGANIZATION INTERVIEWS - APPLICATIONS SOFTWARE

OVERVIEW

In this section, the results obtained by interviewing six of the twelve industry contacts will be discussed. The software developed by these contractors is primarily embedded or mission critical applications software as opposed to support software. Three of the corporations represented are large systems houses; two are large computer manufacturers; and one is a software house. Each works primarily on military systems, whether for the United States or for foreign governments. Customers represented include the Army, the Air Force, the Navy, NATO, NASA, DCA, and numerous others. In some cases the software being developed is basically an upgrade to systems developed in the past; in others, the applications are brand new. Some interviews centered upon specific military projects; others were discussions of the "typical" software development process used by that contractor. Therefore, though the sample interviewed was relatively small, the information gathered pertains to a wide variety of software development efforts.

The programming languages used by these contractors are as shown in Figure 1. The amount of Assembly Language used varied from less than 5 percent to a maximum of 30 percent.

LANGUAGE	CONTRACTOR					
	1	2	3	4	5	6
ALGOL			X			
CMS2	X	X		X	X	
FORTRAN		X	X	X	X	
JOVIAL				X	X	X
PASCAL			X	X	X	
SPL1	X				X	
Assembly Language	X	X	X	X	X	X

Figure 1: Programming Languages used by Applications Software Developers

A variety of operating systems are being used for development, both batch oriented and timesharing. In some cases, card inputs are being used rather than terminals.

The development and target hardware being used is illustrated in Figure 2.

MACHINE	ORGANIZATION					
	1	2	3	4	5	6
AN/UYK-7	T					
AN/UYK-20		T			X	
AN/UYK-77					X	
ROLM 1602, 1666		X				
IBM 370						H
IBM 3032, 3033	H	H				
VAX 11/780			H	H		
Others	T		X	X	X	T

H = host machine, T = target machine, X = both

Figure 2: Hardware used by Applications Software Developers

Special purpose hardware used for testing includes numerous simulators, both government furnished and those developed internally to aid in the testing process for a specific project. Also included are load drivers, logic state analysers, signal synthesizers, test target generators, and interface testers.

MILITARY STANDARDS

The MIL-STD's invoked for each of the contractors surveyed are illustrated in Figure 3.

MIL-STD	ORGANIZATION					
	1	2	3	4	5	6
483		X	X	X	X	X
490		X	X	X	X	X
1521A		X			X	
1679 *	X	X		X	X	X
52779A		X				

* or previous standard

Figure 3: Military Standards Invoked for Applications Software Developers

The subjects of these military standards are:

MIL-STD-483 (USAF)	-	"Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs"
MIL-STD-490	-	"Specification Practices"
MIL-STD-1512A	-	"Technical Reviews and Audits for Systems, Equipments, and Computer Programs"
MIL-STD-1679 (Navy)	-	"Weapon System Software Development"
MIL-S-52779A	-	"Software Quality Assurance Program Requirements"

See Section 3.2 for details on these military standards.

Although these standards are invoked, it should be kept in mind that it is a common and necessary practice to tailor standards for each specific project that they are applied to. Therefore, extreme care must be used when making generalizations about the actual constraints being placed upon the software development process.

The primary strengths of the military standards, in the view of the contractors interviewed, are that the discipline applied to the software development process is much better today than it was 10 years ago and the fact that MIL-STD-1679 addresses software testing and quality. The problems encountered when trying to work under the constraints of the military standards and suggestions for improvement are as follows:

- There is not enough money to implement the standards properly.
- The interpretation of government and industry as to what the standards really require is not consistent.
- MIL-STD-1679 needs to be less restrictive; there is no flexibility when problems arise.
- The amount of documentation required should be dependent upon the size of the project and the applications involved.
- The level of detail required by the Data Item Descriptions needs to be defined.
- All program packages should include a Version Description Document.
- The requirements for the test plans are not flexible enough.
- There is too much redundancy in the test plans.
- It is not the customer's job to dictate methodology.
- The allocation of functions to hardware vs. software should be up to the contractor. No customer approval should be required.
- There is little emphasis on development testing to ensure progress; the emphasis is on completed projects.
- MIL-STD-490 should not be applied to working papers.
- MIL-STD-1521A doesn't discuss how to resolve action items.
- The Using Command is not involved enough in the procurement process; the Procuring Agency doesn't know what the users really want. Suggestion - assign a "friend" in the Using Command to help define the real requirements.

Other comments made by the contractors referred to the difficulties encountered when doing functional flow diagrams for asynchronous systems and the preference to organize Data Base Descriptions according to functional groups rather than generic groups.

INTERNAL STANDARDS

Each of the contractors interviewed has internal standards for software development. All use top-down development and structured programming whenever possible. Two of the contractors require formal internal waivers for any projects not compliant with the internal standards. Two of the other contractors determine the standards that will be used on a project by project basis. One of the deciding factors concerned with which standards will be used is the programming language for the project under consideration.

Other internal standards relate to:

- structured design and analysis
- the use of PDL's
- code reviews or inspections
- documentation practices
  (i.e. preambles and comments within the code)
- naming and labelling conventions
- coding practices
  (i.e. defining parameters rather than "hard coding" actual numbers)

In one instance, the preambles for each module of code are actually the Program Design Specification and detailed design. For realtime software applications, one contractor requires that the design be batch oriented so that the majority of the developers do not have to be concerned with interrupts. The same contractor requires that executives handle all I/O operations.

In each case, the means to enforce the internal standards includes the participation of a Quality Assurance organization. Four of the contractors' QA groups attend all code reviews; the remainder either perform their own reviews or do "spot checks" to ensure compliance. Other reviewers may include section leaders, Configuration Management personnel, and IV&V organizations.

SOFTWARE DEVELOPMENT LIFE CYCLE

The common elements in each of the contractors' software development life cycles are:

- software requirements specification and analysis
- software design specification and analysis
- coding and analysis
- module testing
- integration testing (software with software)
- acceptance testing

In addition to the testing described above, two of the contractors described a level of testing prior to module testing; we will call this pre-module testing. In the cases where the software is developed on a machine other than the target machine, software/hardware integration testing must also take place. Furthermore, for four of the contractors, a level of testing called software system testing takes place after integration testing is complete.

Five of the contractors use the build or incremental approach to development. The other has used parallel development but plans on using sequential development in the future. In one instance the program design specification was not being updated for new capabilities.

DOCUMENTATION ITEMS

All of the contractors are required to deliver software requirements specifications, software design specifications and test plans, procedures, and reports for at least one level of testing to their customers. The names of these documents vary depending upon who the customer is. The software requirements specification is known as the Program Performance Specification (PPS), the B5 specification, or the Part I specification. The software design specification is known as the Program Design Specification (PDS), the C5 specification, or the Part II specification.

Other deliverables described but not common to all of the contractors include:

- concept papers and studies
- system requirements documents
- development plans
- documents reporting the allocation of requirements to software vs. hardware
- Quality Assurance and Configuration Management plans
- Program Description Documents (PDD)
- Data Base Design Documents (DBD)
- Interface Design Specifications (IDS)
- test completeness reports
- error analysis and categorization reports
- operators' manuals
- users' manuals
- programming maintenance manuals

One contractor has an internal requirement for unit development folders which include the requirements, design description, functional capabilities list, unit code, unit test plans and results, problem reports, and comments.

REQUIREMENTS ANALYSIS

Requirements analysis of one form or another is performed by all of the contractors interviewed. The methods employed are:

- studies, simulations, and prototyping done by requirements engineers with good systems analysis background and experience with similar systems.

- reviews where the requirements are analyzed for completeness, correctness, consistency, traceability, and testability.

- inspections by system engineers, software developers, testers, and Quality Assurance personnel.

- Technical reviews where the system operational design is traced into the software requirements document combined with a formal presentation to the customer, test group, and Quality Assurance.

- the development of a requirements matrix for use as a database to track development (one of the contractors which uses this technique has experimented with two automated tools to aid in the requirements specification process).

Two of the contractors referred to the problems of writing software requirements. One observed that with or without the help of automated tools, no one seems to be able to write "good" requirements. The other described difficulties in differentiating between what should be in the software requirements document and what should be in the software design document.

DESIGN ANALYSIS

The Preliminary Design Review (PDR) and Critical Design Review (CDR) are formal government reviews which are required for all of the contractors interviewed. These reviews may be conducted by the customer, the contractor, or both. Attendees include the contractor's project team and representatives from the customer's project office. Other attendees may include the contractor's test team and Quality Assurance group and representatives from the customer's user command, OT&E Agency, IV&V organization, and logistics and training groups. The preliminary design is baselined after PDR: the detailed design after CDR.

Prior the the formal design reviews, each of the contractors performs an internal design analysis. Techniques used are:

- inspections conducted by trained moderators, in accordance with internal standards, who choose team members from system engineering, the development organization, the independent test team, and the Quality Assurance organization (Fagan IO inspections). These inspections include an analysis of units consistency and are conducted for both the high level design and the detailed design.

- peer design reviews using the requirements matrix developed previously to trace the requirements into the design.

- peer design reviews with selected managers present.

- informal technical reviews conducted by the developers and Quality Assurance reviews where any outstanding action items are resolved. Units consistency is checked at section leader reviews.

- dry runs for formal customer reviews.

Other activities include database and interface coordination meetings and desk checking.

Three of the contractors interviewed build software prototypes to help refine the requirements and/or design. In one case, modeling and simulations are used during the requirements phase to aid in this respect.

CODE ANALYSIS

Four of the contractors interviewed perform code analysis prior to testing. The variations employed follow.

- Code inspections including reviews of module test plans and procedures. The integration test team attends these reviews and may refuse to accept the code if it is not satisfied with the module test.

- Peer group code reviews including a review of the module test plans. In some cases error checklists are used.

- Weinberg method using error checklists.

- Informal code inspections and walkthroughs with supervisors; formal with the Quality Assurance organization. The extent of formality or the lack thereof depends on the application, complexity, and size. Also performs desk checking.

Two of the contractors use code analysis on a selective basis. One uses peer or analyst reviews when module testing doesn't seem sufficient. These reviews concentrate on checking the logic rather than coding techniques. Desk checking is also used. The other contractor has made code inspections optional due to the close relationship between the code and the detailed design. Design inspections are always performed by this contractor.

COMMENTS ON ANALYSIS ACTIVITIES

All of the contractors felt that the analysis activities just described have a positive influence on the quality of the software produced. Their comments and concerns follow.

- Testers should be involved in requirements reviews.

- Management should not attend inspections.

- There is a tendency to rush through inspections when there are schedule difficulties.

- Design reviews become instructional sessions when the participants have not reviewed the documents in advance.

- Inspections should be postponed if the participants are not adequately prepared.

- A more diligent application of the review process is needed.

- Formal customer reviews should be incremental rather than program wide.

- Dry runs for formal customer reviews are more beneficial than the actual reviews.

One contractor is currently trying to determine how detailed a design should be and whether or not code inspections are necessary when the correspondence between the design and the code is approximately one-to-one.

PRE-MODULE TESTING

Two of the six contractors interviewed performed pre-module testing. In both cases, the goal was to exercise all paths at this level. One contractor determined that this was not cost effective and no longer strives for this coverage.

MODULE TESTING

Four of the contractors began their preparations for module testing during the design phase. In all cases the developers write test plans and procedures or keep a log of the testing performed. It should be noted that in one case, module testing was combined with module integration testing.

Five of the six contractors require that the extent of module testing be reviewed for adequacy by someone other than the developer. The reviewers may be section leaders, integration test teams, or project managers. The reviews may take place at design reviews, code inspections, or audits. In one case, the customer required test reports for selected modules.

The testing philosophies and criteria used can be summarized as follows:

- Only one contractor performs module tests as black-box tests. In this case, all requirements must be tested using nominal and extreme values.

- For the contractors that use the white-box methodology: one requires that a "reasonable" number of branches be exercised, one measures statement coverage to find any "blatant" areas that are missed, two require 100% branch coverage, and one strives for complete path coverage.

- Four of the contractors exercise the code with nominal values, minimums, maximums, extreme values, and invalid inputs.

- Two contractors use module testing to test any critical timing and performance requirements.

- One contractor uses calibration bugs occasionally, but not as a regular practice.

INTEGRATION TESTING

The testing performed when integrating the software modules into a software system is known by numerous names. Some of these are module integration tests, subprogram tests, subsystem tests, and build tests.

An opinion that all of the contractors have in common is that the key to integration testing is to get the best people on the integration test team. Two of the contractors use independent test teams; the membership of the remainder of the teams is as follows:

- 3 or 4 lead programmers and the programmers responsible for the modules being integrated

- a software requirements specification expert, a software design specification expert, and the section leaders.

- a few "key" people

One of the contractors is in the process of creating a skill center which will be responsible for integration and testing, configuration management, and facilities management.

Preparations for integration testing begin during the requirements phase for two contractors and the design phase for three of the contractors. One contractor described integration testing as "initially ad-hoc". Four of the contractors require formal test plans and procedures for integration testing. In one case, the test plan must be approved by the customer. In that same instance, the test plan and procedures are also reviewed by the internal project manager.

In all cases, the software is tested for functional correctness. The tests are based on the software requirements document and performed in a scenario fashion. If an Interface Design Specification is available, it is also a basis for integration testing. In one case, the tests performed are the same as those used for software system testing. The final result of integration testing, for two of the contractors, is the acceptance test. In one case, once a "full-up" system has been achieved, capacity testing is performed.

Four contractors use a top-down testing strategy; one uses bottom-up; and, one uses a combination of the both. Three of the contractors test both nominal and off-nominal or extreme inputs in the process of integration testing.

Four of the six contractors use the same people and tests to integrate the software system with the hardware. In one case, system engineers perform this testing. Two contractors use simulators for the hardware until the actual hardware is available. Both noted that the sooner the real hardware is used, the better it is for the development process.

SOFTWARE SYSTEM TESTING

Software system testing, also called program performance testing or verification testing, is performed by four of the six contractors prior to acceptance testing. In all cases, this level of testing is performed by a test team independent of the developers. Preparations are begun during the requirements or design phase. Test plans, procedures, and reports are required for software system testing. In one case, the tests are developed by the independent test group and reviewed by the engineers, developers, and Quality Assurance personnel. Another contractor holds reviews with the customer to determine the adequacy of the tests and resolve any disagreements. Three contractors use a test/requirements matrix to ensure that all of the requirements are tested during software system testing. All tests include nominal and off-nominal scenarios. One contractor described this testing as having "no holds barred" - the testers start with the specified requirements, then test for the real world including failure mode testing.

ACCEPTANCE TESTING

As with the other levels of testing, the contractors have assigned their own names to what we will call acceptance testing. The names used include quality testing, formal qualification testing and formal functional/performance testing.

In all cases, the test plans and procedures must be approved as adequate by the customer. The preparations for acceptance testing begin during the requirements phase. The organizations responsible for writing and conducting the acceptance test are different for each contractor interviewed. The acceptance tests may be:

- written and conducted by customer's IV&V organization with contractor's support. (The IV&V organization writes the test plan; the contractor writes the test procedures.)

- written by contractor, conducted jointly by customer and contractor.

- written by contractor's independent test team, conducted by customer.

- written and conducted by contractor's independent integration test team.

- written and conducted by contractor's system test people.

- written by contractor's software system test people, conducted by Quality Assurance organization.

The contractors' Quality Assurance organizations always witness acceptance tests.

The criteria for acceptance tests are that they test every testable requirement and demonstrate every function after complete integration. One contractor described the acceptance test as a "super integration test". Two of the contractors use Validation Cross Reference Matrices to control the acceptance testing process. Fault testing is also conducted at this time.

In one case, acceptance testing is made up of reliability testing, software functional verification testing, and a final test which is a subset of the other tests. The duration of the total acceptance testing process is between 3 and 4 weeks with the final test lasting approximately 50 to 60 hours. The reliability testing takes 1 to 2 weeks. During this time the full hardware/software set is exercised for at least 200 hours. The amount of testing performed depends upon the required reliability. This also determines the amount of time that the system will be exercised under overload conditions. A Mean Time Between Incident (MTBI) is measured over the 200+ hours. The minimum requirement for MTBI is 24 hours.

TESTING TOOLS

The only tools in general use to aid the testing process are simulators, models, data reduction tools, and standard debug tools. The majority of these tools are project specific. In isolated instances, the following types of tools are also being used:

- standard file comparators or output comparators

- general cross-reference generators

- static analyzers, path analyzers, or code auditors

- test file/data generators (project specific)

- dynamic execution verifiers

- symbolic testers

One contractor is trying to combine the tools used within its organizations into a test data management system for Fortran 77.

METRICS

No metrics were found in use.

APPLICATION SPECIFIC TESTING TECHNIQUES

The only application specific testing technique found is that of inserting faults into the hardware to test Built-in-Test software.

COMMENTS ON SOFTWARE TESTING

The following comments on software testing were made by the contractors interviewed.

- A more scientific methodology for software testing is needed.
- Test procedures are too detailed. They should be written from an engineering point of view.
- More stringent rules for test plans and procedures are needed. These should be spelled out in RFP's.
- More money is needed for Quality Assurance so that standards and procedures can be better enforced.
- Interactive test drivers/harnesses are needed.
- Independent test teams are needed. Unfortunately, testing is not a career programmers like.
- Succeess is dependent on the preparation of the testers.
- Configuration management and good documentation are very important during testing.
- There is currently no good way to keep up when the customer keeps changing the requirements.

TEST DOCUMENTATION PROCEDURES

In general, formal test plans and procedures are kept under configuration control. When updates to the software become necessary due to changing requirements or software trouble reports, a software change control board oversees the process. Usually, forms used to request changes also specify which software modules will be effected. This same information is used to update test plans and procedures.

ERROR ANALYSIS AND TRACKING

When software errors are discovered, software trouble reports are submitted to the software change control board described previously. The information submitted includes what errors were found, when they were found, and what modules were involved. The errors are then prioritized according to severity. An example of the priorities used are those defined in MIL-STD-1679 (see Section 3.2). Errors are tracked in terms of the number outstanding, their ages, and their seriousness.

One contractor uses error information to estimate software support requirements since the software produced, in this instance, is maintained forever by the contractor. Error information is also being used to determine the usefulness of design and code inspections. In another case, the frequency of recompilations combined with error information is used to estimate software maturity and determine progress.

REGRESSION TESTING

Four of the contractors interviewed either have or plan to have a standard set of test cases which will be rerun prior to each new release of the software. One contractor intends to do a complete retest of the software system for each delivery. In each case, special tests will be run for all changes made. One contractor uses a combination of engineering judgement and path analysis tools to determine the amount of testing necessary for a given set of changes. Finally, in one case, the regression testing process is largely automated and optimized.

QUALITY ASSURANCE

Each of the contractors interviewed has an independent Quality Assurance organization. In one case, in addition to the corporate Quality Assurance organization, each project is assigned its own QA group which is managed by the project manager but is still independent from the developers. The activities of each of the Quality Assurance organizations are shown in Figure 4.

QA Activities	CONTRACTOR					
	1	2	3	4	5	6
Review Requirements & Design Specs.	X	X			X	X
Review Test Plans and Procedures	X	X		X		
Review All Deliverables		X	X			
Attend Design Reviews	X		X		X	X
Attend Code Reviews	X		X		X	
Witness Acceptance Tests	X	X	X	X	X	X
Perform Final Configuration Audit		X	X		X	X

Figure 4: Quality Assurance Activities for Applications Software Developers

In most cases, QA reviews are conducted for compliance with customer invoked and internal standards. Other activities may include conducting spot checks of documentation when not involved in the actual review process, performing test readiness reviews, monitoring Configuration Management audits, and participating on the software change control board. In one case, QA is responsible for writing test plans and procedures. In another case, a Product Assurance organization supplies the independent test group.

Comments made by three of the contractors interviewed had to do with the fact the QA is basically a non-technical function. These organizations feel that a Quality Assurance organization that understands content as well as form would be much more beneficial. One contractor is experimenting with rotating developers in and out of the QA organization to help in this respect. Another comment made was that QA is usually an understaffed and underbudgeted activity.

INDEPENDENT VERIFICATION AND VALIDATION

Only three of the contractors interviewed have had any experience with IV&V organizations participating in the development process. In each case, the IV&V organization reports directly to the customer. The contractor's responsibilities with respect to IV&V are usually to deliver documentation, etc. to their customers. The customers then forward the items to the IV&V organization. The contractor must also respond to any comments made by the IV&V organizations. In one case, the contractor is required to give the IV&V organization one hour notice for any meetings so that IV&V representatives may attend. The IV&V organization must also be allowed total access to all documentation. In another case with the same contractor, IV&V representatives work on-site during the development process to observe operations for compliance with contractual requirements. In one case, the IV&V organization was involved with the project from the beginning. In the other cases, the initial involvement was during the software requirements phase.

The IV&V activities for the projects discussed above are shown in Figure 5.

IV&V Activities	CONTRACTOR		
	1	2	3
Independent Requirements & Design Analysis	X	X	X
Algorithm Studies	X	X	X
Participation in Code Inspections	X		
Independent Code Analysis	X	X	
Witness Acceptance Tests	X		
Approve Test Plans & Procedures			X
Independent Testing		X	

Figure 5: IV&V Activities for Applications Software Projects

In one case, the acceptance test plans are written and the test conducted by the IV&V organization. Other IV&V activities include witnessing integration and software system tests, independently evaluating software trouble reports, and writing software documentation. In another case, the IV&V organization was responsible for reviewing all deliverable documents.

Concerns were expressed about the relationship between contractors and IV&V organizations. In some cases, contractors feel that IV&V organizations have exaggerated problems to make themselves appear needed. Other comments related to the timing of the initial involvement of the IV&V organizations. It was agreed that the sooner they are involved, the better it is for the project concerned.

OPERATIONAL TESTING AND EVALUATION

The contractors.' involvement in Operational Testing and Evaluation (OT&E) of the systems produced varies from no involvement, to explaining any problems encountered with the software, to helping plan and evaluate the tests conducted.

OT&E Agency involvement in the development process varies from no involvement to attending all customer reviews to performing requirements analysis, conducting design reviews, and witnessing acceptance tests.

Contractors' comments with respect to OT&E are:

- OT tends to be a subset of the contractor's testing.
- OT is conducted at the system level and is not too interested in or concerned with testing software.
- OT is not thorough enough in testing man/machine interfaces.

RISK ASSESSMENT

In most cases, risk assessment consists of analysts and/or systems engineers defining critical functions, etc., based on past experience and intuitive knowledge. One contractor performs failure modes and effects analyses and, based upon the results, recommends code changes and tests to the developers and test organizations. In some cases, simulation and modeling takes place to determine the risk involved with specific algorithms. Prototypes are also built to reduce risk. Based upon the perceived risk involved, more design reviews, code reviews, and testing may take place. One contractor stated that "unofficial importance is given to risk by the people who write the test procedures".

One contractor requires formal proofs of correctness when the software design is "too complex". As a result, the developers redesign until the proofs are no longer required. Another contractor requires formal proofs of correctness for nuclear release mechanisms.

NEW TECHNOLOGY TRENDS

Asynchronous systems: One contractor tries to design asynchronous systems such that there aren't any time criticalities.

Signal processing and data flow machines: Contractors expressed a need to know how to test these technologies.

Firmware: What amount of testing, etc., must be performed when firmware is involved in a system? This is an area of major concern.

Software Testing: One contractor is currently teaching a course for software testers and conducts workshops to promote the transfer of information.

Ada: Two of the contractors interviewed use Ada as a PDL. Two others are planning to do so in the near future. A group in one of the contractors' software engineering area is currently teaching Ada classes. This contractor is now waiting for a project to impose the use of Ada. This is expected in the 1984 to 1985 timeframe. Comments concerning Ada included:

- Ada should help testability.

- Ada should result in an increase in the quality of code.

- Ada will help, but standard ISA's are also needed.

LESSONS LEARNED

The following is a list of "lessons learned" which the contractors interviewed wished to pass on.

- Think through all of the steps required in advance. More planning up front means fewer problems downstream.

- Make sure you have the right tools for software development. It is very helpful to have a compiler, etc., on a mainframe for development purposes.

- Hardware is not usually delivered ontime. When it is delivered, it may not work as expected. Plan for these possibilities when determining schedules.

- Allow time for education.
- Management should interact...not react.
- It is important to have systems people who understand software.
- Find problems early.
- Establish a requirements matrix early on. Identify unreasonable requirements and suggest alternatives.
- ASK QUESTIONS!!!
- It is very important to baseline the software requirements prior to design and the software design prior to coding.
- Use top-down design techniques.
- Eliminate or minimize parallel development.
- Use simple architectures. Many small problems are better than one large problem.
- "Nail down" interfaces.
- Manage at subroutine level - work the details.
- Developers need to feel involved in the total development effort. One way to accomplish this is for each developer to maintain his code "forever".
- Every requirement is a test requirement.
- Code is nothing until it is executed.
- Solve long term problems rather than short term problems (i.e., no patches).
- Recognize the merit of software testing:
  -- get the right tools, test environments, simulators
  -- get good people for testing
  -- keep track of progress - get early visibility
- Controlled testing is very important.
- Make sure people are adequately prepared for testing.
- Keep test teams a reasonable size.
- Independent test groups improve the quality of software.

- Configuration Management is very important to the testing effort.
- Customer interaction and user visibility is important.
- Customers have an idealized view of software development. They must be educated to the fact that things will go wrong!!!

SUMMARY OF SOFTWARE TEST AND EVALUATION ACTIVITIES

SUMMARY	CONTRACTOR					
	1	2	3	4	5	6
PROGRAMMING LANGUAGES						
ALGOL			X			
CMS2	X	X		X	X	
Fortran		X	X	X	X	
JOVIAL				X	X	X
PASCAL			X	X	X	
SPL1	X				X	
Assembly Language	X	X	X	X	X	X
HARDWARE (T = target, H = host, X = both)						
AN/UYK-7	T					
AN/UYK-20		T			X	
AN/UYK-77					X	
ROLM 1602, 1666		X				
IBM 370						H
IBM 3032, 3033	H	H				
VAX 11/780			H	H		
Others	T		X	X	X	T

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR					
	1	2	3	4	5	6
MIL-STD's						
483		X	X	X	X	X
490		X	X	X	X	X
1521A		X			X	
1679	X	X		X	X	X
52779A		X				
REQUIREMENTS ANALYSIS						
Studies, Simulations, and Prototyping					X	X
Reviews	X	X		X		
Requirements Matrix			X		X	
DESIGN ANALYSIS (Internal)						
Inspections	X					
Peer Reviews					X	X
Informal Reviews		X				
QA Reviews		X				
Dry Runs for Formal Reviews			X	X		X

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR					
	1	2	3	4	5	6
CODE ANALYSIS (0 = optional)						
Inspections	X	X		0		
Walkthroughs		X				
Peer Reviews					X	0
Desk Checking		X				X
Review Module Test Plans and Procedures	X				X	
PRE-MODULE TESTING		X				X
MODULE TESTING						
Adequacy Review	X	X	X	X	X	
Black-box	X					
White-box		X	X	X	X	X
Extreme Values & Invalid Inputs	X		X	X	X	X
INTEGRATION TESTING						
Top-Down	X	X	X	X		X
Bottom-Up				X	X	
Nominal & Off-Nominal Scenarios	X		X			X

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR					
	1	2	3	4	5	6
S/W SYSTEM TESTING	X	X		X		X
Use Test/Requirements Matrix	X	X		X		
ACCEPTANCE TESTING	X	X	X	X	X	X
Use Validation Cross Reference Matrix					X	X
REGRESSION TESTING						
Standard Test Set	X		X		X	X
Complete Retest		X				
Selective Testing	X		X	X		
TESTING TOOLS						
File or Output Comparators			X	X		
Cross Reference Generators			X			
Static Analyzers, Path Analyzers, or Code Auditors			X	X	X	X
Test File/Data Generators				X		
Dynamic Execution Verifiers			X	X		
Symbolic Testers				X		

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR					
	1	2	3	4	5	6
QA ACTIVITIES						
Review Requirements & Design Specs.	X	X			X	X
Review Test Plans and Procedures	X	X		X		
Review All Deliverables		X	X			
Attend Design Reviews	X		X		X	X
Attend Code Reviews	X		X		X	
Witness Acceptance Tests	X	X	X	X	X	X
Perform Final Configuration Audit		X	X		X	X
IV&V ACTIVITIES						
Ind. Requirements & Design Analysis		X		X		X
Algorithm Studies		X		X		X
Participation in Code Inspections		X				
Ind. Code Analysis		X		X		
Witness Acceptance Tests		X				
Approve Test Plans & Procedures						X
Independent Testing				X		

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR					
	1	2	3	4	5	6
RISK ASSESSMENT						
Experience/Intuition	X	X		X	X	X
Failure Modes & Effects Analysis				X		
Modeling & Simulation					X	X
Prototyping				X		X

2.4.2. DEVELOPMENT ORGANIZATION INTERVIEWS - SUPPORT SOFTWARE

OVERVIEW

Two of the industry contacts who were interviewed are primarily involved in the development of support software (i.e. compilers, assemblers, loaders, etc.). The efforts put forth by one of the contractors are directed at implementing enhancements to previous systems. The other contractor is currently in the early detailed design phase for a new system. In addition, some pieces of the new system are being prototyped. In both cases, only the machine dependent code is done in assembly language. This reportedly amounts to less than 5% of each system.

MILITARY STANDARDS

MIL-STD-1679 (Navy) on "Weapon System Software Development" was invoked on one of the contractors for their compiler development effort. The other contractor is required to follow MIL-STD-490, "Specification Practices". For a description of the contents of these military standards, see Section 3.2.

The perceived strengths of the standards are the fact that they address specifications, design, QA, CM, etc. The weaknesses described include:

The standards require an extreme amount of paperwork.

The standards do not describe how to test properly.

The required test plans and procedures are too general.

The standards are vague and ambiguous; some sections are inappropriate.

INTERNAL STANDARDS

Internal standards which are common to both contractors interviewed required top-down development, structured programming, and the use of a PDL based on the appropriate programming language. In addition, one of the contractors requires a waiver for any module which contains more than 300 statements. The same contractor requires that the detailed design be embedded within the code via the use of preambles and comments. In each case, project/team leader reviews are used as a means to enforce the required internal standards. QA spot checks and a PDL processor are other means which are also employed by one of the contractors.

SOFTWARE DEVELOPMENT LIFE CYCLE

One contractor described the software development life cycle very simply as determining the specifications, implementing them, and testing them. The other contractor described the process in more detail. Since the requirements had been well defined prior to that contractor's involvement with the system, the first phase of the life cycle described is concerned with developing the design. Once the Critical Design Review (CDR) is completed, coding begins, followed by code reviews and unit tests. Once integration testing of the Computer Program Configuration Items (CPCI's) is completed, the software is turned over to the Quality Assurance organization for functional testing. If, based on their testing, the QA group accepts the software, subsystem and system tests are conducted.

DOCUMENTATION ITEMS

The documentation items required for delivery include those which were common to all of the applications software developers: the software requirements specifications, the software design specifications, and test plans, procedures, and reports. The support software developers must also deliver documentation items selected from the following list: user handbooks or manuals, operating procedures, reference booklets, technical descriptions, maintenance manuals, and revision descriptions.

REQUIREMENTS ANALYSIS

Requirements analysis is not being performed, to a great extent, by either one of the support software developers interviewed. In one case, this is due to the fact that the requirements for the system being developed underwent extensive scrutiny and review prior to the contractor's involvement with the system. In the other case, the contractor is in the process of implementing enhancements to previously developed systems. Although, the Software Change Proposals (SCP's) include an analysis of the impact the change will incur, in some instances, prototyping may also be performed to ensure that the new requirements are well understood and can be implemented satisfactorily.

DESIGN ANALYSIS

The design analysis performed by the contractor implementing enhancements consists of the project leader interfacing with the responsible programmer. The design analysis for the system undergoing initial development, as would be expected, is more extensive. Informal team design reviews are held. In addition, seminars are conducted for future users of the new system. The developers are usually required to justify their designs at these seminars. Prototypes are also built to determine potential performance and/or usability problems which may require modifications to the design. Finally, the usual government reviews (PDR's and CDR's) are conducted. In this case, the design specification is baselined upon acceptance at the Critical Design Review. For the other contractor, each time a certified revision of the system being enhanced goes under configuration management, the corresponding design is baselined.

CODE ANALYSIS

As with design analysis, the code analysis performed by the contractor implementing enhancements consists of informal reviews involving the project leader and the responsible programmer. The level of formality applied is dependent upon the experience of the programmer. One purpose of these reviews is to ensure compliance with the internal coding and documentation guidelines. The other contractor's development teams will perform code reviews on each Computer Program Component (CPC). In this case, the experienced developers will be relied upon to direct attention to areas where common problems may exist.

COMMENTS ON ANALYSIS ACTIVITIES

One of the contractors questions the usefulness of the analysis activities described thus far. In this contractor's opinion it is the proficiency of the programmers that makes the difference in terms of the quality of the code produced. The other contractor believes that the analysis activities are very valuable. They allow you to discover errors while it is still relatively inexpensive to correct them. The weaknesses discussed pertain to the government reviews. It is felt that the military and IV&V personnel involved in the PDR's and CDR's need more time than is currently allocated to prepare for these reviews if they are to be genuinely beneficial to the software development process.

MODULE TESTING

The purpose of module testing is to test the specifications and the error conditions. An objective of one of the contractors is to achieve 100% statement coverage during module testing. The other contractor will record control flow analysis data for information only. The amount of testing actually performed will be left up to the discretion of the responsible programmer. In one case, the modules will be instrumented to aid in the testing process.

INTEGRATION TESTING

The integration testing which will be discussed only pertains to the plans of the contractor developing the new system. For a discussion of the testing performed by the contractor implementing enhancements, see the section on Regression Testing.

Preparations for the integration testing which will be performed by this contractor began prior to contract award. The test plans which will be used were outlined in the proposal which won the contract. The integration testing will consist of functional tests based upon the software requirements and design specifications. The compiler will be instrumented to record information necessary to determine whether performance objectives have been met. A bottom-up testing strategy is planned.

ACCEPTANCE TESTING

The purpose of acceptance testing for these contractors is to certify their respective compilers and systems for government use. In one case, the test plans and procedures are written by either the QA or IV&V organization and reviewed by the developers. It should be noted that, in this case, the original set of test cases used to certify the compiler were developed as a separate task on the contract. In addition, a set of tests was furnished by the government. For each new release of the system, the original test set is enhanced with new test cases which address the specific changes to the system. Upon completion of the certification tests, which are conducted by the contractor, the tests reports are sent to the customer who then approves the new version of the system for release.

The contractor who is developing the new system will have one responsibility with respect to acceptance testing, that being the responsibility to turn the system over to the customer. The acceptance testing will then be performed by the customer with the help of another contractor.

TESTING TOOLS

In addition to the normal Operating System tools which are available to aid in the software development process, the tools found in use by the support software developers were standard file comparators or output comparators and dynamic execution verifiers. One of the contractors also uses simulators of the target machines during the testing process.

The contractor developing the new system plans to develop a test executor for use during regression testing. The test executor will choose which tests to run, from a standard test set, based upon which modules have been modified. It is hoped that these tests will be selfchecking. The other contractor has a standard JCL run stream which accomplishes this same task (in this case, the tests are selfchecking).

METRICS

No metrics were found in use.

TEST DOCUMENTATION PROCEDURES

As with the applications software developers, the support software developers are required to document their tests via plans, procedures, and reports. The process to request changes to the system is a formal one. The forms used specify what areas of the system will be affected by the change and the expected impact on the cost, schedule, documentation, and users of the system. These same forms are used to trigger the appropriate changes to the test sets. The contractor who is in the mode of implementing enhancements to the existing system has organized the test cases in the standard test set according to components and function to facilitate the necessary changes.

ERROR ANALYSIS AND TRACKING

Errors in the existing system, as well as priorities for the correction of the errors, are communicated to the developers via software trouble reports. A patch log is used to document changes made to the system between releases. The reported errors are categorized according to the guilty components to aid the developers in their efforts to find "soft spots" in the system. The contractor developing the new system has no formal plans for error analysis and tracking. The developers speculated that errors in that system will probably be categorized in terms of whether they are logic errors, interface errors, etc.

REGRESSION TESTING

The major difference between the testing performed by the applications software developers and the support software developers is the degree of automation used during regression testing. The support software developers either have or plan to have an extensive set of test cases for use during regression testing. The decision whether to run the entire set or a subset of the test cases prior to the release of the modified system is based on the scope of the changes which were implemented. Due to the use of specialized testing tools, very little human intervention is needed when running these tests or when checking the results (see the section entitled Testing Tools).

QUALITY ASSURANCE

Each of the contractors interviewed has an independent Quality Assurance organization. Both of these organizations review all documentation produced. One specifically checks for readability, consistency, and level of detail - not technical correctness. The other checks for compliance with military standards, etc. This same group also ensures that code reviews are conducted as required. Another activity which is common to both organizations is the writing of test plans and procedures; in one case, these are then reviewed by the developers. One QA group is responsible for tracking the status of testing and determining the sufficiency of the tests via comparison with the specifications. In addition, one of the QA organizations has subsumed the Configuration Management function.

A reported strength of one of the groups is that its personnel are good developers. This is in contrast to the complaint by the other interviewees that the QA group is not technical enough. Another reported weakness is a lack of manpower due to a lack of support from the customer.

INDEPENDENT VERIFICATION AND VALIDATION

Independent Verification and Validation organizations support the customers of each of the contractors interviewed. In one case, the IV&V organization was selected at the same time as the contractor. The other contractor feels that the IV&V organization for their effort came on board too late and that the project schedules will probably impact the amount of IV&V accomplished. One of the IV&V organizations performs a review function only. This takes the form of attending PDR's and CDR's and reviewing all documentation and test reports. In addition to performing requirements analysis and design and code reviews, the other IV&V organization also writes test plans, specifications, and procedures and conducts independent tests.

The general attitude of the contractors toward IV&V is that if it begins early enough it can reveal errors in designs, etc. and documentations allowing the implementation of corrections prior to any significant repercussions. One problem which was experienced with IV&V testing, however, was the discovery of errors in the tests rather than the software as was initially suspected.

OPERATIONAL TESTING AND EVALUATION

Rather than using the customer's OT&E Agency for operational testing and evaluation prior to release, one contractor uses friendly user beta test sites. These test sites communicate directly with the developers and any comments are incorporated into the new system prior to general distribution. In some instances, the test sites issue software trouble reports as an alternative to direct communication. In addition to testing the systems, the beta test sites also review any manuals which are to be distributed with the system. The informality of this type of OT&E is perceived by the developers as a benefit in terms of productivity.

The other contractor's expected alternative to formal OT&E is to perform the final 6 months of development using the system which is being developed.

RISK ASSESSMENT

As with the applications software systems, risk assessment for the support software systems is built on a foundation of past experience and devoid of formal procedures. Areas of concern may surface during design discussions, etc., thereby requiring special attention from that point on. In one case, the amount of testing on the various modules of the system is directly related to the complexity of the modules, thereby reducing the risk. In another case, prototypes may be built for especially risky areas of the system under development.

NEW TECHNOLOGY TRENDS

One of the contractors is using a program generator to produce required tests based on the ISA's of the target machines.

LESSONS LEARNED

The "lessons learned" by the support software developers are equally applicable to other software development efforts. They are:

- Customer participation is very important to the development effort. Development should not take place in a "dark tunnel".
- The role played by the contract is also very important. Make sure that any uncertainties with respect to contractual requirements are resolved early.
- Efforts to produce deliverables in a limited amount of time without reviews and/or feedback are futile.
- Allocate time and money for testing up front.
- Begin planning for testing with the development effort.
- Testers must be as proficient as the developers.
- Test baselines must be built upon. The completeness of the test sets must be maintained.

SUMMARY OF SOFTWARE TEST AND EVALUATION ACTIVITIES

SUMMARY	CONTRACTOR	
	1	2
STATUS		
Detailed Design		X
Enhancements	X	
MIL-STD's		
490		X
1679	X	
REQUIREMENTS ANALYSIS		
Prototyping (for special cases only)	X	
None (this was completed prior to contract award)		X
DESIGN ANALYSIS (Internal)		
Project Leader Reviews	X	
Informal Peer Reviews		X
Prototyping & Seminars		X
CODE ANALYSIS		
Project Leader Reviews	X	
Peer Reviews		X

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR	
	1	2
MODULE TESTING		
White-Box	X	X
Objective of 100% Statement Coverage	X	
Control Flow Analysis (for information only)		X
Program Instrumentation		X
INTEGRATION TESTING		
(See Regression Testing)	X	
Bottom-Up		X
Functional		X
Program Instrumentation		X
ACCEPTANCE TESTING		
Performed by Contractor	X	
Performed by Customer & Independent Contractor		X
REGRESSION TESTING		
Standard Test Set	X	X
Entire Retest or Subset Based on Scope of Changes	X	X
Specialized Text Executors	X	X
Self-Checking Tests	X	X

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR	
	1	2
TESTING TOOLS		
File or Output Comparators	X	
Dynamic Execution Verifiers	X	X
Test Program Generators	X	
Test Executors	X	X
QA ACTIVITIES		
Review Requirements and Design Specifications	X	X
Write Test Plans & Procedures	X	X
Review All Deliverables	X	X
Ensure Proper Code Reviews Are Conducted		X
Track Testing Status	X	
Perform CM Functions		X
IV&V ACTIVITIES		
Independent Requirements, Design, & Code Analysis	X	
Attend PDR's & CDR's		X
Independent Testing	X	
Review All Documentation & Test Reports		X

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR	
	1	2
OT&E ACTIVITIES		
Use Beta Test Sites	X	
Use System Under Development for Development (final 6 mos.)		X
RISK ASSESSMENT		
Experience/Intuition	X	X
Prototyping	X	X
Test Relative to Complexity	X	

2.5. IV&V ORGANIZATION INTERVIEWS

OVERVIEW

Independent Verification and Validation (IV&V) is a risk reducing technique which is applied to many major programs under development today. The results reported in this section were obtained by interviewing four industry contacts whose primary function is to conduct an independent evaluation of the software being produced by another contractor. Due to the high cost of IV&V, the activities which will be described are usually only performed for a portion of any software system.

The IV&V contractors surveyed report directly to the project office for the systems under development. In two of the cases to be discussed, initial involvement with the projects occurred during either the system or software requirements specification phase. In one instance, the high-level design was completed prior to a prime contractor being chosen. The IV&V contractor was hired one month after the prime contract was awarded. For the remaining IV&V contractor, initial involvement with the project of interest occurred well into the software development cycle. All of the projects discussed make extensive use of either embedded or mission critical computer resources. The customers represented include the Army, Navy, and Air Force. In one case, the discussions centered upon the IV&V activities being performed for a tri-service project.

The programming languages used for the systems under development included CMS2, Fortran 77, JOVIAL, PASCAL, PL/1, and Assembly Language. One quarter of one of the systems discussed was coded in Assembly Language. This was the maximum percentage of Assembly Language encountered.

MILITARY STANDARDS AND REGULATIONS

Although there are no military standards in existence specifically for application to IV&V efforts, it is important for IV&V organizations to be aware of those that relate to the software development process in general. Therefore, it is not surprising that the MIL-STD's mentioned by the IV&V contractors are the same ones which were discussed with the development organizations (see Section 2.4).

In addition, since an IV&V contractor may, in some instances, perform the role of a technical contract monitor, cognizance of the relevant military regulations is also helpful. Those regulations referred to by the IV&V contractors interviewed follow.

Air Force:

AFR 800-14, Volume I	-	"Management of Computer Resources in Systems"
AFR 800-14, Volume II	-	"Acquisition and Support Procedures for Computer Resources in Systems"

Army:

DARCOM Regulation 70-16	-	"Management of Computer Resources in Battlefield Automated Systems"

Navy:

SECANAVINST 5200.32	-	"Management of Embedded Computer Resources in the Department of the Navy Systems"
TADSTAND A	-	"Standard Definitions for Embedded Computer Resources in Tactical Digital Systems"
TADSTAND B	-	"Standard Embedded Computers, Computer Peri]herals, and Input/Output Interfaces"
TADSTAND C	-	"Computer Programming Language Standardization Policy for Tactical Digital Systems"
TADSTAND D	-	"Reserve Capacity Requirements for Tactical Digital Systems"
TADSTAND E	-	"Software Development, Documentation, and Testing Policy for Navy Mission Critical Systems"

For a discussion of the contents of these regulations, see Chapter 3.

Again, it should be noted that in most cases, the regulations and standards are tailored for the specific system under development. For the tri-service project, the Computer Resources Working Group (CRWG) addresses any commonality or life cycle issues which arise between the individual services.

The perceived strengths of the regulations and standards centered on MIL-STD-1679's methodology for software development. The primary weakness mentioned was concerned with the lack of tri-service standards. It was felt that MIL-STD-SDS will be of great benefit to tri-service programs.

DOCUMENTATION ITEMS

In all cases, evaluation reports of various types are supplied to the Project Office by the IV&V organizations. These may be document review reports, algorithm study reports, review reports (for design reviews, etc.), test reports, problem reports, and/or status reports. In one case, the test plans and procedures developed by the IV&V contractor are submitted to the customer for approval.

REQUIREMENTS ANALYSIS

All of the IV&V contractors analyze the software requirements for the system under development. The methods employed include the following:

- The software requirements are developed independently and then compared with those of the development contractor. Alternatively, a "classical" review of the software requirements may be conducted.

- The system specification is traced into the software requirements specification to ensure that "nothing has fallen through the cracks". In addition, interface analysis studies, mathematical accuracy studies using simulation, and man/machine interface studies with the Using Commands are also conducted.

- Individuals review the requirements independently and then collaborate on the evaluation to be forwarded to the Project Office. In this case, traceability receives the most attention, although some consistency checking is also performed.

- A requirements check matrix is developed to aid in tracking the software development process. An informal analysis based on prior experience and knowledge of potential pitfalls is also conducted.

Three of the IV&V contractors use automated tools to aid in the process of analyzing the requirements. The tools mentioned include the AFFIRM Specification and Verification System and PSL/PSA. Another tool used is an automatic requirements tracing tool. This tool is basically a database information system which is used to enter requirements and "pointers" to the modules where each requirement is implemented, etc. Although the requirements and "pointers" are entered manually, the system does alert users to missing "pointers". One IV&V contractor also uses a listing processor which prints requirements documents, etc. allowing room for comments. A comment was made that requirements analysis tools don't necessarily help; it is the knowledge of what the requirements really are that makes a difference.

The development contractors interviewed describe problems encountered when trying to determine the level of detail which should be present in a software requirements specification. One of the IV&V contractors made the following observation: Systems engineers don't have the detailed knowledge of computers which is necessary to write software requirements. Computer scientists have that knowledge; however, when computer scientists write software requirements, they tend to write at a level of detail such that the requirements, in actuality, are the design.

DESIGN ANALYSIS

Each of the IV&V contractors also conducts some type of design analysis. The following describes the techniques which are used.

- The design specification is defined independently using PSL/PSA for comparison with the design prepared by the development contractor. Alternatively, a "classical" review of the design specification may be performed.

- The design is evaluated for traceability, consistency, and feasibility. Math and logic analyses are also conducted. (These analyses are performed without the benefit of modeling.) Depending upon the application, an analysis of units consistency may also take place. In addition, critical algorithms are derived independently for comparison with those of the development contractor. In this case, the analyses are conducted independently by individuals who then collaborate on the final evaluation to be forwarded to the Project Office.

- The design specifications are reviewed for traceability. In addition, the IV&V organization conducts independent design walkthroughs where the analysts act out the roles of the designers. A review of the preliminary documentation is also performed. Occasionally, this IV&V contractor sends representatives to participate in the development contractor's internal design walkthroughs and technical interchange meetings.

- The IV&V contractor witnesses the design walkthroughs conducted by the development contractor to ensure that proper review procedures are being followed. The participants in these walkthroughs are the designer, the designer's co-workers, one management representative, and the IV&V contractor's representatives. In some instances, the IV&V contractor may also perform its own independent design reviews.

In addition to the design analysis activities described above, each of the IV&V contractors also participates in the formal government reviews which are conducted (i.e., the Preliminary Design Review and the Critical Design Review).

CODE ANALYSIS

For three of the four IV&V contractors interviewed, code analysis is one of their "standard" tasks. The other IV&V contractor only gets involved in this when there are major problems. Then, code analysis is performed primarily to check for inefficiencies in the coding techniques used. The IV&V contractors which do perform code analysis on a regular basis described the following activities.

- First of all, the code is traced back into the design specification. Code inspections and walkthroughs are conducted to ensure that maximum levels of nesting are not exceeded, no unreachable code exists, etc. In some cases, critical algorithms are coded independently to perform accuracy checking.

- The code is evaluated using metrics. This will be described in more detail later.

- The code is analyzed for understandability.

COMMENTS ON ANALYSIS ACTIVITIES

The comments made on analysis activities referred to the formal government reviews rather than the IV&V activities. It was felt that:

- Early reviews should focus away from computer resources, thus allowing more flexibility in future activities.

- A weakness of the government review process is the lack of reviews between the Critical Design Review and final acceptance. A formal method to review progress during the implementation and testing phase is needed.

- The reviews described in MIL-STD-SDS are an improvement over what currently exists.

INDEPENDENT TESTING

All of the IV&V contractors interviewed are involved in the testing process in one way or another. This may be characterized by either conducting independent tests, witnessing the development contractor's tests, or both. The levels of independent testing performed and strategies employed vary from contractor to contractor. The descriptions follow:

- Module level tests or tests on a limited number of integrated modules. The testing of this IV&V contractor is constrained by a simulator which must be used. A prioritized list of modules to be tested is prepared during the design phase. The tests performed include minimum values, maximum values, and illegal inputs. Error guessing is another source of test inputs. Objectives of these tests include executing every option and every branch of the modules tested. Complete statement and branch coverage are required for mission critical functions.

- Functional testing of complete software systems. Test inputs include both operationally realistic and implementation dependent critical values.

- Software system level tests. The tests are functional tests whose requirements are derived from the software requirements. Worst case scenarios are emphasized. All tests are scenario based unless specific problems are being investigated. In that case, generic testing may be used.

In addition to witnessing the development contractor's tests, one of the IV&V contractors also reviews the development contractor's test plans and procedures. Another IV&V contractor described the testing activities as 50% independent testing and 50% evaluating the development contractor's tests. This was felt to be sufficient for IV&V. In addition, or as an alternative to independent testing, one of the IV&V contractors performs a monitoring role to enforce the use of pre-specified testing techniques. This IV&V contractor advocates an aggregate statement coverage measure of at least 85% with explanations of why the remainder of the statements in the total software system were not exercised.

TESTING TOOLS

In addition to the tools which are used for requirements analysis, one of the IV&V contractors uses the following types of testing tools: simulators, file comparators, code auditors, and units consistency analyzers. Another IV&V contractor uses dynamic execution verifiers.

METRICS

In April 1980, the Rome Air Development Center published two reports on software metrics. They were entitled "Software Quality Metrics Enhancements" and "Software Quality Measurement Manual". The metrics framework described in those reports is being used by one of the IV&V contractors as a basis for further study. One of the objectives of this study is to determine the usefulness of the metrics framework. In addition, efforts are being made to tie the metrics together with cost estimates for the purpose of doing tradeoff analyses. The quality metrics framework has been tailored for a specific project and is being applied to both the code and the documentation. PSL/PSA is being used to gather some of the data needed for the metrics calculations. It is hoped that these metrics can be used to answer some of the questions which are posed by the Defense System Acquisition Review Council (see Section 3.6).

TEST DOCUMENTATION PROCEDURES

The test documentation procedures employed by the IV&V contractors are basically the same as those used by the development contractors, though the level of formality may be somewhat reduced. The requirement for human intervention during testing is still a problem in terms of automating the process of regression testing. As was the case with the development contractors, updates to "standard" regression test sets are usually a fallout of traceability efforts.

ERROR ANALYSIS AND TRACKING

Only one of the IV&V contractors described any involvement in error analysis and/or tracking. That contractor uses the same type of program trouble reports as the development contractors to perform an informal categorization on functional errors.

OPERATIONAL TESTING AND EVALUATION

One of the IV&V contractors has been approached by the Operational Test and Evaluation Agency of the customer to provide technical support during OT&E. The responsibilities of this contractor will be to recommend specific test cases for OT&E and aid in the resolution of any problems which may be encountered.

RISK ASSESSMENT

As was mentioned previously, due to the high cost of IV&V, the activities described are usually only performed for a portion of any software system. Therefore, it is obvious that some type of risk assessment must be conducted to determine the scope of any IV&V applied to a project. In many cases, it is the IV&V contractor who determines which areas of a software system warrant special attention. Description of these analyses and the effect of the results, follow.

- The functions of the software system as defined in the software requirements specification are assessed in terms of technical, schedule, and/or other risks and assigned a criticality rating. This criticality rating determines whether a portion of the software system will be tested independently, reviewed, or completely ignored by the IV&V contractor. The criticality criteria used are defined on a project by project basis.

- Modeling and simulations are performed for critical functions. Independent testing is conducted on a prioritized list of the software modules. Complete statement and branch coverage are required during independent testing on all mission critical modules.

- The applications of the software system are studied to determine which are the most important so that appropriate actions may be taken during the IV&V process.

NEW TECHNOLOGY TRENDS

An IV&V contractor who is working on a multiprocessor, asynchronous, realtime system looks for worst case scenarios to apply during testing. Significant efforts are also put into modeling and simulation.

The IV&V contractors interviewed felt that the principle of commonality underlying Ada and the APSE is good. It is also felt that Ada should make a difference in the quality of the software produced. One of the contractors did comment, however, that any idea of "outlawing" Assembly Language is unrealistic.

A problem which concerns the IV&V contractors, as well as the development contractors, is that of firmware. Requirements for the definitions of interfaces with, and the testing and documentation of firmware need to be established.

Another problem mentioned relates to the standard Instruction Set Architectures which were recently defined. The concern is that these standards are not consistent with the Navy standard hardware.

LESSONS LEARNED

The "lessons learned" by the IV&V contractors include the following:

- IV&V should begin as soon as possible on any given project.
- RFP's should include a clause concerning independent evaluation. This should not be added as an afterthought.
- User involvement in the initial phases of system development is very important.
- The life cycle planning document should be finalized prior to full scale development to protect the life cycle from compromises which may be made for the ease of implementation.
- The transition of a program from single service to tri-service is not trivial.
- Customers are always optimistic in terms of schedules, costs, etc.

SUMMARY OF INDEPENDENT VERIFICATION AND VALIDATION ACTIVITIES

SUMMARY	CONTRACTOR			
	1	2	3	4
TIME OF INITIAL INVOLVEMENT				
Requirements Phase	X	X		
One Month after Prime Contract Award			X	
Well into Development Cycle				X
REQUIREMENTS ANALYSIS				
Independent Derivation of the Software Requirements				X
System Specification Traced into Software Requirements			X	
Reviews for Traceability and Consistency	X			
Interface Analysis Studies			X	
Mathematical Accuracy Studies			X	
Man/Machine Interface Studies			X	
Individual Reviews, then Collaboration for final Evaluation	X			
Requirements Check Matrix and Informal Analysis		X		

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR			
	1	2	3	4
REQUIREMENTS ANALYSIS TOOLS				
PSL/PSA				X
AFFIRM		X		
Requirements Tracing Tool			X	
Listing Processor		X		
DESIGN ANALYSIS				
Independent Derivation of the Software Design				X
Independent Derivation of Critical Algorithms	X			
Evaluation of Traceability, Consistency, and Feasibility	X			
Reviews for Traceability			X	
Math and Logic Analyses	X			
Analysis of Units Consistency	X			
Independent Design Walkthroughs		X	X	
Participation in Development Contractor's Design Walkthroughs		X	X	
Technical Interchange Meetings			X	
Individual Reviews, then Collaboration for Final Evaluation	X			

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR			
	1	2	3	4
CODE ANALYSIS				
Independent Coding of Critical Algorithms			X	
Evaluation using Metrics				X
Inspections and Walkthroughs			X	
Code Traced into Design			X	
Review for Inefficiencies	X			
Review for Understandability		X		
INVOLVEMENT IN TESTING				
Independent Testing	X	X	X	
Software System Level Testing	X	X		
Module and/or Integration Testing on Prioritized List of Modules			X	
Worst Case Scenarios	X			
Operationally Reslistic & Implementation Dependent Critical Values		X		
Extreme Values, Invalid Inputs, and Error Guessing			X	
Statement and Branch Coverage			X	
Evaluate Development Contractor's Testing	X	X	X	X

STEP - Current Defense Practices Overview

SUMMARY	CONTRACTOR			
	1	2	3	4
TESTING TOOLS				
Simulators			X	
File Comparators			X	
Code Auditors			X	
Units Consistency Analyzers			X	
Dynamic Execution Verifiers		X		
MISCELLANEOUS ACTIVITIES				
Error Analysis			X	
Involvement in OT&E			X	
RISK ASSESSMENT				
Experience/Intuition	X	X	X	
Criticality Ratings	X			
Modeling and Simulations			X	

CHAPTER 3

MILITARY STANDARDS AND GUIDANCE

OVERVIEW

In this section, we will describe the current military standards and guidance. In addition, modifications planned for the near future will also be discussed.

The Department of Defense issues directives and instructions to the Office of the Secretary of Defense (OSD), the Military Departments, the Organization of the Joint Chiefs of Staff (JSC), and the Defense Agencies. These directives and instructions provide guidance and uniformity of thrust, which the separate military Services may tailor, supplement or amplify for their own particular applications, including more detail as appropriate. The military standards are used as requirements on contracts by program managers. The military Services' regulations and standards are requirements on members of those services.

Documents were chosen for inclusion here based on their applicability to the issues addressed in the Software Test and Evaluation Project, or when they were recommended by the interviewees as of use or applicability to the pertinent programs. Included in this summary of regulations and standards are Air Force Regulations (AFR), Army Regulations (AR), Navy Standards, and summaries of other documents related to military efforts toward regulating, delineating, describing, and proscribing procedures necessary for the entire software life cycle for Embedded Computer Resources (ECR) and/or Mission Critical Computer Resources (MCCR).

3.1. DEPARTMENT OF DEFENSE DIRECTIVES AND INSTRUCTIONS

DoDD 5000.1: Major Systems Acquisitions. 19 March 1980.

This Directive applies to the Office of the Secretary of Defense, the Military Departments, the Organization of the Joint Chiefs of Staff, and the Defense Agencies. Each DoD official who has any responsibility for the acquisition process shall make "every effort" to ensure that an effective and efficient acquisition strategy is developed for each system acquisition program, minimize time from need identification to introduction of the system into operational use, achieve the most cost-effective balance between acquisition costs and system effectiveness, and integrate support, manpower, and related concerns into the acquisition process.

The provisions of this Directive shall apply to the acquisition of systems designated as major, as well as others, where appropriate. As a part of routine planning, DoD Components shall conduct analyses to identify deficiencies in capability or more effective means of performing assigned tasks.

The designation of a system as major is based on development risk, urgency of need, estimated requirement for the system's research, development, test and evaluation (RDT&E), or Congressional interest. Affordability must be considered at every milestone. To proceed to the Demonstration and Validation phase, the DoD component must assure that it plans to acquire and operate the system and that sufficient RDT&E resources are available to complete development. To proceed into Full-Scale Development, and the Production and Deployment phases, the DoD component must assure and reaffirm that resources are available to complete development and acquisition, and to operate and support the deployed system.

Acquisition of equipment satisfying DoD component needs should also include consideration of intraservice and interservice standardization and interoperability requirements. Logistic supportablility shall be a design requirement as important as cost, schedule, and performance. Milestones 0, I, II, and III are defined, and the following documentation for Milestone Decisions is described:

- The Mission Element Need Statement (MENS);
- The Decision Coordinating Paper (DCP);
- The Integrated Program Summary (IPS);
- The Milestone Reference File (MRF):
- The Secretary of Defense Decision Memorandum (SDDM).

These documents are referenced and described in the review of DoDD 5000.2.

The Defense Systems Acquisition Review Council (DSARC) is defined and described. Elsewhere in this document (see Section 3.6), the publication, "Embedded Computer Resources and the DSARC Process", is reviewed. The Defense Acquisition Executive (DAE) is explained. This Directive and DoD Instruction 5000.2 are first and second in order of precedence for major system acquisitions except where statuatory requirements override.

DoDD 5000.2: Major System Acquisition Procedures. 19 March 1980.

This Instruction applies to the same DoD Components as DoDD 5000.1, which has the subject of Major System Acquisitions and is used in conjunction with DoDI 5000.2. Specific procedures for major system acquisition include: designation of a major system, listing of major systems, Milestone 0 Documentation, and Defense Systems Acquisition Review Council (DSARC) involvement. The Secretary of Defense designates certain acquisition programs as major systems, which may be recommended by the Defense Acquisition Executive (DAE) at any point in the acquisition process. The DAE may also withdraw the designation of "major systems" when changing circumstances dictate. The Executive Secretary of DSARC shall maintain and distribute a list of designated major systems.

Milestone 0 Documentation consists of the Mission Element Need Statement (MENS), the document upon which the Milestone 0 decision is based, and the Secretary of Defense Decision Memorandum (SDDM). The MENS identifies and defines:

- A specific deficiency or opportunity within a mission area;
- The relative priority of the deficiency within the mission area;
- The Defense Intelligence Agency validated threat forecast or other factor causing the deficiency;
- The date when the system must be fielded to meet the threat;
- The general magnitude of acquisition resources that the DoD component is willing to invest to correct the deficiency.

It should be noted that a MENS is not required for programs, regardless of size, directed toward developing and maintaining a viable technology base.

The SDDM is prepared, by the action officer, when the DAE plans to recommend approval of the MENS and designation of a system as major. The SDDM documents the Secretary of Defense's milestone decision including approval of goals and thresholds for cost, schedule, performance, and supportability, exceptions to the acquisition process and other appropriate action. The DAE forwards the SDDM to the Secretary of Defense after formal coordination.

The DSARC acts as the top level DoD corporate body for system acquisition and provides advice and assistance to the Secretary of Defense. DSARC reviews are held at Milestones I, II, and III. Documentation for Milestones I, II, and III includes the Decision Coordinating Paper (DCP), Integrated Program Summary (IPS), and the Milestone Reference File (MRF).

DoD directives, regulations, and instructions that relate to the acquisition process are part of the Defense Acquisition Regulatory System (DARS). The object of this system is to provide detailed functional regulations required to govern DoD acquisition of materials, supplies, and equipment. Program managers must tailor their programs to DoD issuances that are part of DARS.

Special attention in the development of acquisition must be given to the following matters:

- Mission Analysis;
- Operational Requirements;
- Threat;
- Acquisition Strategy;
- Management Information;
- Competitive Concept Development;
- Contracting;
- Design Considerations;
- Reliability and Maintainability;
- Test and Evaluation;
- Logistics;
- Computer Resources;
- Command and Control Systems;
- International Programs: NATO Rationalization, Standardization, and Interoperability.

Although the acquisition strategy developed is not a document requiring DAE approval, the program manager is required to keep all management levels informed on strategy and to summarize certain aspects of it at the milestone decision points (i.e., Milestones 0, I, II, etc.).

Embedded computer resource acquisition must be managed within the context of the total system. Plans for computer interfaces must be identified early in the life cycle, and special attention must be given to plans for software development, documentation, testing, and update during deployment and operation. Computer hardware and software must be specified and treated as configuration items.

Another matter requiring special attention is the topic of Command and Control systems. These systems require unusual management procedures because they have a rapidly evolving technological base, multiple requirements for internal and external interfaces, and a reliance on automatic data processing hardware and related software. These systems differ from other weapon systems in that they are acquired in small numbers, or are one of a kind, and their operational characteristics are largely determined by the users in an evolutionary process. For such systems, acquisition management procedures should allow early implementation and field evaluation of a prototype system using existing commercial or military hardware and software.

The provisions of DoDD 5000.1 and this Instruction are first and second in order of precedence for major system acquisition except where statuatory requirements override. Enclosures for this Instruction include a list of references, sample formats for the MENS, the DCP, the IPS, and a list of DoD policy issuances related to acquisition of major systems.

DoDD 5000.3: Test and Evaluation. 26 December 1979.

Department of Defense Directive 5000.3 addresses the subject of Test & Evaluation (T&E). It establishes policy for the conduct of T&E in the acquisition of defense systems, designates the Director Defense T&E (DDTE) as having overall responsibility for T&E matters within the DoD, defines the responsibilities of DDTE, the organization of the Joint Chiefs of Staff (OJCS) and the guidance for Test and Evaluation Master Plans (TEMPs). The provisions of the Directive apply to the Military Departments, the Office of the Secretary of Defense (OSD), the OJCS, and the Unified and Specified Commands. These provisions encompass major defense system acquisition programs in all responsible DoD components, as well as the management of system programs not designated as major.

This directive requires that Test and Evaluation (T&E) shall begin as early as possible and be conducted throughout the system acquisition process. Before tests begin, meaningful critical issues, test objectives, and evaluation criteria shall be established. Successful accomplishment of T&E objectives will be a requirement for decisions to commit additional resources to a program or to advance it from one acquisition phase to another. To minimize dependence on subjective judgment concerning system performance, appropriate test instrumentation will be used to provide quantitative data for system evaluation.

Development T&E (DT&E) is that T&E conducted to assist the design and development process and to verify attainment of technical performance specifications and objectives. It includes T&E of components, subsystems, hardware/software integration, related software, and prototype development models of the system, as well as compatibility and interoperability with existing systems. DT&E is prescribed during the system acquisition phase before the decision Milestone I, to assist in selecting preferred alternative system concepts; before the Milestone II decision, to identify the preferred technical approach, technical risks and feasible solutions; before the Milestone III decision, to ensure that engineering is reasonably complete, that all design problems have been identified, and that solutions to these problems are in hand; and after the Milestone III decision, for development, acceptance, and introduction of system changes for improvement, new threats, or to reduce life cycle costs. Multiservice DT&E may be required for systems that interface with equipment of another DoD component or that may be acquired by more than one DoD component. "DT&E is normally accomplished or managed by the DoD Component's material development agency."

Operational T&E (OT&E) is that T&E conducted to estimate a system's operational effectiveness and suitability, identify needed modifications, and provide information on tactics, doctrine and personnel requirements. Acquisition programs shall be structured so that OT&E begins as early as possible in the development cycle. Initial operational test and evaluation (IOT&E) must be completed prior to the Milestone III decision. IOT&E must be accomplished, as appropriate, before the Milestone I decision, to assess the operational impact of candidate technical approaches; before the Milestone II decision, to examine the operational aspects of the selected alternative technical approaches and estimate the potential operational effectiveness and suitability of candidate systems. Before the Milestone III decision, adequate OT&E shall be accomplished to provide a valid estimate of the system's operational effectiveness and suitability; and, after the Milestone III decision, follow-on OT&E (FOT&E) must be managed as necessary, to ensure that the initial production items meet operational effectiveness and suitability thresholds.

Multiservice OT&E shall be accomplished when systems have an interface with equipment of another DoD Component or may be acquired by more than one DoD component.

Throughout the system acquisition process, the DoD's component agency shall ensure effective planning during all acquisition phases, participate in system acquisition planning and test design, ensure that OT and DT are sufficient and credible to support analysis and evaluation needs, and include recommendations regarding system readiness for operational use at Milestone III.

DT and OT can be combined "when clearly identified and significant cost and time benefits will result, provided that the necessary resources, test conditions, and test data required...can be obtained". When a combined testing program is chosen, the OT&E agency shall provide a separate and independent evaluation of the test results, in all cases.

For computer software, quantitative and demonstrable performance objectives and evaluation criteria must be established during each system acquisition phase. Testing shall be structured to demonstrate that software has reached a level of maturity appropriate to each phase. For embedded software, these objectives and criteria shall be included in the performance objectives and evaluation criteria of the overall system.

Decisions to proceed from one phase of software development to the next shall be based on quantitative demonstration of adequate software performance through appropriate T&E. OT&E agencies shall participate in the early stages of software planning and development to ensure that adequate consideration is given to the system's operational use.

Each DoD component is required to have a major field agency, separate and distinct from the material developing/procuring agency and from the using agency, that is responsible for managing operational testing and for reporting test results and an independent evaluation of the system under test directly to the Military Service Chief or Defense Agency Director.

The directive specifies that, for one-of-a-kind systems, particularly space, large-scale communications, and electronic system programs, the principles of DT&E of components, subsystems, and prototype models of the system shall be applied. Compatibility and interoperability of these systems with existing or planned equipment shall be tested during DT&E and OT&E. Subsequent OT&E may be conducted to refine estimates and ensure that deficiencies are corrected.

It is specified that the DoD component shall prepare and submit a T&E Master Plan (TEMP) for OSD approval, before Milestone I. The TEMP is a broad plan that relates test objectives to required system characteristics and critical issues. An enclosure to the directive specifies guidelines for the TEMP. The DoD component is required to provide the following information to the Director Defense T&E (DDTE): appropriate test reports when testing has been accomplished, system operational concepts, how tests were accomplished, and test limitations.

When Joint T&E (JT&E) is required by the DDTE, the Joint Chiefs of Staff (JCS) have a requirement for JT&E results that provide information on joint doctrine, tactics, and operational procedures. The JCS can annually nominate exercises for JT&E, as can the Joint Staff, the Military Services, and the Commanders in Chief (CINC) of the Unified and Specified Commands. Control of JT&E will be exercised by the DDTE.

DoDD 5000.29: Management of Computer Resources in Major Defense Systems. 26 April 1976.

Department of Defense Directive 5000.29 addresses the subject of the management of computer resources in major defense systems, establishes a management steering committee for embedded computer resources (MSC-ECR), and establishes policy for the management and control of computer resources during the development, acquisition, deployment and support of such systems. Due to a 1982 expiration date, it is currently being updated. The directive specifically excludes general purpose, automatic data processing (ADP) applications. Since embedded computer resources have a cost measured in the billions of dollars, these resources must be treated as being of major importance, especially with respect to integration with hardware.

The directive specifies that requirements validation and risk analysis must be conducted, that computer resources (HW and SW) will be treated as configuration items, and that a computer resource life cycle plan will be developed and maintained during the life cycle. Support items required to effectively develop and maintain the delivered computer resources, such as compilers, documentation aids, test case generators and analyzers, and training aids, are required as deliverables. The use of High Order Programming Languages (HOLs) where effective or practical is required.

DoD Components are required to review their existing regulations and modify, cancel, or supplement them as necessary to render them consistent with this directive. Furthermore, they are required to maintain guidance documentation for the software life cycle for program managers and other responsible organizations, maintain education, career paths and career incentives to foster development and retention of professional computer resource engineers, managers, and technicians, and plan and execute a research and development program to provide the technological base to support the policy, practice, and procedure requirements of this directive, using the Technology Coordinating Paper.

The DoD management steering committee for ECR (MSC-ECR), formerly named "Weapon Systems Software Management Steering Committee", is chartered to implement the provisions of DoDD 5000.29 and issue ensuing policies. Its objectives are to improve computer resource management (CRM), increase visibility of computer resources in overall acquisitions, formulate a coordinated DoD Technology Base Program for software basic research, development and demonstrations addressing critical software issues, and to "guide the assimilation and integration of computer resources policy, practice, procedure and technology into the normal process of major Defense systems acquisition."

The MSC-ECR is to be composed of representatives from the DoD offices, and representatives of the Army, Navy, Air Force, Office of the Joint Chiefs of Staff, Defense Communications Agency, National Security Agency, Defense Advanced Research Projects (DARPA), and TRI-TAC.

<u>DoDI 7920.2</u>: Major Automated Information Systems Approval Process.
20 October 1978.

The subject of DoD Instruction 7920.2 is Major Automated Information Systems Approval Process. It does not cover Command, Communication, and Control Systems (CCCS)or Embedded Computer Systems (ECS). The purpose of this Instruction is to establish the review and decision process and procedures for major automated information systems (AIS).

It establishes requirements for the system decision paper (SDP), which must be prepared following the approval of the Mission Element Need Statement (MENS), to support DoD Component and OSD reviews, coordination, and decisions before continuation of the AIS development. The SDP process provides for appropriate policy level involvement in key decisions during the life cycle of each major AIS.

The Milestone 0 Decision follows the phase where a mission need is identified, that need is validated, and the exploration of alternative function concepts have been recommended. That decision allows the DoD Component to proceed to identify alternative concepts to satisfy the functional need.

The next phase is the concept development phase, where one or more feasible concepts for further exploration are recommended and alternative methods are synthesized and evaluated. "This phase is completed upon approval at Milestone I to define and design an AIS based upon a selected concept."

The Definition/Design Phase precedes the Milestone II decision. The purpose of this phase is to define fully the functional requirements and to design an operable AIS. This phase is completed when ADP and the telecommunications technical adequacy have been validated, and approval is issued to fully develop the system.

The system development phase is to develop, integrate, test and evaluate the ADP system and the total AIS. At the end of this phase, prior to Milestone III, the following tasks are to be completed: the mission need has been reaffirmed, computer programs and data bases have been fully developed, standardization and interoperability requirements have been satisfied, system support documentation has been developed, and unit and system level T&E results support a decision to proceed with the deployment. The Milestone III decision by the appropriate officials allows deployment of the system at the operating site(s).

System Effectiveness Milestones shall be conducted if required at convenient time periods after the first year of full system operation to determine the continued effectiveness of the system, to identify potential obsolescence, and to certify continued need for the system.

Other DoD publications that are also related to or have had impact on the testing and evaluation of computer software in embedded computer system applications will be reviewed or listed in Section 3.6.

3.2. MILITARY STANDARDS

MIL-STD-483(USAF): Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs. 1 June 1971.

MIL-STD-483 (USAF) addresses the subject of configuration management practices for computer programs, systems, equipment, and munitions. It is specified that practices are to be tailored to specific programs. This standard establishes configuration management requirements that are not covered in MIL-STD's 480, 481, 482, and 490. The scope of this standard includes configuration management, identification, control, audits (functional and physical), interface control, engineering release control, and reports and records. It applies during appropriate system life cycle phases of CIs (configuration items), whether part of a system or an independent CI.

Configuration management is "a discipline applying technical and administrative direction and surveillance to (a) identify ... configuration item characteristics, control changes to these characteristics, and record and report change processing and implementation status". Requirements of this standard include a Configuration Management Plan, Baseline Management, System Engineering, Interface Control, and Configuration Identification (functional, allocated, and product). Functional Configuration Identification is required for all systems and all CIs specified in the contract which are allocated from a system requirement, except privately developed items. Allocated CIs "shall be used to govern the development of selected CIs that are allocated from system requirements or are part of a higher level CI." Product CIs "shall be used to prescribe necessary 'build-to' or form, fit and function requirements and the acceptance tests for those requirements."

Baseline management is formally required at the beginning of an acquisition program. A baseline may be established at any point in a program where it is necessary to define a formal departure point for control of future changes. "System program management normally employs three baselines ... to include the functional, allocated, and product baselines ... Computer program management may employ all three baselines or employ only the functional and product baselines depending upon complexity or peculiar requirements." Furthermore, "All descriptions of baselines ... used to state product performance and design requirements ... must be contained in specifications."

Baselines are the basic requirements from which contract costs are determined. Once defined, changes in these requirements are formally approved and documented to provide an equitable way to adjust contract costs.

System engineering for the total system or a functional area is normally vested in a single contractor or government agency. System engineering as it relates to configuration management, is the application of scientific efforts to transform an operational need into a description of system performance parameters. The system engineering agency generates requirements for configurations which will satisfy the operational need, constrained technically only by the content of the system specification.

The interface control contractor is a coordinator with responsibility to "assure that configuration item identification conforms to the functional interfaces established by system engineering and that the configuration items, including computer programs as finally designed, are physically compatible...and can be operated and maintained as intended".

The formal qualification review (FQR) establishes "that a new design configuration item has satisfactorily qualified to the specification requirements." Procedures for accomplishing "the FQR when the procuring activity requires contractor participation are contained in appendix XII." The FQR "establishes that the configuration item performs in its use environment as required by the CI specification." Where practicable, configuration audits shall be accomplished in conjunction with other audits and reviews. CM records/reports shall ensure that there will be a configuration record documenting all approved changes to all configuration items.

Appendices to this standard give detailed criteria and guidelines for the CM Plan; Establishment of Interface Control; Computer Program Configuration Items (CPCIs); Specification and Support Document Maintenance for Computer Programs; System Allocation Documents; and Configuration Audits.

MIL-STD-490: Specification Practices. 1 February 1969

MIL-STD-490 addresses the subject of specification practices. It establishes the format and contents of specifications for program peculiar (one-of-a-kind) items, processes and materials, and establishes criteria for a uniform specification program for all contractor-prepared documents. It describes:

A. System Specifications
B. Development Specifications
C. Prime Item Product Specifications
D. Process Specifications
E. Material Specifications (such as raw materials, chemicals, electrical cable)

Other specifications described which do not mention computer software explicitly, include the B1 - Prime Item Development Specification, which is applicable to complex items such as aircraft, missiles, radar sets, or fire control equipment, and is used as the functional base line for a single item development program or as part of the allocated base line where the item covered is part of a larger system development program.

Specifications prepared in accordance with this standard are intended for use in the design and procurement of configuration items, computer programs, and required services for program peculiar application. Requirements include configuration identification, functional, allocated, and product configuration identification, and detail and general specifications.

Requirements for specification types are:

Type A - The system specification states the technical and mission requirements for a system as an entity, allocates requirements to functional areas, and defines interfaces between the functional areas.

Type B - Development specifications state the requirements for the design and development of a product during the development period.

Type B5, the Computer Program Development Specification (CPDS) contains a brief description of the overall computer program by major functions (tasks) and a summary of the specification content, composition, and intent. The requirements section is the major section of the CPDS and consists "of a series of paragraphs that specify in detail the performance requirements of the computer program". Program definition, detailed functional requirements, input and output data, processing descriptions, and system parameters and capacities shall be described. The program test plan and procedures at the subprogram and program level must be developed, and computer program acceptance and system integration testing is required. Test requirements at each level of testing, except the acceptance test level, and test tools and facilities required shall be specified. A separate paragraph on acceptance test requirements is required to "establish the means by which the procuring agency may formally accept the computer program as fulfilling the performance requirements."

Type C - Product specifications are applicable to any item below the system level, may be oriented toward procurement of a product through specification of primarily function (performance) or fabrication (detailed design) requirements.

Type C5 - Computer Program Product Specification (CPPS) is applicable to the production of computer programs and specifies their implementing media. The CPPS will include a statement of scope, a brief review of the major functions of the computer program, its structure and function as a whole, description of storage allocation, functional flow diagrams, program interrupts, detailed description of control logic, and any special control features.

Type B5 specs apply to computer program development, C5 apply to computer program production and specifies their implementing media such as magnetic tape, disc, drum, etc. Two-part specifications (which combine B5 and C5 specs) provide a translation of the performance requirements into programming terminology and quality assurance procedures. When two-part specifications are used, Type B5 shall form Part I and Type C5 shall form Part II.

This standard describes the format and identification of specifications. "Specifications shall contain six numbered sections ... as shown below":

1. Scope
2. Applicable Documents
3. Requirements
4. Quality Assurance Provisions
5. Preparation for Delivery
6. Notes
7. Appendix

<u>MIL-STD-1679 (NAVY)</u>: Weapon System Software Development.
1 December 1978.

MIL-STD-1679 (Navy) has the subject of weapon system software development, which is necessary because of factors that are unique or have a significantly different degree of emphasis, including: criticality of performance, changing operational requirement (sometimes technical design efficiency must be sacrificed to facilitate efficient change), and life-cycle cost considerations (requires standardization of program design, languages used, and system interfaces).

The scope of this standard is weapon system software within the Department of Defense. It applies to "weapon system software (including firmware), which is developed either alone, or as a portion of a weapon system." Adhering to this standard should "ensure that the weapon system software so developed possesses the highest degree of reliability and maintainability feasible".

This standard defines the weapon system and its software to be "any system or subsystem contributing to the combat capability of the operating forces - land, air, sea, C^2 systems, etc. Systems serving both the individual unit and those supporting a tactical commander fall within the definition.

This standard describes varieties of weapon system software (e.g., operational, test and maintenance, trainer, and support), and numerous other terms used in software design and development.

General requirements include the use of High Order Languages (HOL), the use of configuration management disciplines, and software development management. Deviations and waivers must be processed and approved by the procuring agency.

Detailed requirements for program performance and design, top-down development, code walk-throughs, documentation, system description, and flow charts are specified. The contractor is required, as a minimum, to utilize "those items available of the following to determine the program performance requirements":

a. System performance requirements.
b. System design specifications.
c. Equipment design specifications.
d. Interface design specifications.
e. Operational standards, doctrine, and tactics.
f. System design standards.

Program performance requirements are subject to the review and approval of the procuring agent.

"Total system memory, input and output channels, and processing time reserves of at least twenty percent shall exist at the time of program acceptance by the procuring agency."

The following programming design and coding standards are required: control structures (only the five basic ones); included/copied segments written in an HOL only; entry-exit structure; program traceability; no self-modification; recursive procedures only used if the target computer has a stack oriented architecture; modules not to exceed a maximum of two hundred executable HOL statements; branching statements to be approved by the procuring agency; and programs to be built in the form of relocatable object modules. It is required that numerous further programming conventions shall be observed with the intent of producing quality software.

The program shall be generated/implemented in a top down fashion. Code walk-throughs of each program component shall be conducted prior to compilation. Copious use of comment statements shall occur to provide documentation and clarification. Source code statements shall be optimized for execution efficiency. A listing of a compiled program with object machine instructions and equivalent assembler statements, if available, is a requirement for acceptance as a deliverable.

The contractor determines the scope of tests required to ensure that the program meets all specified requirements and the acceptance criteria. The contractor is responsible for accomplishing all development testing. Test planning must include development of program acceptance criteria, levels of testing to verify performance, procedures for scheduling and conducting tests at each level and reporting those procedures.

Module tests must be adequate to determine compliance with technical, operational and performance specifications. Then the modules shall be integrated individually into particular subprograms for subprogram testing, that ensures error-free linkage of the modules, ensures that the subprogram fully satisfies the detailed performance and design requirements, exercises the subprogram so that the satisfaction of detailed performance and design requirements is demonstrated, ensures the subprogram level man-machine interfaces, and ensures the capability of the subprogram to handle properly and survive erroneous inputs.

Program performance tests shall be performed to ensure the total man-machine interface, system initiation, the proper interfacing of all specified equipment, and the capability of the program to satisfy all applicable system performance requirements. A system integration test may be required at some outside facility, requiring technical support to the integration testing on the part of the contractor.

It is required of the contractor that he maintain an internal procedure for handling software trouble reports (STR), whose final disposition, after all appropriate action has been completed, is determined and recorded by the contractor. STR categories are as follows:

S: Software trouble. Software does not operate according to supporting documentation and the documentation is correct.

D: Documentation trouble. The software does not operate according to supporting documentation but the software operation is correct.

E: Design trouble. The software operates according to supporting documentation but a design deficiency exists.

L: Logic trouble. The software has a logical error with no directly observable operational symptom but with the potential of creating trouble.

Priorities of software errors by severity are as follows:

Priority 1 - An error which prevents the accomplishment of an operational or mission essential function in accordance with official requirements (causes a program stop), or interferes with an operator or jeopardizes personnel safety.

Priority 2 - An error which adversely affects the accomplishment of an operational or mission essential function in accordance with official requirements for which no alternative work-around solution exists, or which interferes with an operator so that he adversely affects the accomplishment of an operational or mission essential function so as to degrade performance and for which no alternative work-around solution exists. (Reloading or restarting the program is not an acceptable work-around solution).

Priority 3 - An error, as defined in Priority 2, with the difference that there is a reasonable alternative work-around solution.

Priority 4 - An error which is an operator inconvenience or annoyance and does not affect a required operational or mission essential function.

Priority 5 - All other errors.

The contractor determines the initial status of each STR when it is reported, and monitors and records any and all changes of the status of each STR.

The contractor is required to implement quality assurance procedures in each stage of the development to verify that the product program will meet current performance specifications approved by the procuring agency. The contractor's quality assurance organization shall conduct quality audits "throughout the development phase starting with design development and ending with test, certification, delivery and acceptance which measure system conformance with technical and management requirements and standards."

"The program shall have successfully completed the software quality test" prior to program acceptance. The software quality test "shall be conducted by a testing activity designated by the procuring agency and independent of the procuring agency and the development contractor(s)." The software quality test "is intended to exercise all of the functions of the software for a period of time in order to demonstrate that the software is reasonably free of serious or numerous errors." For systems designed to operate continuously, that time period shall be 25 hours, for other type systems, the time period shall be the length of time required to fulfill the system's mission, including any premission or postmission periods. Three distinct periods of stress testing are required, wherein "the software shall be required to operate at saturation levels which stress the software's capabilities in terms of response times and data handling capacity."

An authorized list of Data Item Descriptions (DIDs) is given for use by the procuring agent to order data generated from having invoked pertinent work tasks, which are essentially the same as previously used DIDs required for weapon system software development.

MIL-S-52779A: Software Quality Assurance Program Requirements.
1 August 1979.

MIL-S-52779A (Military Specification) delineates software quality assurance (SQA) program requirements, and applies to software (including firmware) acquisition, where the acquisition involves either software alone or software as a portion of a system or a subsystem, and to non-deliverable design, test, support and operational software. Periodic assessment of the SQA program is required as well as consistency with the configuration management and test and development plans.

At the time of the contract award, the contractor shall plan and implement a SQA Program which includes practices and procedures to assure compliance with all the software requirements of the contract. The contractor must document his QA program with an SQA Plan. The Plan must address tools, techniques, methodologies to be used, Computer Program Design (CPD), work certification, documentation, and computer program library controls, as well as reviews and audits, configuration management and subcontractor control. The SQA Program shall require periodic assessment and, where necessary, realignment of the Program to conform to changes in the acquisition program.

Testing requirements include review of software testability; "review of test criteria and requirements, for adequacy, feasibility, and traceability and satisfaction of requirements;" review of test plans, procedures and specifications; and verification of approved conducting of tests. The contractor shall ensure that support software and computer hardware to be used for any part of the system development are acceptable to the Government.

The SQA plan must document or reference procedures to assure prompt detection, documentation, and correction of software problems and deficiencies, analysis of data and examination of problem and deficiency reports to determine their extent and causes, analysis to prevent development of non-compliant products, and any other analysis or review provided for in the contract.

The ensurance of repeatability of tests, and documentation of the quality assurance of subcontractor software is required. The title, number, and date of this specification shall be specified in the procurement requirements for the program.

DI-S-30567A: Computer Program Development Plan (CPDP).
2 February 1978. Air Force DID.

The CPDP is a document wherein the contractor describes his specific detailed plan for the management and development of all of the computer programs and documentation that he needs to fulfill the contract. The plan may be used by the procuring activity both to assess an approve the contractor's approach and methods for computer program development, and to assist in monitoring and evaluating the contractor's efforts during development and test of the products defined by the contract.

This document is a Data Item Description (DID), which applies to the computer resources portion of system development and acquisition. It is to be utilized during the validation and subsequent phases of the DoD system acquisition cycle. A CPDP may be obtained precontractually in the bidders' proposals and may be a product of the validation contract or acquired during the full scale engineering development contract. It is intended to complement other contractual management plans which address such disciplines as systems engineering, configuration management, and test and evaluation.

Preparation instructions are given for the CPDP, with the following items required as a minimum: requirements assessment summary; project objectives; work definition; work schedule; activity network (e.g., PERT); organizational chart with names of key skilled managers and employees; resource allocation description; definition of engineering standards and practices; design assurance techniques definitions, definition of procedures for design, coding and checkout; presentation of the integration and test philosophy leading to preliminary and formal qualification tests; plans for system test and evaluation; methods of anomaly detection and documentation; management controls description; description of documentation tools and techniques; any special aspects of configuration management not addressed in the overall Configuration Management Plan; procedures for qualifying and documenting vendor-supplied computer resources, and means for accomodating revision of vendor-supplied computer resources; and description of support resources for the deployment phase.

DI-T-3703A: Computer Program Configuration Item (CPCI) Test Plans/Procedures. 18 May 1977. Air Force DID.

The Computer Program Configuration Item (CPCI) Test Plans/Procedures is a Data Item Description generated by the Air Force to establish detailed qualification requirements, criteria, general methods, responsibilities, and overall planning for the Development Test and Evaluation (DT&E) qualification of a computer program configuration item (CPCI) and for subelements of the CPCI.

The DT&E CPCI test plan is normally obtained in the validation phase as a complete plan applicable to the computer program. Procedures are normally obtained in the design and development phase. The test plan contains sections stating purpose, reference documents, test concepts, qualification requirements and criteria, qualification objectives/test phase summary, a DT&E CPCI qualification test implementation plan, and control and reporting procedures.

The test procedure must have a caption containing test identification, contract item to which the test applies, and the primary functions to be tested. The location and schedule for briefings, tests, debriefings, and data reduction/analysis related to the test efforts shall be shown. Procedures must be specified for initiating the computer program operation, maintaining the computer program operation, and terminating and restarting the computer program operation.

Footnotes to this DID define verification to be the iterative process of determining whether the product of each step of the CPCI development process fulfills all of the requirements levied by the previous step, and validation to be the evaluation, integration and test activities carried out at the system level to ensure that the finally developed system satisfies the mission requirements set down as performance and design criteria in the system specification.

DI-T-3717A: Computer Program Configuration Item (CPCI) Development Test and Evaluation Test Report. 18 May 1977. Air Force DID.

The Computer Program Configuration Item (CPCI) Development Test and Evaluation Test Report is an Air Force DID that is used to report the results of an individual Development T&E (DT&E) preliminary or formal qualification test for a computer program configuration item (CPCI) and to report a summary of the total DT&E process. This DID serves as the major link between the end of CPCI DT&E and the start of system DT&E testing, and is also applicable to validation and verification efforts.

The requirements for the report include the following:

a) Draft incremental reports shall be submitted in accordance with the planned test groupings but in no case at intervals less than 3 months;

b) The CPCI DT&E Final Test Report shall consist of a final summation report of the total CI/Subsystem test process. Incremental reports previously submitted and revised, in final form, shall be resubmitted.

The report must contain the number and name of the CPCI to which the test applies, the identification of the individual qualification test as shown on the test procedure, and the CPCI's primary functions or segments to which the test applies.

Test results must be stated, as well as recommendations for subsequent action, based on the test results. They may include revising the CPCI to meet specifically identified, but not fulfilled, requirements; conduction of additional tests; and qualifying those functions for which test objectives have been fulfilled.

MIL-HDBK-255 (AS): Nuclear Weapons Systems, Safety, Design and Evaluation Criteria for. 5 May 1978.

MILITARY-HANDBOOK-255(AS), prepared by the Naval Air Systems Command, provides information on nuclear safety design and describes the criteria for nuclear weapons systems, safety, design and evaluation. It applies only to weapon systems that use nuclear components. This document provides information and guidance to those individuals responsible for design, test and procurement of nuclear weapons components and systems.

Any computer software (or methods of weapon control) which can exercise automated control over any critical nuclear weapon system function must be subjected to a software nuclear safety analysis (SNSA). The purpose of the SNSA is to assure that the implemented program controls cannot contribute to accidental or fault activation of the nuclear weapon system functions. Software is categorized by the degree to which it could affect critical functions or contribute to an unauthorized launch or release.

A first level interface is defined as any software used by automata having a direct electrical connection to a nuclear weapon. This includes all resident or processor-accessible programs. Any software used by automata having a direct electrical connection to automata having a first level interface is categorized as a second level interface.

"Nuclear safety critical software analyses shall be conducted throughout the weapon system life cycle whenever new or modified software is developed. Modified software may have a modified SNSA covering the changed segments, but all interfaces to the original program must be checked. Software which is not subject to recurring changes requires only an initial review, analysis, and certification."

Special circumstances wherein maintenance or diagnostic software may require an SNSA are delineated. The organization performing the SNSA and the software developer are required to be managerially and financially independent of each other, and may not be in direct contact except under the direction and control of the Program Manager.

MIL-STD-SDS: Defense System Software Development (Working Papers).
15 April 1982.

MIL-STD-SDS, now in development, is a military standard for defense system software development which is being proposed by the Joint Logistics Commanders/Joint Policy Coordinating Group on Computer Resource Management (JLC/JPCG-CRM) - Computer Software Management Subgroup. In April 1979, this subgroup sponsored a joint Government-Industry workshop in Monterey, California. The initiatives resulting from this workshop's recommendations include the development of policy for software acquisition addressing the entire software lifecycle, development of military standards that are consistent with the policy framework, and the definition of Data Item Descriptions (DID's) that support the acquisition policy and standards.

The philosophy being followed in developing MIL-STD-SDS is that the standard should serve as a "bridge" between the general guidelines expressed as policy, and the detailed information in DIDs. Therefore, "since information is a major by-product of software development, many of the detailed requirements in MIL-STD-SDS are information generation requirements."

The standard requires a structured requirements analysis approach, the establishment of requirements for each Computer Software Configuration Item (CSCI), top down design, and the use of a program design language. It requires unit and integration testing, and software performance testing. The standard also requires a software specification review, a critical design review, and mandates that top level design exhibit modular architecture. Detailed descriptions of interface purpose and requirements must be provided. The standard describes formal and informal test planning. It describes and requires both software quality excellence and documentation thereof. It requires that the contractor implement and document configuration management, implement and document developmental baseline management, maintain configuration change control, and participate in functional and physical configuration audits. Because "sizing and timing" is usually a major problem in most defense system software developments, MIL-STD-SDS directs the contractor to pay close formal attention to processing resources throughout the development.

Other requirements that the contractor shall fulfill include unit development folders maintained by each programmer and a program support library to be delivered to the procuring agency with unlimited data rights. Specifications of data base requirements shall be divided into three categories: general environment, system parameters, and system capacities. Special requirements that affect the design of the CSCI may include protection of classified or limited access information, maximum degradation of performance allowed under various situations, features to facilitate testing (such as intermediate printouts), and human performance considerations.

A Comparison of MIL-STD-SDS and MIL-STD-1679 (Navy). 15 April 1982.

A comparison was performed by the JLC/JPCG-CRM-CSM with the purpose of tracing the evolution of MIL-STD-SDS, "Defense System Software Development", describing its intent, and comparing it with MIL-STD-1679 (Navy), "Weapon System Software Development". Separate descriptions of MIL-STDs SDS and 1679 (Navy) are given elsewhere in this document.

The Monterey workshop participants, consisting of experts in software acquisition from Government and industry, found that, in the area of software acquisition policy, no general policy exists that defines a common software acquisition framework for all the service components.

Each service has implemented DoDD 5000.29 somewhat independently. Nomenclature, emphasis, interpretations, and degree of implementation differs among the services. A general policy framework was proposed by the workshop participants to address the entire software lifecycle, be consistent with defense systems acquisition policy as in the 5000 series of directives, specify a common set of functional elements and milestones, and describe the elements of software life cycle in sufficient detail to allow common implementation procedures. This framework would also provide a foundation for formulating and revising software acquisition and development standards and software documentation. Participants in the workshop recommended that a comprehensive set of DIDs be defined and developed for joint service application.

The workshop participants found that users of numerous standards related to software were confronted with:

- Requirements which are difficult to understand;

- An acquisition process which is not fully supported by its accompanying documentation system;

- Requirements which cannot be measured;

- Incompatibilities with more modern methods of developing and acquiring software.

Rather than perform the extensive effort required to revise MIL-STD-1679, it was decided to develop a new standard, drawing upon MIL-STD-1679, that would be appropriate for joint service application.

Tables are given, comparing MIL-STD-1679 with MIL-STD-SDS, paragraph by paragraph, with explanations of omissions. For example, MIL-STD-1679, in the paragraph on Module Tests, states that the module shall "have completed a code walk-through prior to being subjected to developmental testing," adequate to determine compliance with the applicable technical, operational, and performance specifications. MIL-STD-SDS replaces this with a paragraph on Unit Testing that requires testing of "individual units to check for agreement with the detailed design, for correct execution, and for proper data handling." In MIL-STD-SDS, unit testing must verify at the minimum:

- correctness of all computations using nominal, singular, and extreme data values;

- correct operation for valid and invalid data input options;

- correct handling of all data output options and formats, including error and information messages;

- that all executable statements execute as intended.

MIL-STD-1679 requires performance of module testing to:

- "ensure error-free compile/assembly of the coded module;

- ensure that the coded module fully satisfies the detailed performance and design requirements and that all code to be delivered has been exercised;

- exercise the module in terms of input/output performance with the results satisfying the applicable detailed performance and design requirements."

MIL-STD-1679, for subprogram tests, requires individual integration of modules into particular subprograms. As a minimum, subprogram tests must be performed to ensure error-free linkage of the modules; ensure that the subprogram satisfies the performance and design requirements; exercise the subprogram in terms of input/output performance with the results satisfying the applicable detailed performance and design requirements; ensure the subprogram level man-machine interfaces; and ensure the capability of the subprogram to handle properly and survive erroneous inputs. MIL-STD-SDS replaces this with two paragraphs that require the contractor to successively integrate tested units of code and perform Preliminary Qualification Tests (PQTs) on selected aggregates of integrated software until complete Computer Software Configuration Items (CSCI) are built. The contractor is required to specify and implement an integration approach which accomplishes the following: segments the integration of the CSCI into manageable steps; integrates code in a sequence, and provides a meaningful measure of CSCI development progress by demonstrating selected CSCI capabilities early in the software integration testing activity.

Rather than telling the contractor how to do his job, MIL-STD-SDS emphasizes the goals of the procedure and allows the contractor to specify how he will perform the task. Eight sentences were omitted because: "It would be difficult if not impossible, to verify compliance with this requirement." Eight sentences were omitted because: "This requirement delves too deeply into the contractor's internal management of the development." Explanations for other omissions included "this requirement is arbitrary", "this requirement would not necessarily be applicable in most joint service cases and, therefore, would not be appropriate for inclusion in a joint standard", "this is not really a requirement", "this requirement was inadvertently omitted, and will be considered in the final version", "this requirement is ill-defined", and, "this requirement is not supported by the JLC document system".

Proposed Revisions, MIL-STD-483 (USAF). 15 April 1982.

The proposed revisions of MIL-STD-483 (USAF), "Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs", generated by the Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management (JLC/JPCG/CRM), Computer Software Management (CSM) subgroup, at the Monterey workshop, will render MIL-STD-483 consistent with MIL-STD-SDS, and make it more appropriate for use by all the services.

A paragraph by paragraph list is given of changes incorporated into MIL-STD-483 and the reasons for those changes. Table 1 consists of a Paragraph Cross Reference for Original Appendix VIII and Revised Appendix VIII. Changes incorporated into the proposed revisions to MIL-STD-483 include the following:

- Changes to reflect the distinction between software configuration management requirements from those for hardware CIs;

- Changes from "Air Force" to "procuring activity" to reflect an expanded role as a tri-service standard;
- Changes to require the preparation of a verification matrix rather than allowing it as an option;
- Addition of a paragraph (6.1) containing a list of all Software Data Item Descriptions referenced within the standard, to facilitate maintenance of the standard and provide a useful cross-reference table.

JLC policy was used as a criterion for appropriateness of terminology and consistency. Appendices III, VI, XII, and XVI from the original version have been deleted because their contents were either inconsistent with JLC policy or redundant with information contained in other standards and data item descriptions. These appendices related to, respectively, System Specification/System Segment Specification; Computer Program Configuration Item Specification; Configuration Audits; and Non-Complex Computer Program Specifications. The terms "program" and "programs" were changed to software, and "CPCI" or "CPC" to "CSCI" or "CSC", in keeping with JLC terminology.

The paragraph on baseline management was revised to reflect JLC policy which requires three baselines for software: functional, allocated, and product baselines, as well as a developmental baseline which is normally employed in software management.

Text that was perceived unclear or contradictory was changed. Sections that were derived from hardware standards were revised to address the unique characteristics of computer software. "The most significant section revisions were rewriting Appendices VIII and XVII." Appendix VIII deals with maintenance and change control procedures for computer software specifications and support documentation; the subgroup considered this appendix to be confusing, resulting primarily from the fact that it was derived from MIL-STD-483 Appendix VII which deals with hardware CI documentation, and hardware procedures are not always amenable to software documentation maintenance. Appendix XVII was rewritten to provide better guidance for selecting Computer Software Configuration Items (CSCI). New information for this appendix was derived from the report of the June 1981 JLC Software Workshop Panel on Hardware/Software/Firmware Configuration Item Selection Criteria.

This version of MIL-STD-483 requires the inclusion of a verification matrix in the specifications prepared, and gives a sample of the verification cross-reference matrix, where previously this was an option.

Proposed Revisions, MIL-STD-490: "Specification Practices". 15 April 1982.

The proposed revisions of MIL-STD-490 were generated by the Joint Logistics Commanders/Joint Policy Coordinating Management Subgroup. This standard addresses the subject of practices for the preparation, interpretation, change and revision of program peculiar specifications. Its purpose is to establish uniform specification practices for configuration identification concepts of the DoD Configuration Management Program.

The rationale for the proposed changes is the same as for all revisions of MIL-STD's recommended by this subgroup, as well as the generation of MIL-STD-SDS, i.e. to develop a general policy framework for software acquisition throughout the entire software life cycle, develop unified military standards for use by all Services which are consistent with the policy framework, and provide uniform terminology and definitions.

The proposed changes consist mainly of a new section, modified appendices, and terminology changes. A section was added to allow for the appropriate DID's to be referenced. Appendices were changed to reduce redundant material but were not deleted because other documents that reference 490 would be inconsistent.

Terminology changes were made to accomodate the new JLC policy and JLC Data Item names, for example: "Computer Program Product Specification" was changed to "Software Product Specification", "System" to "System/Segment", and "Quality Assurance Provisions" to Qualification Requirements". "Type C2b Critical Item Fabrication" under Type C Product Specifications in the paragraph addressing Classification for specifications was added for consistency.

Proposed Revisions, MIL-STD-1521A (USAF): "Technical Reviews and Audits for Systems, Equipments, and Computer Programs." 15 April 1982.

The type of technical reviews and audits that the program manager may select, according to this revision, include, but are not limited to, the following:

- Systems Requirements Review (SRR). Conducted to ascertain the adequacy of the contractor's efforts in defining system requirements.

- System Design Review (SDR). Conducted to evaluate the optimization, correlation, completeness, and risks associated with the allocated technical requirements.

- Software Specification Review (SSR). Finalizes Computer Software Configuration Item (CSCI) requirements so that the contractor can initiate preliminary software design.
- Preliminary Design Review (PDR). Conducted for each CI/CSCI to evaluate progress, technical adequacy, and risk resolution of the selected design approach.
- Critical Design Review (CDR). Conducted for each CI/CSCI when the detail design is essentially complete.
- Test Readiness Review (TRR). Determines that the software test procedures are complete and that the contractor is prepared for formal software performance testing.
- Functional Configuration Audit (FCA). Validates that the development of a CI/CSCI has been completed satisfactorily and that it has achieved the performance and functional characteristics specified in the functional or allocated identification.
- Physical Configuration Audit (PCA). Examines a designated CI/CSCI to verify that the CI/CSCI "As Built" conforms to the technical documentation wich defines the CI/CSCI.
- Formal Qualification Review (FQA). The test, inspection, or analytical process by which products at the end item or critical item level are verified to have met specific procuring activity contractual performance requirements.

The contractor's responsibilities and participation is defined. The reviews and audits must be conducted at the contractor's facility or a designated subcontractor facility if approved by the procuring activity. The procuring activity's participation in Reviews/Audits is defined to include serving as co-chairperson, providing information on each participating individual to the contractor, and providing formal acknowledgement to the contractor of the accomplishment of each Review/Audit after receipt of the minutes.

The background and rationale for the revisions are the same as those given previously. New sections have been added to MIL-STD-1521A for the Software Specification Review (SSR) and Test Readiness Review (TRR). The new section on the SSR basically requires close review of the information developed in accordance with MIL-STD-SDS. The new section on the TRR requires review of the current statements of requirements and design, test plans, descriptions, and procedures.

Modifications have been made to the sections on the SDR, PDR, and CDR to accomodate the evolution of information required by MIL-STD-SDS. Terminology changes include changing "Part I Development Specification" to just "Development Specification" or, if appropriate, "Software Requirements and Interface Requirements Specifications." "Design" was deleted from the foreword, since the scope is more encompassing than just design. The feature of the existing MIL-STD-1521A, which orders the appendices according to the relative chronological occurrence of the reviews and audits, has been preserved by inserting the new appendices on the SSR and TRR in their appropriate places.

3.3. AIR FORCE REGULATIONS

AIR FORCE REGULATION 80-14: Research and Development, Test and Evaluation. 12 September 1980.

Air Force Regulation 80-14 outlines policy for test and evaluation (T&E) activities during development, production and deployment of defense systems in the Air Force. It applies to all Air Force activities and implements Department of Defense Directive (DoDD) 5000.3. Automatic data processing resources are exempt from the T&E provisions of this regulation unless specified otherwise by HQ USAF.

A distinction is made between development test and evaluation (DT&E) and operational test and evaluation (OT&E), either of which may occur at any point in the life cycle of the system. "Their primary purposes are to identify, assess, and reduce the acquisition risks, to evaluate operational effectiveness, and operational suitability, and to identify any deficiencies in the system."

Through DT&E, the Air Force must demonstrate that the system engineering design and development is complete, that design risks have been minimized, and that the system will perform as required and specified. OT&E is conducted, in as realistic conditions as possible, to estimate a system's operational effectiveness and suitability, to identify any operational deficiencies, and to identify the need for any modifications. OT&E uses personnel with the same type of skills and qualifications as those who will operate, maintain, and support the system when deployed.

Other types of T&E may include initial OT&E (IOT&E), follow-on OT&E (FOT&E), qualification OT&E (QOT&E), and multiservice T&E (MST&E), where applicable. IOT&E is conducted before the first major production decision. It is done by the OT&E command or agency designated by HQ USAF. As a rule, it is done using a prototype, preproduction article or a pilot production item as the test vehicle.

FOT&E is that operational testing usually conducted after the first major production decision or after the first production article has been accepted. It may go on all through the remainder of the system life cycle.

QOT&E and Qualification T&E (QT&E) are performed instead of DT&E and IOT&E, respectively, on programs where there is no funding for research, development, test and evaluation (RDT&E). Multiservice T&E (MST&E) is the T&E conducted by two or more services for systems to be acquired by more than one service. Test resource management is described, as well as the test planning working group and the test and evaluation master plan which are required elements of T&E.

Responsibilities assigned to HQ USAF, the Implementing Command, the OT&E Command (which is almost always AFTEC), the Major Commands (MAJCOMs), the Operating Commands, the Air Force Logistics Command (AFLC), the Air Training Command (ATC), and the Electronic Security Command (ESC) are presented in detail. Those agencies that are involved in DT&E include HQ USAF (publish and review documents), the OT&E Command, AFTEC (overview, appoint OT&E test director, and report), the MAJCOMs (manage OT&E for operational training), and ESC, in some cases.

Some of the responsibilities of the Implementing Command are the planning, management, conducting and reporting on DT&E; collecting, processing, and evaluating reliability, availability, and maintainability data; and preparing the threat assessment to be used for T&E planning.

AIR FORCE REGULATION 800-14: Acquisition Management: Management of Computer Resourses in Systems (Vols. I & II). 12 September 1975.

This regulation has the objective of insuring that computer resources in systems are planned, developed, acquired, employed and supported to effectively, efficiently, and economically accomplish Air Force assigned missions.

Air Force policy intends that computer resources in systems are managed as elements of major importance during all phases of development and operation, management responsibility for the integration of computer equipment/programs into a system remains centralized for the life of the system, organic computer equipment maintenance and computer program development and maintenance capabilities are established where economical, computer programs are standardized to the extent practical within and across systems, Automatic Data Processing (ADP) standards and higher level programming languages are used to the maximum extent practical in the system under development, and Data Item Descriptions are identified and developed as required for program documentation support. Moreover, user involvement is an integral part of computer program development, test, operational maintenance, and major modification; "common purpose automatic test equipment is desirable"; and, there must be comprehensive testing of computer equipment and verification and validation of computer programs.

Other considerations include trade-offs of computer equipment and computer programs to minimize cost; early identification of organizational responsibilities and computer resource requirements; configuration management procedures; prime development directives (PDDs) and program management plans (PMPs); and the level of simulation to be employed. "Special emphasis is directed to these items during the testing and evaluation conducted in accordance with AFR 80-14."

The responsibilities of the program manager are to:

a. Provide management and technical emphasis to computer equipment and computer program requirements identified in the program management directive (PMD);

b. Direct the preparation, revision, and implementation of the PMP consistent with the policies of this regulation;

c. Ensure that the Program Office (PO) work with Air Force Logistics Command (AFLC) and the user to incorporate their needs into the PMP, and other system documents prepared and implemented by the PO.

AFR 800-14, Volume II, contains procedures for Acquisition Management and Support Procedures for Computer Resources in Systems. It consolidates and explains the applicability of other publications to computer resource acquisition and support, which are required by the uniqueness of computer resource management. These publications may include: AFR 80-14, MIL-STD-482, and MIL-STD-490. This regulation applies to ADP resources, as well as Embedded Computer Resources.

Computer resources will undergo a System Acquisition Life Cycle which, in general, has five major phases:

- the conceptual phase
- the validation phase
- the full-scale development phase
- the production phase
- the deployment phase

Figure 1 shows the Computer Program Life Cycle (CPLC) as a function of time. This life cycle is not bound to the system acquisition life cycle; for example a mission simulation computer program may undergo all of the phases of the CPLC during the conceptual phase, while a mission application program may undergo these phases during the validation, full-scale development, and production phases. Activities need not be sequential, there are potential loops between all the phases.

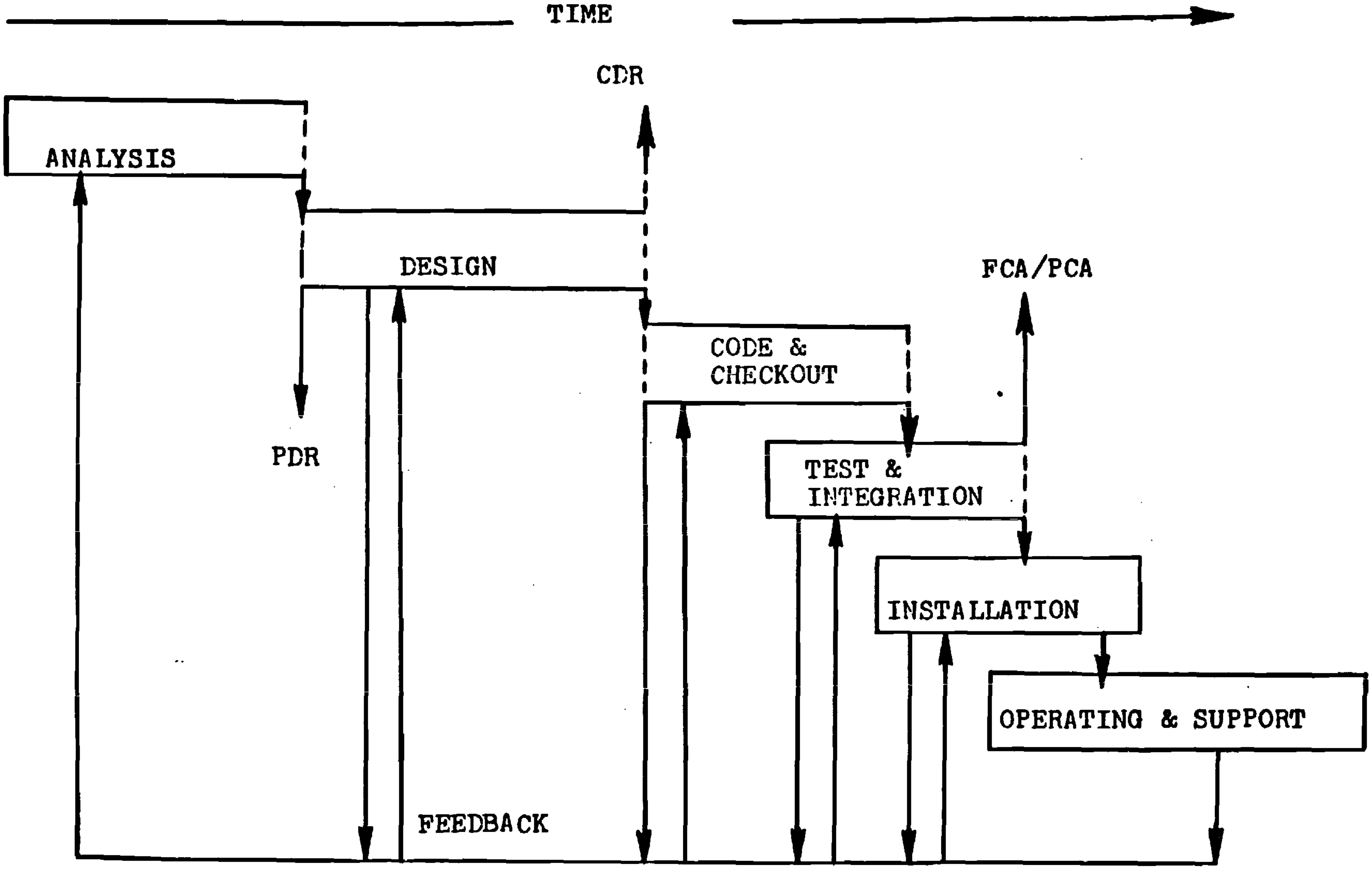


Figure 1: Computer Program Lifecycle

Guidance in planning acquisition and support of computer resources in the case where the computer resources are identified during the course of system or equipment development and in the case where the computer resources are known to be required at the outset is provided. Requirements for computer resources evolve from overall system requirements as a result of applying system engineering disciplines.

Computer resources must be considered as an integral part of the system and must be subjected to optimization and trade-off studies. "Higher level languages may simplify programming and thus reduce programming costs, but for real time processing systems more powerful and expensive computer equipment may be necessary to efficiently process computer programs written in these languages."

The Program Management Directive includes numerous guidelines for the program manager relating to computer resources, documentation, audits, testing and maintenance, and program development and support requirements. The Computer Resources Integrated Support Plan (CRISP) identifies organizational relationships and responsibilities for the management and technical support of computer resources. The Computer Program Development Plan (CPDP) identifies the actions needed to develop and deliver computer program configuration items and necessary support resources. The Computer Resource Working Group (CRWG) consists of representatives from the implementing, supporting and using commands, and is responsible for preparation and revision of the CRISP.

A fundamental concept associated with engineering management is the use of a series of configuration management baselines which aid in assuring an orderly transition from one major decision point to the next throughout the system acquisition life cycle. Baselines are established at discrete points in a program when it is necessary to define a formal departure point for control of future changes.

The principles of AFR 80-14 apply to testing of computer resources. Testing is reported in accordance with that regulation.

To perform Computer Program Validation/Verification (V&V) in the:

a. Analysis Phase, a review of all available documentation for logic and completeness should be made; a timing and sizing study should be conducted to insure that the proposed computer system is adequate;

b. Design Phase, all models should be checked for logic and completeness; a scientific simulation of the system may be produced, to develop algorithms and to check system interfaces;

c. Code and Checkout Phase, desk-checking or a correctness proof may be performed, or the following types of automated test tools (available for static code checking) may be utilized:

1) Instruction-by-instruction comparators of two versions of the same program;
2) Editors that flag coding errors and produce cross-reference listings;
3) Flowcharters;
4) Logic/equation generators used to reconstruct arithmetic text and to flowchart assembly language programs;
5) Pathfinders, traps and traces, which analyze possible paths through a given program;
6) Interpretive Computer Simulation (ICS), a simulation of the operational computer on a host computer;

d. Test and Integration Phase, several different types of simulation are used;

e. Operational and Support Phase, simulations may be useful for reproducing operational problems and for retesting the system.

The configuration management practices and procedures of AFR 65-3 shall be applied to computer resources throughout the system acquisition life cycle, giving attention to the importance of specifying and controlling interfaces, and keeping the using command involved in the approval of any changes that may be effected by a separate supporting command. MIL-STDs 480 and 483 contain appropriate procedures for configuration control and processing Engineering Change Proposal (ECPs) to CPCIs. Configuration audits are performed in accordance with MIL-STD-1521A (USAF).

Specification documentation for systems, system segments, computer equipment, programs and other system components are established in MIL-STD-483 (USAF) and MIL-STD-490.

<u>AIR FORCE REGULATION 122-9</u>: The Nuclear Safety Cross-check Analysis and Certification Program for Weapon Systems Software. 1 July 1974.

This regulation establishes the requirement for performing the Nuclear Safety Cross-Check Analysis on specifically identified weapon system software that involves nuclear safety, and establishes the responsibility for nuclear safety certification of the software, for the Air Force. Positive measures must be established to prevent accidental or unauthorized arming, launching, firing, or releasing of a nuclear weapon. Certification of cross-check identified software provides assurance that the Nuclear Safety Cross-Check Analysis (NSCCA) has been properly performed.

Terms such as "critical component", Firmware, Software, Certified Software, and NSCCA are explained. "Critical" refers to functions, circuits, "hardware and software components which apply directly to, or control, the pre-arm, arm, fuze, unlock, release, launch, or targeting functions of a nuclear weapon system." Firmware is explained to be logic circuits in read-only memory that may be altered by the software under certain circumstances. Software is information used to control or program, and which is processed or produced by automatic machines. In the context of this regulation, software includes those types of machine-stored logic devices known as firmware. Certified Software is cross-check identified software which has had a successful NSCCA and is published on the nuclear certified software list. The NSCCA is an analysis performed by an organization which is independent from the software developer to ensure that cross-check identified software contains no improper design, programming, fabrication, or application which could contribute to premature, unsafe, or unauthorized operation of a nuclear system.

Responsibilities for NSCCA and certification lie with the Inspector General, AF. The Director of Nuclear Safety supervises the program and ensures implementation and maintenance of the program, certifies software, coordinates all nuclear safety certification requirements for software for use with nuclear weapons, and designates software that requires NSCCA.

Commanders of AFSC and AFLC must designate an agency within the command which serves as the focal point for all NSCCAs and certification requirements, and must provide to the Director of Nuclear Safety recommendations for certification, denial or decertification, a copy of the NSCCA results report, and criteria recommended for publication in AFR 122-10. The commander of SAC uses cross-check identified software on weapon systems wich nuclear warheads installed only after notice of certification is received, establishes procedures for changes, and ensures that all certified software is managed in such a manner that nuclear safety is not degraded. A sample format of the NSCCA results report is included as a supplement.

AIR FORCE REGULATION 122-10: Nuclear Weapon Systems Safety Design and Evaluation Criteria. 27 November 1978.

Minimum criteria for design, development, and modification of nuclear weapon systems are specified, and criteria to be used in the evaluation of systems, equipment, and software for nuclear safety certification are outlined. A glossary of terms is provided.

"Nuclear safety requirements are set up to prevent nuclear accidents and to minimize both the number and consequence of nuclear incidents and deficiencies." Each nuclear weapon system is designed and operated to control critical functions in the sequence leading to detonation of the weapon. As a minimum, the following functions are considered to be critical:

- Authorization to use the weapon (enabling),
- Intent Command Signal and Prearming (prearming is a separate and distinct function from enabling),
- Launching or Releasing,
- Environmental Sensing and Final Arming (several environmental measurements are taken to determine if the environment is within limits defined for operational use).

Design and Evaluation criteria include minimum standards for design, minimum standards for nuclear safety certification, and minimum standards for specific items. General safety design criteria for human engineering requires that at least two independent human errors will not allow prearming, arming, launching, or releasing a nuclear weapon in an operational weapon system.

In separate chapters, design criteria for ground launched missile systems, combat delivery aircraft systems, and automata and software are presented. This review will address in detail only the chapter on automata and software. The design criteria apply to automata and software which receive, store, process, or transmit data to monitor, prearm, arm, enable, unlock, target, launch or release a nuclear weapon.

The design must include a nonvolatile core or main memory with characteristics that make sure that the contents of memory are not altered or degraded over time. "The system will be designed to prevent automatic control until all valid and correct data have been loaded and verified." Once memory has been loaded and verified, programs and data must be protected against unauthorized changes. A single hardware fault must not cause a memory change that could initiate a critical function.

A method must be provided to erase any secure codes from memory. "Any software used to process the data for, provide the status of, or which can exercise automated or automatic control over any critical nuclear weapon system function, may be subjected to an NSCCA, designated a critical component, and certified according to AFR 122-9 to insure system nuclear safety integrity."

This Page Intentionally Left Blank

3.4. ARMY REGULATIONS

AR 70-1: Army Research, Development and Acquisition.
15 February 1977.

This regulation establishes policy, responsibilities, and general procedures for conducting Army research, development, and acquisition. The objective of Army research and development is timely development of weapons and systems, at minimum cost, with adequate performance to meet approved operational requirements and capable of being effectively manned and supported in any environment under all conditions of war. With reference to "Policy and principles for conducting research, development, and acquisition", that section which is appropriate to developing computer software states that:

> "Throughout the research and development cycle, emphasis must be placed on the design, test, and production of equipment operable and maintainable by individuals possessing common skills, aptitudes, and education levels in order to reduce training cost."

Furthermore, it is specified that test and evaluation will begin as early as feasible.

Development contract clauses should be flexible and state so as to encourage the contractor to conduct trade-offs and request cost effective waivers. The project manager may waive technical requirements of MILSPEC/STDs which are not determined to be within the overall program objectives. "All approved requirements documentation is also published in complete form as separate documents (AR 71-9)."

Responsibilities of the Army Staff and major commands are listed and the role of the US Army Operational Test and Evaluation Agency (OTEA) in support of materiel acquisition and force development is identified. OTEA has the responsibility of supporting the materiel acquisition and force development processes by exercising responsibility for all operational testing and by managing force development testing and experimentation and joint user testing for the Army. In conjunction with TRADOC, OTEA verifies that known hardware deficiencies affecting combat capability have been corrected and such corrections incorporated into production hardware prior to initial issue to units in the force. The Commanding Officer, US Army Computer Systems Support and Evaluation Agency has the responsibility for technical evaluation services to developers "during design and development of electronic data processing and computing equipment", as requested, and the US Army Computer Systems Command has responsibilities, in coordination with other agencies, for developing the Army-wide program of R&D in computer software techniques.

The life cycle system management of Army systems is divided into four phases: conceptual, validation, full-scale development, and production and deployment. Management decisions during the acquisition cycle are made at milestones appropriate to the particular program. Reviews are held to provide a sound but flexible decision making process. The Materiel Acquisition Decision Process (MADP), a vital part of the materiel acquisition process, may (depending on the type of program) include the following reviews:

(1) Defense Systems Acquisition Review Council (DSARC) reviews for major system acquisitions.

(2) Army Systems Acquisition Review Council (ASARC) reviews to prepare for the DSARC reviews for major systems, or make major decisions on non-DSARC major acquisition programs.

(3) In-Process Reviews (IPR's) to make major type decisions for non-major system acquisitions.

Essential ingredients of an effective MADP review are a full interchange of information (including Development Test (DT) and Operational Test (OT) plans, reports, and evaluations, and Cost Operational Effectiveness Analyses (COEA)) and the freedom to consider and accept other courses of action.

Strategies or special documentation for major programs include:

- the Special Task Force or Special Study Group (STF or SSG), a group convened to conduct analysis, ensure inclusion of all alternatives within an analysis, monitor experimentation, etc.

- the Decision Coordinating Paper (DCP), an OSD acquisition document that brings the rationale for starting, proceeding into the next acquisition phase, reorienting, or stopping a development program at each of the critical milestones in the acquisition cycle.

- the Program Memorandum, a document similar to a DCP and initially prepared by the materiel developer.

- the Requirements Control Board, a panel of selected senior officials which reviews trade-off options concerning technical requirements (formed on an exception basis).

- the system/project management concept that uses a system/project manager who is responsible for planning, organizing, directing, and controlling all phases of the development, procurement, production, distribution and logistical support.

- the OSD management reviews that evaluate the organization and procedures for the management of selected major programs, usually in the form of a briefing by the project manager to a panel of senior OSD officials.

Strategies for non-major programs include conducting In-Process Reviews (IPR) to evaluate project/system status and recommend a course of action. Participants in these reviews include representatives of the materiel developer, combat developer, logistician, and trainer. For systems using ADP software, membership in the IPR will include a representative of the organization responsible for such effort. Test agencies will present DT and OT evaluations directly to the IPR. Formal IPRs that may be held are the:

- Validation IPR, held upon completion of the advanced development effort.

- Development acceptance IPR, where results of development and operational testing and evaluations will be discussed by the testing agencies.

- Special IPR, directed by the materiel developer or the Deputy Chief of Staff for Research, Development, and Acquisition (DCSRDA) when other formal IPR's are not appropriate.

At a minimum, normally, the validation and the development acceptance IPR will be required.

Nondevelopmental items are those items available for procurement to satisfy an approved materiel requirement with no expenditure of RDTE funds for development, modification, or improvement. "The MADP review applicable to nondevelopmental items is the special IPR." Nondevelopmental items only undergo that T&E necessary to assure "acceptability of the item for entry into the operational inventory and adequacy of logistic support concepts and resources."

Certain criteria that are applicable to all acquisition programs and must be considered at each MADP review are the criteria for entry into the Validation Phase, criteria for entry into Full-Scale Development (includes DT/OT I independent evaluations, when scheduled), criteria for low rate initial production (includes OT III, DT/OT II test results and independent evaluations), and criteria for full production and deployment (DT II and OT II or DT III and OT III test reports, as appropriate).

This regulation divides testing during the development and acquisition of materiel into DT, OT, and production and post-production testing. Development testing is designed to demonstrate that the engineering design and development process is complete, design risks have been minimized, the system will meet specifications;

and, to estimate the system's military utility when introduced. Operational testing is accomplished with typical user operators, in as realistic an operational environment as possible, to provide data to estimate the system's operational effectiveness and suitability, the system's desirability to the user considering equipment already available, the need for modification to the systems, and the adequacy of doctrine, organization, operating techniques, tactics, training for employment of the system, and maintenance support for the system.

Production and post-production testing is accomplished by or for the procuring activity on full-scale production quantities (see AR 700-78). The coordinated test program (CTP) will state what DT and OT are to be accomplished, tailor test requirements to the particular system and be as specific as possible, and will identify critical issues to be examined through testing. OTEA is the independent evaluator throughout the CTP.

Automatic data processing equipment (ADPE), including that ADPE which is integral to combat weapons systems, is within the purview of this regulation. Advanced development, engineering development, and operational system development addresses software test data packages and considerations for electronic and signal security, and requires software and documentation for tactical data (command and control) systems to be developed and tested concurrently with hardware.

The objective of system-advanced, engineering, and operational systems development is to conduct the necessary engineering, development, and T&E to ensure qualitatively superior weapons and equipment, which meet requirements, are simple to operate and maintain, and are reliable and affordable. The materiel developer is required to manage the detail design effort necessary "to provide a specific design approach, and to provide test hardware and software test data packages which will approximate production hardware as closely as practicable". This includes actively seeking and obtaining the combat developers' advice and assistance at project initiation and continuously through the development program. The developer must prepare and update a development plan. "Signal intelligence vulnerability must be considered for all systems ... involving communications, data processing, or intentionally radiated electromagnetic energy."

Sufficient funds must be programmed to provide for "the technical uncertainty inherent in the development effort, including programming for necessary design, engineering, testing, fault location and correction, Producibility, Engineering, and Planning (PEP); and an allowance for engineering changes, as well as concurrent development and testing of the maintenance test package (AR 750-1)." PEP measures include, but are not limited to, developing technical data packages, and computer modeling or simulation of the production process to better assess producibility. PEP will be conducted so that the results of DT II/OT II can be incorporated into the PEP process.

Product engineering and integrated logistic support (ILS) must be completed prior to the decision to produce/deploy the item or system, and will include reliability and maintainability, quality assurance, data acquisition and analysis for system effectiveness assessment.

AR 70-10: Research and Development Test and Evaluation During Development and Acquisition of Materiel. 29 August 1975.

This regulation implements DoD Directive 5000.3 and incorporates recommendations from the Army Materiel Acquisition Review Committee (AMARC). It applies primarily to the development testing (DT) and operational testing (OT) that are accomplished during the materiel acquisition process, providing separate and independent evaluations for consideration at decision reviews. A decision review is a program review conducted by the Defense Systems Acquisition Review Council (DSARC), Army Systems Acquisition Review Council (ASARC), or by an In-Process Review (IPR). DT and OT are defined and described. DT is that T&E conducted to demonstrate that the engineering design and development process is complete, that the design risks have been minimized, and that the system will meet specifications. OT is T&E that is conducted to estimate a prospective system's military utility, operational suitability and effectiveness. OT also provides information on personnel requirements and operating instructions and handbooks.

The role of the Operational Test and Evaluation Agency (OTEA) is described. OTEA is responsible for all OT and manages Force Development T&E (FDTE) and joint user testing for the Army. OTEA determines when, where, how, and by whom operational testing will be accomplished for all major and selected non-major systems.

Planning for testing is addressed, and testing during validation and full-scale development phases are described.

Specific responsibilities of Army Staff and major Army Commands for participating in and managing test and evaluation in the materiel acquisition process are stated. The roles of materiel developers, combat developers, the logistician, OTEA, the trainer, the operational tester, whether or not it be OTEA, and the Concepts Analysis Agency (CAA) are listed and described.

The goals for Army testing include:

1) Maximum efficient use of resources, avoiding duplication of efforts, facilities, or programs;
2) Complete testing, with a minimum of DT II, OT II, and DT III;
3) Objective testing, where normally DT and OT are conducted separately.

The testing to support decision making includes DT I, II, and III, which are, respectively:

1) Testing beginning early in the development cycle to demonstrate that technical risks have been identified and solutions are "in hand";

2) The source of the final technical data for determining the system's readiness for transition into either the low-rate initial production portion or the full production portion of the production and deployment phase;

3) The test that is conducted on production prototypes or items delivered from an initial or a pilot production run to verify their adequacy and quality when produced in quantity using quantity production processes.

Operational testing is done by an organization independent of the developing, procuring, and using commands, and usually conducted in phases keyed to an appropriate decision review in the materiel acquisition process. OT I is a test of the hardware configuration of a system or its components to provide an indication of military utility and worth to the user. OT II is the test of engineering development prototype equipment prior to the initial production decision. OT III is normally a test of initial production items and has the fundamental purpose of providing data on the item or system to estimate its operational suitability.

Nondevelopmental item and system testing is required prior to type classification. Type classification identifies the life-cycle status of a system by the assignment of a type classification designation and records the status of a system in relation to its overall life history as a guide to procurement, authorization, logistical support, assets, and readiness reporting. Redundant testing is prohibited. Force development testing and experimentation (FDTE), technical and operational feasibility testing, innovative testing and on-site user testing are described for relevant circumstances.

When computers and computer programs are part of the system to be acquired, integration of the testing of the computer programs must be given special consideration. The extent to which the computer programs form the basis for satisfying the functional and performance requirements of the overall system will impact the extent to which separate or integrated hardware testing must or can be accomplished.

Testing of special materiel, such as SAFEGUARD and Site Defense systems, aircraft testing, nuclear weapons and nuclear reactor systems testing, chemical weapons testing, and communications security (COMSEC) equipment testing is described in terms of responsible agencies and technical criteria.

The final chapters of the regulation deals with test administration and test funding, specifically the coordinated test program (CTP); the five-year test program (FYTP), a compendium of approved outline test plans requiring user troops, independent evaluation and test plans and reports; and test funding, support, and organization. An appendix provides an explanation of terms used.

AR 702-9: Product Assurance - Production Acceptance Testing and Evaluation. No Date.

This regulation is a revision that includes definitions and selection criteria for First Article testing, instructions regarding accountability for and auditability of testing, and an expansion to cover computer software and materiel evaluation. Supplementation of this regulation is permitted but not required. First Article tests are a group of tests performed on preproduction models or prototypes that have been manufactured at the intended production facility using the intended production components, processes, and personnel.

The purpose of this regulation is to assure product conformance to baseline performance, safety, reliability, availability, and maintainability (RAM), and quality requirements; to determine the suitability of those requirements for satisfying operational needs; and to provide for the reporting, evaluating, and correcting of production test incidents/defects prior to release of the materiel to the supply system. The tests and evaluations described are those required for Product or Quality Assurance purposes, after development and operational tests are conducted and the decision to enter production is made, for reconfiguration of production materiel, and when the Army elects to acquire equipment or software for which it does not own the baseline.

In the situation when limited production is authorized on an exception basis, Development Testing III (DT III) can be conducted on preproduction prototypes or production items. The evaluated test data provide information to the production and deployment decision process such as results of tests similar to First Article Preproduction Tests of the item's quality and conformance to contract requirements. "The conduct of DT III is RDTE funded, and the test items are procurement funded."

No preproduction or production test may start without government approved explicit, firm, and auditable test requirements, plans, and procedures. Production test plans and requirements may not be waived nor changed during a test.

"Required testing during production includes First Article Tests (FAT) that determine the producer's ability to produce a conforming product using the production facilities, tooling, processes, and personnel intended for the production run; Comparison Tests (CPT) and Interchangeability Tests (ICT) to insure that production items conform to contract requirements; and Production Acceptance Tests (PAT) to insure that only conforming products are accepted." The First Article Preproduction Test (FA-PPT) is conducted on one or more production prototypes prior to initiating full-scale production. It serves the purpose of confirming contract compliance, proving out the producer's detailed design for production, and confirming corrective modifications from earlier testing on engineering prototype materiel.

Comparison testing is a periodic test of random samples of full-production items that is conducted as a quality assurance measure to detect any design, manufacturing, or quality deficiencies that have developed during volume production. It is conducted or supervised by an agent independent of producer or Government on-site quality assurance personnel.

Quality conformance acceptance inspections are the examination and verification tests normally prescribed in the technical data package for performance by the contractor. They include, as necessary, measurements or comparisons with technical quality characteristics. Production quality conformance acceptance inspection requirements will be prescribed in the technical data package. Quality conformance inspections include materiel test, environmental test, reliability tests, and endurance tests. These tests must be designed to provide data which may be used as the baseline for subsequent testing.

It is specified that previous validated test data, including contractor generated test data, shall be used to avoid costly duplicative testing. A section on software testing will be added to this document.

Chapter 2 addresses the specific responsibilities of the Army agencies which are involved with quality assurance testing, operational testing, safety activities, health precautions, and research and development.

AR 1000-1: Utilization, Basic Policies for System Acquisition.
1 June 1981.

Army Regulation 1000-1 addresses Basic Policies for Systems Acquisition, adds a term, Designated Acquisition Program (DAP), for programs reviewed by ASARC, but not by DSARC, and provides guidance for integrated logistics support (ILS), reliability, availability, and maintainability (RAM), and rationalization, standardization, and interoperability (RSI). This regulation applies to all elements of the Active Army. It does not apply to ADP equipment or services, but does apply to computer resources that are integral to or in direct support of battlefield systems. These resources are dedicated and essential to the specific functional task for which the higher order system was designed.

System acquisition policy for Federal agencies shall:

(1) express needs and program objectives in mission terms and not equipment terms;

(2) encourage innovation and competition in creating, exploring, and developing alternative system design concepts (ASDC);

(3) place emphasis on the initial activities of the system acquisition process;

(4) ensure appropriate trade-offs among acquisition, operating and support costs, schedules, and performance characteristics;

(5) provide strong checks and balances by ensuring adequate system T&E, and conduct such T&E independently where practicable;

(6) accomplish system acquisition planning based on analysis of agency missions;

(7) rely on private industry in accordance with OMB policy;

(8) ensure that each system fulfills a mission need, operates effectively in its intended environment, and "demonstrates a level of performance ... that justifies the allocation of the nation's limited resources for its acquisition";

(9) depend on competition between similar or differing system design concepts throughout the entire acquisition process, when economically beneficial.

Command and control (C^2) systems require special management relating to their rapidly evolving technology, multiple interface requirements, and reliance on ADP hardware, software, and communications. It is specified that they will be developed in an evolutionary manner, with special attention to computer processing and storage space.

Four milestone decisions and four phases of activity comprise the normal DoD system acquisition process. Milestone 0 includes the approval of the Mission Element Need Statement (MENS) and the authorization to proceed into concept exploration, and exploration of ASDC. "ASCD refers to alternative types of systems which solve the need." Milestone I includes the selection of alternatives and the authorization to proceed into Phase I, Demonstration and Validation (D&V). Milestone II includes authorization to proceed into Phase II, Full-Scale Development, which includes limited production for OT&E. At this time the decision authority intends to deploy the system. Milestone III includes the authorization to proceed into Phase III, Production and Deployment.

Acquisition of ECR for operational military systems, including C^2 and automatic test equipment, will be managed within the context of the total system. Interface requirements and plans to achieve interface must be identified early in the life cycle, and special attention must be paid to the plans for the following:

(1) Software development
(2) Documentation
(3) Testing
(4) Allied interface
(5) Post-deployment software support (PDSS)
(6) Communications
(7) Automation security

Standardization policy and plans for ECR include coordination with US Army Materiel Development and Readiness Command (DARCOM); minimization of types of battlefield computers, software support requirements, and assembly language programming; use of DoD approved HOLs, use of a standard ISA after 1982; and development of a military computer family for use on new systems after 1983.

The Integrated Program Summary (IPS) describes the program management plan for the complete acquisition cycle and includes 23 topics but should not be longer than 60 pages. These topics include program history and alternatives, cost information, threat assessment, system vulnerability, overview of acquisition strategy, and overview of computer resources.

<u>DARCOM Regulation 70-16:</u> Management of Computer Resources in Battlefield Automated Systems. 16 July 1979.

This regulation implements DoD Directive 5000.29, Management of Computer Resources in Major Defense Systems. It establishes policy and assigns responsibilities for the planning phase through the testing phase, training and support of Army battlefield automated systems employing computer resources, that is, those that employ computer resources and operate or have components that operate within the boundaries of the battlefield. Computer resources in these systems must be managed as elements of major importance throughout the entire system life cycle with particular emphasis on computer software.

Risk areas and a plan for their resolution consistent with operational requirements must be identified prior to Milestone II and included in the Materiel Acquisition Decision Process documentation at the Milestone II review. Interoperability and communications support requirements for using computer resources must be identified, defined, validated, and included in appropriate planning documentation during the Demonstration and Validation phase.

A Computer Resource Management Plan (CRMP) must be prepared for each Army battlefield automated system during the Demonstration and Validation Phase of system acquisition. The CRMP identifies important computer resource acquisition and life cycle planning factors and establishes guidelines to ensure adequate consideration of these factors. The CRMP is the primary document used to establish the necessary framework and support system for software control during production and post deployment.

Software quality and support will be addressed as a major consideration during all phases of the system life cycle. Computer resources, including hardware, software, and support items, with associated documentation required for the development and support of operational systems, will be specified as deliverables in all solicitation documents with the Government acquiring rights and data as specified in the Defense Acquisition Regulations (DAR's).

DoD approved High Order Programming Languages (HOL's) will be used to develop all battlefield automated system software unless it can be demonstrated that none of the approved HOL's are cost-effective or technically practicable over the system life cycle.

The Associate Director of Battlefield Automation Management within the Directorate for Development and Engineering, DARCOM (Development and Readiness Command) has responsibility for the overall DARCOM computer resource management policy for Army battlefield automated systems. The Commander of the U.S. Army Test and Evaluation Command (TECOM), Aberdeen Proving Ground, MD, has the responsibility for developing the capability and methods necessary to:

- Support the T&E of Army battlefield automated systems during development and production;
- Determine system conformance with established requirements, including reliability, maintainability, and performance;
- Conduct testing in a realistic environment, or a controlled and reproducible test environment that stresses the system design limits (worst-case testing).

Chapter 2 addresses the subject of life cycle computer resource management. In all applicable systems, the hardware and accompanying software shall proceed through the system life cycle concurrently. The system life cycle, including Milestones 0, I, II, and III, is defined and described. Technical milestones and attainment criteria that are identified include:

- The system specification, which establishes the system baseline functional requirements, and is prepared, reviewed, and evaluated prior to Milestone I;

- The development specification, which establishes the design necessary to implement, test, and maintain the functional requirements established in the system specification, and is placed under configuration management upon entry into the Full-Scale Engineering Development phase;

- The product specification, which documents the details of system implementation for production and maintenance, and is prepared prior to Milestone III;

- The system requirements review (SRR), whose objective is to ascertain the adequacy of the contractor's efforts in defining system requirements, and is conducted when a significant portion of the system functional requirements has been established.

The system design review (SDR) is conducted at an intermediate point in the definition effort, and has the goal of ensuring a technical understanding between the contractor and the procuring activity on system segments identified in the system specification and the configuration items identified in the system specification and the configuration items identified in the Computer Program Configuration Item (CPCI) development specifications. The preliminary design review (PDR) is conducted for each CPCI and is a formal technical review of the basic design approach. It is conducted after approval of the development specification and prior to the start of detailed design. The critical design review (CDR) is a formal technical review of a single or functionally related Computer Program Components (CPC), when detailed design is essentially complete, and when the draft computer program product specification and test procedures have been prepared.

A formal qualification review (FQR) is held for each CPCI, and is conducted with the functional configuration audit (FCA), after the formal qualification test has been completed. At the FCA, test plans, procedures, and test results will be reviewed for compliance with specification requirements, and it will be determined whether any tests should be repeated. The physical configuration audit (PCA) is conducted for each CPCI to establish that the CPCI technical data package is complete, and that all physical items called for by the contract have been produced in the specified configuration.

The computer resource management plan (CRMP) is developed by a group including operational testers and evaluators, materiel developer, combat developer, and designated post deployment support activities prior to Milestone II, and is maintained throughout the system life cycle. The CRMP must address the responsibilities for integration of computer resources into the total system and the test and evaluation of that system to determine entire system quality and integrity, and complete management planning for the acquisition, test, evaluation and post deployment support for all functions related to the computer resources in the Army battlefield automated system.

Test management, quality assurance, data management, integrated logistics support (ILS), training, personnel, deployment planning, compatability and interoperability, and validation and verification are defined and described. Standards for measuring software performance must be adopted on a project by project basis by the materiel developer, and include computer resource planning, specifications, documentation, programming, quality control, testing, and configuration management standards.

An appendix explains terms used in this regulation; a second appendix specifies the sections and details of a CRMP; and a third appendix addresses the issue of system acquisition reviews and provides supplementary questions for milestone review checklists.

DARCOM Regulation 702-6: Quality Assurance and Product Quality Management. 13 March 1979.

This regulation addresses Quality Assurance and Product Quality Management. It is designed to prescribe the functions that DARCOM considers necessary to verify the fact that product quality is being achieved in accordance with technical requirements and quality assurance provisions throughout the life cycle of individual items. These functions may be assigned to an individual, an organization, or a group of individuals or organizations.

Product Quality Managers must be knowledgeable in the areas of design and contract requirements; production; inspection and test methods; quality control procedures; quality assurance provisions; maintenance; serviceability; and shelf life standards, data analysis, ILS requirements, and other QA related specifications and regulations.

The objectives of product quality management are to:

- Plan appropriate steps throughout the acquisition cycle to assure that quality of materiel performance is achieved;

- Provide a focal point for QA between Government activities involved with development, production, maintenance, storage and use of DARCOM materiel;

- Provide a system wherein past experience on quality problems can be effectively used in preventing a recurrence or by which existing quality problems can be easily and quickly acted upon;

- Identify significant quality hardware problems that require intensive management;

- Plan track, and report materiel quality.

Functions of product quality management programs are listed, and include reviewing and appraising activities, coordinating activities, providing quality assurance technical assistance, and assisting QA organizations. General product quality management advance planning is described. "To the maximum extent, the 'lessons learned' approach should be practiced in all aspects of advance planning to capitalize on prior mistakes and to achieve improvement in all areas, whether technical or administrative." Quality Assurance Letters of Instruction (QALI's) are described and explained in Appendix B. In Appendix C, the Quarterly Product Quality Management Summary Report is explained.

DARCOM Regulation 702-10: Quality Assurance Provisions for Army Materiel. 22 May 1979.

This regulation deals with Quality Assurance Provisions (QAP's) for Army Materiel. Product assurance is concerned with all aspects of fitness of products for use, including performance, safety, reliability, availability, maintainability, interchangeability, and quality. This regulation defines the format, content and requirements for QAP's, establishes organizational responsibilities for preparing QAP's, establishes the relationship of QAP's to acquisition activities in each phase of the life cycle, and provides for configuration management and technical audit of QAP's.

Requirements for QAP's include:

- Identifying all features and characteristics important to assure that the design conforms to user requirements and that materiel produced conforms to design;

- Specifying acceptance standards for each characteristic defined;

- Establishing test, examination and audit methods to assure the design or product conforms to the established acceptance standards through all stages of design, development, and use;

- Specifying the test and measuring equipment needed to carry out the required procedures;

- Providing test and examination instructions;

- Providing for an Initial Production Test (IPT);

- Specifying critical processes that are essential to achieve the production design and performance requirements.

Quality characteristics of a product include reliability and maintainability requirements, material process requirements, electrical and electronic requirements, and safety features.

Appendices to this document address preparation of a QAP form, preparation of the Quality Engineering Planning List (QEPL), Sampling Plan Considerations, and Related Publications.

3.5. NAVY REGULATIONS AND STANDARDS

TADSTAND 9: Software Quality Testing Criteria Standard for Tactical Digital Systems. 18 August 1978.

The Naval Software Quality Testing Criteria Standard for Tactical Digital Systems (TADSTAND 9) was promulgated out of a growing concern over the problems of software cost and reliability within the DoD. It has been difficult for the DoD to establish meaningful test requirements supported by objective test criteria to determine software quality. "Within the practical limitation of testing protocol, resources, and time available, a test vehicle and associated testing criteria are necessary to provide reasonable assurance that a computer program will operate reliably if placed into service use." This TADSTAND has been superseded by TADSTAND E and cancelled as of 25 May 1982.

Extensive testing is not seen to be a panacea that will ensure a satisfactory system throughout its life cycle. "For new programs, the software acquisition manager must initiate a rigorous quality assurance effort at the beginning of program development." System specifications and design documentation must be reviewed to ensure that test specifications and procedures based on them are valid. "The criteria for judging a program's performance and quality should be contractually binding, derived from requirements specifications, and kept in mind throughout the development cycle." Then, if the software satisfies quality testing criteria during its development and acceptance, there is some assurance that the program will operate satisfactorily.

The life cycle support activity must sustain the quality assurance (QA) effort when programs are placed into operational use. Corrections and enhancements must receive as much attention to detail as the original development.

All tactical digital system software programs are required to meet the Software Quality Testing Criteria delineated by this standard. These criteria apply to "all operational programs, including the operating systems and on-line test programs, that are used in computers, including microprocessors, embedded in a tactical digital system." The Software Quality Testing Criteria (SQTC) also applies to the entire integrated program, including a system's interfaces with other systems and devices. However, the SQTC is not intended to be a comprehensive testing criteria of itself; "specific performance requirements and acceptance criteria are to be defined in test plans, test specifications, and test procedures." Requests for deviations from the standard must be extensively documented, including a statement of criteria that will be used to ensure software correctness.

The intent of the Software Quality Test is to exercise all of the functions of the software for a period of time in order to demonstrate that the software is "reasonably free of serious or numerous errors". Stress testing is required, in the ultimate user environment for which the system program was designed, if possible. Otherwise, an alternate test site should be a fully integrated facility equipped with the same hardware found in the ultimate user environment. A testing activity independent of the system acquisition manager and the developer shall conduct the test. The length of time that the test shall be run is a function of the complexity and mission of the system under test, and shall be 25 continuous hours for those systems that are designed to operate for more than one day at a time, or a length of time equal to the mission plus premission and postmission periods. The program test run shall not be stopped until scheduled test completion, and any stop prior to the scheduled stop is considered to be a failure to meet the requirements.

Saturation level operation is required, and at least three distinct stress periods representing at least 1/3 of the total length of the time of the test are required, where stressing includes, but is not limited to:

- Providing more information to be processed than the processor is designed to accomodate;

- Saturating the data transfer capabilities of the system;

- Exceeding assigned storage area capacities;

- Physically degrading hardware to create each possible reduced capability mode for those systems designed to operate in such a mode.

Software error limits and patch limits are specified, and it is required that they be documented.

TADSTAND A: Standard Definitions for Embedded Computer Resources in Tactical Digital Systems. 2 July 1980.

Tactical Digital Standard (TADSTAND) A has the purpose of establishing standard definitions for terms applying to embedded computer resources (ECR) in tactical digital systems, so that each TADSTAND is consistent in its policy specifications. Deviations from this TADSTAND are inappropriate.

TADSTAND B: Standard Embedded Computers, Computer Peripherals, and Input/Output Interfaces. 2 July 1980.

TADSTAND B is promulgated to establish Standard Embedded Computers, Computer Peripherals, and Input/Output Interfaces for use within the Naval Material Command. It cites DoD Instruction 5000.2 and DoD Directive 5000.29 as references. Except for major systems acquisitions that are based on DoDI 5000.2, this TADSTAND applies to all phases of tactical digital system acquisition, including initial concept formulation and requirements definition, design, development, installation, production, and post-development support throughout the system life-cycle. For the excepted systems, this TADSTAND will be applied to all phases of acquisition commencing with the Demonstration and Validation phase.

The requirement to use standard ECR in tactical digital systems comes from the perceived need to stem ECR proliferation, achieve an acceptable level of supportability, and reduce costs over system life cycles. Standardization is expected to improve reliability and maintainability and reduce cost and schedule risks in development and acquisition of new tactical digital systems. The following are designated as standard or planned standard embedded computers:

AN/UYK-7
AN/UYK-20
AN/AYK-14
AN/UYK-44 (Militarized Reconfigurable Processor-microprocessor, Software compatible with UYK-20, AYK-14)
AN/UYK-43 (Software compatible with the UYK-7)

Only these computers may be used unless a waiver is obtained. For each major system upgrade, a new waiver request must be submitted.

Configuration management will be exercised by the development, acquisition, maintenance, or support offices under the guidance of established Configuration Control Boards (CCBs). Each problem and corrective action will be dealt with by use of the Engineering Change Proposal (ECP).

If adherence to this TADSTAND is technically infeasible, economically prohibitive, or operationally impracticable, a waiver request must be submitted. Requests for waiver must include detailed descriptions of the system; software and hardware requirements; data description; reliability and maintainability information; acquisition, testing, and integrated logistic support information; and reasons, with rationale and documentation, why standards cannot be used.

TADSTAND C: Computer Programming Language Standardization Policy for Tactical Digital Systems. 2 July 1980.

TADSTAND C has the subject of Computer Programming Language Standardization Policy for Tactical Digital Systems. Excluded from the provisions of this TADSTAND are:

(1) Hardware-intensive applications authorized to use non-standard microprocessors;

(2) Automatic Data Processing (ADP) software;

(3) Special purpose languages such as requirements definition languages, design specification languages, automatic test languages, job control languages, and simulation languages.

The goal of standardizing languages is to improve total life cycle maintainability and cost effectiveness, and to provide "a significant reduction to the spiraling costs for developing, testing, and maintaining tactical digital systems."

Low level code may be used for required machine oriented functions such as input/output where the programming language does not provide high level support, or for software functions which require special optimizing or fine-tuning such as executives, interrupt handlers, math routines, etc. The programming languages CMS-2Y, CMS-2M, and SPL/I are identified as Navy standards. Ada is designated as a planned Navy standard language and is not authorized currently unless an explicit waiver is obtained. The SPL/I/CMS-2 preprocessor is designated as standard and Fortran is only approved for use in development of applications software when a waiver has been granted for the use of a non-standard embedded computer.

Waivers to this TADSTAND require documentation similar to that of TADSTAND B.

TADSTAND D: Reserve Capacity Requirements for Tactical Digital Systems. 2 July 1980.

The subject of this standard is the Reserve Capacity Requirements for Tactical Digital Systems, and applies to those systems under the purview of the Naval Material Command. This encompasses all phases of the life cycle of tactical digital systems, except for major systems acquisitions based on DoDI 5000.2, in which case the TADSTAND will be applied to all acquisition phases commencing with the Demonstration and Validation phase. The reserve capacity requirements shall apply to the first production delivery of a new system or a modified system that incorporates new Embedded Computers or hardware modifications to Embedded Computers (EC) already in the system, and shall not include capacities reserved for future growth, when the growth requirements are known prior to acquisition commitment to the configuration of the system.

As a minimum, main memory shall have a 20% reserve capacity, measured at peak main memory loading of the EC during its operational mission, with all programs and data required for successful operational mission execution. Secondary storage shall have a 20% reserve capacity, as a minimum, measured at peak secondary storage loading of the EC, with all secondary storage information included. It is required that secondary storage and central processor throughput each must have a 20% reserve capacity, as a minimum. This reserve capacity is expressed as a percentage of available capacity at full operational loading over a specific period of time (determined by operational mission characteristic). The number of reserve input/output channels required is a minimum of 18.75% (3/16) of those available.

Requests for deviations from this standard will follow the procedures similar to those specified for waivers in TADSTANDS B and C.

TADSTAND E: Software Development, Documentation, and Testing Policy for Navy Mission Critical Systems. 25 May 1982.

TADSTAND E addresses the subject of Software Development, Documentation, and Testing Policy for Navy Mission Critical Systems. It cancels TADSTANDs 2,3, and 9. It defines mission critical systems to be those systems which are required for the conduct of the military mission of the DoD. This definition includes systems related to intelligence activities; cryptology activities related to national security, command and control of military forces, a weapon or weapon systems; and, the direct fulfillment of military or intelligence missions that are not routine administrative or business applications.

TADSTAND E "applies to all mission critical systems under the cognizance of the Chief of Naval Material that use embedded computer resources," as well as other programs that are intended to be employed as mission critical systems, throughout all phases of system acquisition.

A concern exists over the high cost of software development, life cycle support, reliability and supportability within the DoD. Therefore, a need is perceived for establishing meaningful system operational and performance software requirements and objective test criteria for determining software correctness. Management controls are lacking to govern the software development, acquisition, and life cycle support process. Clear and detailed guidance to specify software specifications and requirements throughout the life cycle must be supplied to the program and acquisition managers.

Previous standards and instructions were issued as an initial step toward correcting the problem areas, but some problems arose in uniform implementation because of a lack of direction on specific Data Item Descriptions (DIDs), and a resulting proliferation of Unique Data Item Descriptions (UDIDs).

TADSTAND 9 invoked standard software quality testing criteria for tactical digital systems, was primarily concerned with acceptance testing rather than the full range of requirements covering the complete software development process, and therefore applied more to Navy software acquisition managers than to software contractors.

MIL-STD-1679 establishes minimum uniform requirements covering the complete development process of weapon system software, including program test, quality assurance, and program acceptance criteria, and must be invoked in all new contracts for development, documentation, or testing of software. When MIL-STD-1679 is invoked on any organization, the Navy development activity or acquisition manager is not relieved of still further responsibility regarding the requirements of this TADSTAND.

Limited tailoring of MIL-STD-1679 is permitted under certain circumstances, such as system size or complexity that does not require the complete minimum set of software documentation, but "in no case will such tailoring result in the development of a new DID, i.e., a UDID." Guidance for configuration management of software and related documentation is specified. Waivers of TADSTAND E may be granted, but the waiver must be granted before proceeding with the software development. Each waiver must be considered on a case-to-case basis.

<u>COMOPTEVFOR NOTICE 3960</u>: Operational Test and Evaluation of Software Intensive Systems and Computer Software Subsystems. 6 July 1979.

COMOPTEVFOR (Commander Operational Test and Evaluation Force) Notice 3960 provides general guidelines for conducting OT&E of Software Intensive Systems and Computer Software Subsystems in accordance with the software initiatives contained in DoDD 5000.3 and DoDD 5000.29. Because the cost for weapons system software was over $3 billion in 1979, and the cost, especially for maintaining operational software, is steadily rising, a directive is needed to promote higher visibility and a more disciplined approach to management of software design, engineering and programming to ensure production of effective software at minimum life cycle cost. "Due to lower visibility in the acquisition process, development and testing of software is not given the same emphasis as hardware even though it is just as critical to the operational performance of a new or existing weapons system."

The provisions of this directive are intended to provide standardized guidance for Operational Test Directors, e.g.,

(1) At an early point in system development planning and definition, COMOPTEVFOR will participate by involving typical operator personnel in the software functional design and developmental testing, identifying the extent to which DT&E accomplishes OT&E objectives and which tests might be combined, and defining what system operational issues should be monitored closely throughout the software development;

(2) During initial planning of software development, configuration management procedures should be reviewed for the purpose of ensuring that management plans and specifications required by DoDD 5000.29 (Management of ECR) are complete and promulgated. Development plans should provide sufficient configuration baselines to ensure stable software and documentation prior to final IOT&E test phases;

(3) Requirements Analysis. Early development planning can be significantly augmented by:

(a) Conducting a system and subsystem operational requirements analysis
(b) Interpreting requirements from an operational viewpoint
(c) Relating system and software subsystem operational requirements to mission needs
(d) Commenting on the adequacy of program performance specifications to capture the functional part of these requirements
(e) Identifying critical system operational issues or limitations related to software for OT-III (OPEVAL);

(4) Test planning will consist of reviewing development test plans and procedures for performance and quality oriented testing, to be sure that this testing is clearly defined in the TEMP, and that performance testing is planned at the completion of significant phases, especially the integration phase.

Stages (1)-(4) are also known as OT-I & OT-II. In the stage of OT-III Integration Testing, combined DT/OT may occur, software standards are reviewed for conformance, scenario driven tests are exercised, and an early estimate of system operational effectiveness and operational suitability may be prepared.

At the stage of OT-III addressing OPEVAL, a total system test with fully integrated software and hardware in the ultimate user environment will determine system operational effectiveness and suitability. OT-IV is designed to complete unfinished IOT&E, test fixes, and refine tactics for hardware and software, and may be continued or reopened until the TEMP objectives for that phase have been attained. OT-V is designed to ensure demonstration of the achievement of program objectives for system operational effectiveness and suitability.

In summary, the functions of COMOPTEVFOR include:

- The analysis and relating of system and software subsystem requirements to mission needs;
- Monitoring of software development by identifying and tracking operational issues and requirements;
- Preparing OT&E objectives and required operational characteristics;
- Reviewing development test plans and procedures to ensure that operational objectives have been considered;
- Providing user oriented services to developmental testing;
- Evaluating the operational impact of major changes;
- Planning and conducting OPEVAL on new or modified systems.

OPNAV INSTRUCTION 3960.10, TEST AND EVALUATION, 22 October 1975.

This instruction, issued by the office of the Chief of Naval Operations, establishes policy for test and evaluation in Navy acquisition programs, defines responsibilities, establishes procedures for planning, conducting, and reporting T&E, and establishes procedures and a format for test and evaluation master plans (TEMPs). The key element of DoD acquisition policy affecting T&E is that "Programs shall be structured and resources allocated to ensure that the actual achievement of program objectives is the pacing function." It references DoD Directive 5000.1, "Major Systems Acquisition".

Three types of T&E are defined:

- Developmental Test and Evaluation (DT&E), that test and evaluation conducted to demonstrate that the engineering design and development process is complete, that the design risks have been minimized, that the system will meet specifications, and to estimate the system's military utility when introduced;

- Operational Test and Evaluation (OT&E), that test and evaluation conducted to estimate the prospective system's military utility, operational effectiveness and suitability, and need for any modifications;

- Production Acceptance Test and Evaluation (PAT&E), that testing conducted on production items to demonstrate that systems meet contract specifications and requirements.

Authority for these three types of T&E is described, and the phases of the program development lifecycle where each is utilized are delineated. Acquisition categories are established according to dollar values allocated to funds for RDT&E and funds for production. The amount of T&E required for a program is then specified as a function of the acquisition categories (ACAT), and is more intensive for more expensive programs.

"The TEMP is the controlling management document which defines the test and evaluation for each acquisition program." This document is described and explained. It contains the integrated requirements of the developing agency (for DT&E) and COMOPTEVFOR (for OT&E). The TEMP is prepared early in each new acquisition program and approved prior to Milestone I. The initial version of a TEMP will lack many specifics that will be added in later revisions as developed. The TEMP shall be retired upon completion of the last phase of OT&E.

Special T&E situations such as T&E of ships, combined DT and OT, waivers, and special complex programs are described. T&E reports must be correlated to program key decision points and secondary milestones, and requirements for reports will be specified in the TEMP.

3.6. MISCELLANEOUS DOCUMENTS

Department of Defense Acquisition Improvement Program: Carlucci's Initiatives. 1 January 1982.

The Deputy Secretary of Defense chartered five working groups from the military Services to develop recommendations for improving the DoD acquisition process, in light of the fact that the DoD had a significant budget increase at the same time that almost all other Federal agencies were experiencing substantial cuts in funds. Recommendations resulting from the efforts of the working groups reflect "major changes both in the acquisition philosophy and the acquisition process itself". DoD is demonstrating its commitment to act by a series of "far-reaching measures" whose ultimate goal is implementing the entire group of 32 actions.

A series of initiatives were approved which reflect a shift in management style to a "controlled decentralization" style, whose objective is to return significant amounts of responsibility and authority to lower levels of management while continuing to hold program managers accountable for their management decisions. Other actions were centered around the goals of:

1) improving the planning and execution of weapon system programs,

2) strengthening the industrial base which supports the DoD,

3) reducing the burdensome administrative requirements that make the acquisition process more costly and time-consuming than necessary,

4) increasing the readiness of weapon systems, particularly in the early stages of their lives in the field.

Concerns with the acquisition process include the failure to stick to long-range plans, the burden of reporting and checking, the rising costs of acquisition, unrealistic cost estimates, the weakness of the industrial base, the length of the acquisition cycle, the high cost of ownership, and the low readiness of fielded systems. These concerns were voiced in all sectors, from Congress to program managers.

The thirty-two Carlucci initiatives are as follows:

1. Management Principles should include improved long-range planning; greater delegation of responsibility, authority and accountability; emphasis on low-risk evolutionary alternatives; more economic production rates; realistic budgeting and full funding; improved readiness and sustainability; and strengthening the industrial base.

2. Preplanned Product Improvement should be used as a means of achieving performance growth.

3. Multiyear Procurement should be used, on a case-by-case basis, to reduce unit production costs.

4. Increased Program Stability in the Acquisition Process should be achieved by fully funding R&D and procurement in order to maintain the established baseline schedule.

5. Encourage Capital Investment to Enhance Productivity through legislative, contractual and other economic incentives.

6. Budget to Most Likely Costs to achieve more realistic long-term defense acquisition budgets, reduce apparent cost growth and achieve increased program stability.

7. Economic Production Rates should be used whenever possible and advantageous.

8. Assure Appropriate Contract Type in order to balance program needs and cost savings with realistic assessment of contractor and Government risk.

9. Improve System Support and Readiness by establishing objectives for each development program and "designing-in" realiability and readiness capabilities.

10. Reduce the Administrative Cost and Time to Procure Items by raising the limit on purchase order contracts and reducing unnecessary paperwork and review.

11. Incorporate the Use of Budgeted Funds for Technological Risk by quantifying risk and incorporating budgeting techniques to deal with uncertainty.

12. Provide Adequate Front-End Funding for Test Hardware in order to emphasize early reliability testing and to permit concurrent development and operational testing when appropriate.

13. Governmental Legislation Related to Acquisition which unnecessarily burdens the acquisition or contracting process should be eliminated.

14. Reduce the Number of DoD Directives by performing a cost-benefit check and requiring that the DAE be the sole issuer of acquisition-related directives.

15. Funding Flexibility should be enhanced by obtaining legislative authority to transfer individual weapon system procurement funds to RDT&E when appropriate.

16. Contractor Incentives to Improve Reliability and Support should be developed and introduced in RFPs, specifications and contracts.

17. Decrease DSARC Briefing and Data Requirements in order to increase the efficiency of DSARC and other program reviews.

18. Budgeting Weapons Systems for Inflation should be adopted in order to more realistically portray program cost.

19. Forecasting of Business Base Condition at Major Defense Plants by coordinating interservice overhead data and providing program projections to plant representatives.

20. Improve the Source Selection Process by placing added emphasis on past performance schedule realism, facilitization plans and cost credibility.

21. Develop and Use Standard Operational and Support Systems to achieve earlier deployment and enhanced supportability with lower risk and cost.

22. Provide More Appropriate Design to Cost Goals to provide effective incentives during early production runs.

23. Assure Implementation of Acquisition Process Decisions by initiating an intensive implementation phase.

24. (ISSUE A) DSARC Decision Milestones should be reduced to "Requirements Validation" and "Program Go-Ahead".

25. (ISSUE B) MENS should be submitted with Service POM thus linking the acquisition and PPBS process.

26. (ISSUE C) DSARC Membership should be revised to include the appropriate Service Secretary or Service Chief.

27. (ISSUE D) The Defense Acquisition Executive (DAE) should continue to be the USDRE.

28. (ISSUE E) The Criterion for DSARC Review should be increased to $200M RDT&E and $1B procurement in FY 80 dollars.

29. (ISSUE F) Integration of the DSARC and PPBS Process will be achieved by requiring that fiscally executable programs be presented for DSARC review.

30. (ISSUE G) Logistics and Support Resources will be included in the Service POM by weapon system, and Program Managers will be given more control of support resources, funding and execution.

31. (ISSUE H) Improved Reliability and Support for expedited ("Fast Track") programs will be achieved by requiring an early decision on the additional resources and incentives needed to balance the risks.

32. Increase Competition in acquisition by establishing management programs and setting objectives (July 27, 1982).

Numerous major studies of the acquisition process have been conducted in the decade of 1970-1980. Some of the progress resulting in that time period can be perceived from the promulgation of DoD Directive 5000.1 and OMB Circular A-109. Both of these documents are notable for a strong emphasis on tailoring of the acquisition process to yield the optimum acquisition strategy.

In parallel with DoD actions to improve the acquisition process, OSD initiated a major activity to simplify and improve the Planning, Programming and Budgeting System (PPBS). The DoD Management Philosophy is to:

- Define the national military strategy;
- Achieve integrated and balanced forces;
- Assure that we are ready;
- Manage defense resources effectively within national resource limitations;
- Keep the Secretary of Defense informed.

Documentation of the FY 83 Program Objective Memorandum (POM) is required to be cut by 50%, and the comptroller is required to slash the high amount of paperwork required by the Zero-Base Budgeting (ZBB) process.

The objectives of controlled decentralization and participative management arise out of a concern with two decades of increasing centralization, increased reporting requirements, increased number of policies and procedures, and delays occasioned by the decision-making process.

To illustrate the complicated, time-consuming path to a DSARC review, data is presented on the number of DSARC prebriefings for sample programs, such as the F-16 Aircraft: 56; Joint Tactical Information Distribution System (JTIDS): 42; and, the F-18 Aircraft: 72.

The thrust to improve industrial productivity has the following stated objectives:

- Enable American industry to undertake a program of capital investment;
- Improve American self-sufficiency in the area of critical raw materials;
- Insure sufficient skilled manpower exists to meet the demand of American industry;
- Improve the quality of American workmanship and products;
- Impose stability on military procurement programs and resource demands;
- Make the defense market an attractive place for American industry to do business;
- Make military equipment designs compatible with commercial industrial production capabilities;
- Create an industrial base which is responsive to mobilization needs.

A table at the end of this document lists relevant statutes, their purpose, the issue that they address, and the status of the statute. The implications of the new program management environment created by these initiatives are delineated: greater authority, responsibility, and flexibility for the program manager, as well as more accountability; DoD must maintain its credibility, commitment and discipline; opportunities for innovation will proliferate.

Strategy for a DOD Software Initiative: Draft. August 1982.

The Software Technology Coordinating Committee was formed by the Office of the Secretary of Defense (OSD) to provide direction and oversight to the Software Technology Initiative. A questionnaire was distributed to all segments of the DoD/academic/industry community to evaluate the effectiveness and desirability of the candidate thrusts of the initiative and suggest other possibilities. These results were published in Summary of Responses to the Software Technology Initiative Questionnaire.

Dr. Edith Martin, the Deputy Under Secretary of Defense for Research and Engineering/Research and Advanced Technology (DUSD/R&AT) was concerned that the plans evolving from this effort might not address all critical DoD problems. Therefore, she chartered a Joint Service Task Force to report on DoD problems in using computer technology. Not all the opportunities and needs revealed by initial studies are technological; therefore, the scope of the initiative was increased to include non-technical concerns.

In order for the U.S. to maintain its military supremacy, aggressive action is needed to surmount the problems preventing us from fully exploiting computer technology. The DoD-wide initiative provided in this document will improve the state of practice in the acquisition, management, development, and support of computer software for military systems. Objectives are established and top-level plans for achieving the objectives are presented. Steps are identified to develop the next level plans for implementation.

"The military power of the United States is inextricably tied to the digital computer." Computers are integral to our strategic and tactical capabilities, and software is the essential element of the system. Software provides the flexibility to respond to changing threats, needs, and requirements. The problem is that software is a complex human endeavor that may require hundreds of people, for five or more years, and costs exceeding $100 million. The body of accepted practice and supporting theory is inadequate for development of complicated systems. Representing the immaturity of the underlying technology base, the state of practice in DoD ranges from a reasonably effective disciplined approach in a few systems "to near chaos in others".

DoD has not been ignoring the problems, but a high-level effort involving attention and coordination is required. For too long, software-related activities have lost out in the competition for resources because managers have not understood how software helps to build better systems. A cooperative effort among all DoD research activities must be coordinated to develop improved technology that will be used. Productivity must be increased, as well as the quality of software. Improved technology must be developed and used.

The goal of this initiative is to improve the state of practice to achieve faster, less expensive, and more predictable development and support to get more powerful, reliable, and adaptable software. The challenge is to advance the technology base and adopt practices facilitating the widespread use of the technology. One conservative estimate suggests that DoD can improve productivity in the current environment by a factor of four by 1990 using existing techniques.

The objectives of the initiative include:

- Improve the personnel resource by
  .. Increasing the level of expertise,
  .. Expanding the base of expertise available to DoD;

- Improve the power of tools by
  .. Improving project management tools,
  .. Improving applications-independent technical tools,
  .. Improving application-specific tools;

- Increase the use of tools by
  .. Developing the incentives to use the tools,
  .. Improving tool usability,
  .. Increasing the level of tool integration,
  .. Increasing the level of tool automation.

The strategy of the initiative is to establish the funding impetus and organizational incentives to coordinate improvement in the state of software practice in the DoD community through the planned evolution of a sophisticated software environment.

Stage 0 of the initiative will consist of a year of preparation in FY83, during which the necessary organizational mechanisms will be established, detailed planning will be conducted, initial studies launched, and requests for proposals developed. The Ada Program has already established the sociological and technological basis for sharing tools, and will be a cornerstone for this initiative. "With Ada serving as a focus during the early stages, the initiative is responsive to recent congressional direction to accelerate adoption of Ada."

Stage 1 will focus on consolidation of demonstrated practices, educational programs, and other tools to structure an environment consistent with the state of the art.

Stage 2 will focus on enhancement of the environment adopted in Stage 1. Techniques and technology that undergo refinement during Stage 1 will be introduced in Stage 2. Stage 3 will focus on transition of the initiative and funding responsibility to a steady state. The environment will undergo a stage of transition that is either evolutionary or revolutionary.

The initiative will be managed by a vertical organization. A directorate will be established under the DUSD (R&AT) with representatives assigned from each of the services. A Software Engineering Institute will be established to bridge the gap between R&D activities which experiment with new techniques in a constrained domain and exploitation of those techniques on real systems. The Institute will be supported by DoD and will be composed of both a permanent staff and a visiting staff drawn from the DoD, industry, and academic communities.

Recent studies have recommended that DoD undertake a significant effort to improve the state of practice in the acquisition, management, development, and support of computer software for military systems. This document establishes overall objectives and implementation plans for such an effort: the DoD Software Initiative. Computer software is a two-edged sword in that it can cause critical failures of our future military systems because it is still an immature field.

The problems of computer software are not just due to an inadequate technology base; they include "inappropriate acquisition and management practices and an increasing shortage of expertise."

The microelectronic revolution has dramatically improved the cost/performance ratio of computers. This improvement has been so great that embedded computer systems (ECS) are now the primary means of introducing new capabilities and sophistication into our military systems. Furthermore, the reliance on software has increased because, when system modifications are required, software changes are easier and less costly to make than physical system changes. "Hardware changes cost fifty times as much as software changes and took three times as long to make" (in the F-111 A/E and F111 D/F programs).

There are difficulties that hinder exploiting the advantages of computer software. There is inconsistency in management practices and supporting technology requiring project-specific support facilities to be developed and maintained. A fundamental difficulty lies in poorly defined or changing requirements. In extreme cases, projects have been abandoned after delivery because they are inappropriate to users' operational needs.

Because the difficulties are often technological, the technical community has a responsibility for solutions. Furthermore, the U.S.'s economic survival lies in maintaining leadership in software and systems technology. The lead in computer technology requires not only a strong hardware base, but also the complementary software and systems technology to exploit the hardware.

The emphasis will be on technology transfer, which will be an important responsibility of the Software Engineering Institute. A set of "initial high level plans to serve as a baseline from which a set of fully detailed and integrated plans may evolve" is presented. This task plan addresses the development of "quantifiable indices of merit that can support comparisons and evaluations of people, software products, and the processes associated with software development and support."

In order to increase human resources skill levels available to DoD, career incentives to motivate software personnel will be provided. These incentives will be designed to reward software engineering skills and the application of Ada/APSE-related tools. Funds will be provided to improve software engineering curricula, pay for support staff, and upgrade computing facilities at participating educational institutions. The human resource base available to DoD will be expanded by increasing the flow of qualified software engineering university graduates, and augmenting the capabilities of lower skilled personnel with knowledge-based expert systems.

The expected rapid advancement of both hardware and software technology over the next decade relates to the systems tasks focus on reliability and architectures. The recent emergence of VLSI technology raises the question of which parts of a system should be implemented in hardware and which parts in software.

Other tasks proposed by the initiative include:

- Conducting application-specific demonstrations,
- Improving software acquisition procedures,
- Addressing techniques in human engineering,
- Defining disciplined methods,
- Developing powerful, automated environments,
- Addressing numerous related problems and issues.

This initiative augments the current low level of funding for software related research development and improvement in DoD. The initiative expands or accelerates many existing activities. The primary responsibility for the program lies with the Deputy Under Secretary of Defense for Research and Engineering (Research and Advanced Technology), who will chair an executive advisory committee with members designated by the Military Departments and appropriate Defense Agencies. The estimated budget for this initiative ranges from 5.75 million dollars (FY84 Dollars) in 1983 to 57 million dollars in FY88 for a total of 227.75 million dollars.

<u>Embedded Computer Resources and the DSARC Process:</u> Revised Report. 30 April 1981.

This report has the purpose of providing guidelines to assess the adequacy of embedded computer resource (ECR) planning and utilization, and is promulgated by the Department of Defense. The Defense System Acquisition Review Council (DSARC) has a level of interest in embedded computer resources that is related to the percentage of development, acquisition and support funds represented, and to the criticality of system performance and support that these resources represent.

DSARC reviews are required for all major defense systems as designated by the Secretary of Defense (DoDD 5000.1 and DoDI 5000.2 have more details on the classification of major systems). Three sections in this document address issues of Milestone I, II, and III reviews, which are held prior to entering the demonstration and validation phase, the full-scale engineering development phase, and the production and deployment phase, respectively.

By the time the Milestone I decision point has been reached, the critical period where data rights are established, a High Order Language and an Instruction Set Architecture have been chosen, and an overall ECR strategy has been largely decided, is completed. Candidate acquisition strategies should be developed prior to Milestone I. Important issues that should be addressed by Milestone I include:

1) Who will perform the analysis for reliability and maintainability, and perform independent quality and reliability assessments (DoDD 4155.1)?

2) What design reviews are planned during the life cycle? What agency has overall responsibility for the scheduling and conduct of design reviews? Will reviews be conducted in accordance with MIL-STD-1521A, or another standard?

3) How will the system requirements and design be validated prior to implementation? How will the system design be evaluated for feasibility?

4) Who in the Program Office (PO) has overall responsibility for software acquisition or for coordinating requirements with the acquisition agency? Who will develop the advanced acquisition plan for the Program Manager?

5) How will software design maturity and supportability be quantitatively assessed?

6) What is the scope of the IV&V effort? To whom will the IV&V organization report? How will the funding be handled? If performed by a contractor, when will the contract be let?

7) What are the areas of greatest risk? How will risk analysis be performed?

8) What are the critical computational and decision algorithms? What are the plans for validating these algorithms and the timing assumptions of these algorithms?

9) Who will perform design reviews for quality and for reliability and maintainability?

10) How will the processor memory capacity be determined?

11) How will timing requirements be determined?

12) How will safety margins and growth capacities for memory, processor time, and input/output capabilities be determined? How will these resources be partitioned?

13) What new technology (computer, sensor, and control) must be developed or utilized? What are the risks in such a development effort?

14) What special tasks must be performed in the demonstration and validation phase to perfect new technologies?

15) Which existing operational application and software support packages will be utilized? Are the application programs operational on the proposed computer? If not, what are the major hardware/software differences? To what extent have the contractor's personnel used these packages previously?

16) What hardware and/or software will be Government Furnished Equipment (GFE)? What hardware and/or software will be Contractor Furnished Equipment (CFE)?

17) How were the percentages of GFE and CFE determined? If there is a mix of GFE and CFE, who is responsible for solving system integration problems?

18) When and where will the final acceptance of the embedded computer resources be made? Who will determine whether the system is acceptable?

Milestone II is reached when the demonstration and validation activity has been completed and a recommendation on the preferred system for full-scale development can be made. Important issues that should be addressed by Milestone II include:

1) How were the requirements for computer resources, including software and its support documentation, validated?

2) How was risk analysis performed?

3) How will you ensure that the planned computer resources will meet stated operational requirements?

4) Has a Computer Resources Management Plan been written? By whom? Has it been approved? How and when will the plan be updated?

5) How will the computer resources be integrated into the total system?

6) How will the overall system quality be determined?

7) When will the system and program designs be baselined?

8) How will software testing be performed? What levels of testing will be employed? Will an independent analysis and evaluation be accomplished?

9) How will you ensure the test data is representative of the total range of data and operational conditions that the system might encounter?

10) Are the software module test plans and software module test procedures adequate?

11) How will testing be used to clearly identify deficiencies as software or hardware related? How will the determination of whether errors are caused by hardware or software be made? How will regression testing be performed?

12) Are "test beds" or "hot benches" required to adequately test software? Will they become government property after testing is complete? If not, does the government have equivalent integration and testing facilities available? What "test bed" documentation is listed as a contract deliverable item(s)?

13) How will software modules be interfaced with one another? How will these interfaces be tested? How will software be integrated and tested as part of the system?

14) What critical questions and areas of risk still need resolving by testing? What are the test plans and milestones for resolving these problems?

15) How will test-related documentation and media be maintained to allow repeatability of tests? How will support documentation be maintained to allow traceability?

16) What test and calibration software documentation and media are listed as contract deliverables?

17) How will verification and validation be performed? Who will perform it?

18) How will you assure the software architecture is modular?

19) How will you assure that the "top-down" software development methodology and structured programming will be used?

20) What programming standards and conventions will be used? How will they be enforced?

21) Which automatic debugging tools will be used during program development? Were they developed during the program? Are they deliverable?

22) How will error data be defined, collected, analyzed, and reported?

23) How will the software be integrated with the hardware during full-scale engineering development?

24) How will software be documented as it proceeds from concept to design to the final operational system?

25) Will Automatic Test Pattern Generators be used for support? If so, are they proprietary? How will they be maintained? What support documentation is contract deliverable? How will it be validated?

26) What is the government's mechanism to make an independent assessment of the software?

The Milestone III decision point is reached when a production recommendation for the system can be made. Important issues that should be addressed by Milestone III include:

1) What are the results of the latest series of operational tests (on the entire weapon system)? Where are the current tests in relationship to the overall test plan?

2) What is the profile of the last three months of Discrepancy forms and Software Change Requests? How many discrepancies are still to be corrected? How is the error data collected and analyzed?

3) How much of the recent software change activity has been due to program errors and how much has been due to change in requirements? Were changes in requirements due to increased or decreased requirements? Who has the authority to change software requirements?

4) How has delivered code been verified to conform to original software design? Who prepared test data for the validation? How has delivered code been shown to satisfy original operational requirements? How was the support documentation validated?

5) How was hardware/software integration and validation performed?

6) How was software maturity (versus design maturity) measured during development?

7) How can the completion of software development be shown quantitatively?

8) Are there any "lessons learned" that should be passed on? What process will be used?

Computer Resources are defined to include computer data, hardware, programs, resource documentation, personnel, supplies, contractual services, and software, including support software, utility programs, test software, and operational software. The Embedded Computer is defined to be a computer that is incorporated as an integral part of, dedicated to, or required for direct support of, or for the upgrading or modification of, major or less-than-major systems.

Definitions for Software Maintenance, Software Modification, Validation, and Verification of Computer Programs are provided, among other important concepts. A matrix shows available regulations and standards that pertain to various computer resource topics, and a form is provided which can be used to provide suggestions to improve the guidebook.

Proceedings of the Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management: Computer Software Management Subgroup/Second Software Workshop. 1 November 1981.

This was a continuation of the documentation panel of the first software workshop held in April 1979. The original panel examined the existing software documentation arena, and developed overview descriptions for what they considered to be a comprehensive set of documents that could be used for DoD contracts. Their recommendations for further action included preparation of Data Item Descriptions (DIDs) for these documents, the preparation of modifications to existing military standards or the creation of a new one, "from which the DIDs can be evolved", and the preparation of guidelines to help program managers select an appropriate subset of documents for specific contracts. The first task has been completed via contract, and a contract has been awarded to accomplish the second task.

Five panels, for Software Documentation, Hardware/Software/Firmware, Standardization & Accreditation, Software Cost, and Software Reusability, were convened and each report is included separately in this document. Panel A, the Software Documentation group, pursued four objectives:

a) Provide recommendations to the JLC for developing project management guidelines for the selection of software documentation;

b) Clarify the relationship between the DoD acquisition life-cycle (milestones, phases) and the JLC-list of software documents;

c) Provide recommendations to the JLC concerning the addition, deletion or modification of documents in the JLC-list of software documents;

d) Provide recommendations to the JLC for implementing the standard set of software documents within the OSD/JLC/Services.

The initial objectives provided to this panel were to develop guidelines for project managers to help select the appropriate subset of documents for their projects, prepare draft implementation plans for the DIDs, and recommend a method of evaluation. The panel determined that several issues needed to be clarified or resolved in order to meet the more long-term objectives of acquisition management guidelines and DID implementation plans. Furthermore, the alloted time of three days was insufficient to meet these objectives.

Four subpanels were formed to address issues of deep-seated concern, which were:

- Guidelines for Acquisition Managers;
- Life Cycle;
- Document Set and DIDs;
- Implementation Plan.

The first subpanel recommended that guidelines be developed via contract, and directed their efforts at drafting a Statement of Work (SOW) for such an effort. The subpanel agreed that DID tailoring should be discouraged, on the basis of the belief that document standardization is the most important concept behind the use of a single DID set, and that prohibiton of tailoring does not prohibit offerors or acquisition managers from stating particular sections are not applicable if such is the case.

This subpanel had three concerns, regarding the question of DID selection. It was agreed that some of the documents had considerable information overlap, not all of the documents were necessary on every project, and acquisition managers, when in doubt as to what documents to require, would tend to procure all of them. A selection matrix, flow chart or selection algorithm in a guidebook form is recommended for appropriate DIDs versus project characteristics.

The second subpanel, Life Cycle, deliberated the necessity of including the Software Specification Review (SSR) and the Test Readiness (TRR) in the new DID set, decided that this was not their charter, and concluded by examining the role of these documents in the life cycle, projecting them against the life cycle chart, showing when each should be available in preliminary form, baselined, or modified. Other conclusions regarding details of the life cycle and timing were also drawn.

Subpanel three, Document Set and DIDs, was tasked with reviewing the results of the JLC DID development contract. This was supplemented by panel members' knowledge of existing software documents. The first task was to review the JLC software documentation list for sufficiency. The panel concluded that three documents should be deleted from the list (two of these are government-generated and do not belong in the contractor set of documents), some changes were necessary to those remaining, and three should be added, a Firmware Support Data Document, an Operational Concept Document, and a Requirements Traceability Matrix.

Subpanel four, Implementation Plan, had the objective of examining the problem of bringing the revised DIDs developed under the DID study panel to the point of official approval. The on-going OSD Standardization Program is the program within DoD that should act as the focal point for administering the approval of new documentation. The ultimate plan for document implementation should validate the effectiveness by choosing on-contract pilot projects. Transition to the new MIL-STD and DIDs should be done using a pilot program in conjunction with a phased approach.

Panel B addressed the issue of Hardware/Software/Firmware Configuration Item Selection Criteria with the objective of developing a set of criteria to aid in the selection and documentation process, and recommending an approach for defining firmware/software categories and support documentation requirements.

In order to develop a process for recognizing the hardware intensive or software intensive nature of firmware Configuration Items (CIs), the panel pursued the following objectives:

a) To conduct a top down analysis of technical and management considerations important in the selection of systems hardware, software and firmware components;

b) To establish technical, programmable and management guidelines/criteria for CI/CPCI selection and treatment;

c) To test the criteria against representative hardware/firmware/software architecture for adequacy and clarity;

d) To define sensible categories of reprogrammable CIs for treatment of their software nature as less than full CPCIs;

e) To review recommended firmware DIDs and define documentation requirements for the different reprogrammable CI categories.

Recommended actions included adoption of standards relating to current and future perceived hardware intensive applications of firmware, and documentation for firmware. The use of HOLs on firmware is encouraged, but it is recommended that HOL direction on hardware intensive firmware not be imposed. An element in a system must be identified as a separate CI/CPCI if failure of that element would adversely affect security, human safety, the accomplishment of a mission, or nuclear safety. Continued work in the CI/CPCI was recommended to provide guidance for selection, coordinate the effort in making recommendations with on-going JLC documentation and standards development activities, and plan to review and develop a DID firmware applicability matrix.

Panel C, the panel on Standardization and Accreditation of Computer Architecture, had the objective of evaluating the potential for utilizing accreditation of computer architectures as a viable tri-service computer acquisition strategy. Accreditation has been promoted as an alternative computer acquisition strategy to solve the problems resulting from the acquisition of systems with an approved standard, yet technologically obsolete, computer. Accreditation has the goals of stimulating competition in production, easing technology insertion, increasing flexibility of choice for program managers, shortening the acquisition cycle, and minimizing cost of ownership.

Accreditation is an acquisition strategy by which a product is certified to be suitable for service use in accordance with documented criteria. It was decided that acquisition strategies must be evaluated by specific criteria in order to focus on their relative merits. The criteria include reliability and maintainability, effect on logistics, effect on personnel, sources for development and qualification funding, promotion of a competitive environment, effect of shortening the acquisition cycle, life cycle cost, product availability, and capability of achieving technological currency.

This panel was divided into three sub-panels with responsibilities for a group of related criteria. Some of their conclusions include:

1) Except for reliability and maintainability, the more restrictive acquisition strategies favorably improve the impact that the acquisition strategy has on the measures of comparison. Further, increasing the level of standardization also has a favorable effect on the criteria considered.

2) Reliability and Maintainability (RAM) is the exeption to the first conclusion. The reason is that the less restrictive the acquisition strategy, the greater the potential for contractors to be innovative in using the most current technology to achieve improved levels of reliability.

The entire panel had substantial disagreement regarding the degree of industrial concentration which would take place. One school of thought held that industry would respond to the certain knowledge that its participation will be rewarded, while the other school believed that the greater the opportunity to make a sale (laissez faire as the extreme), the greater will be industry's willingness to risk investment.

The panel recommended that studies of computer designs be conducted to determine the impact of standardization level, including those designs capable of direct HOL execution. They recommended that the data base of acquisition/accreditation issues developed at Monterey II be used as a basis for further study, leading to the selection of computer acquisition strategies best able to achieve the benefits of standardization.

Panel D addressed the issues of software costs with the objective of evaluating existing software cost estimating models and recommending an approach to improve software cost estimating methodology. They produced the following conclusions:

- The current cost estimation models have insufficient accuracy for JLC application.

- Performance of current models varies greatly from one development environment to another.

- Current models do not cover all of the life cycle phases of a software product in the required level of detail.

- Complicated models have not proven to perform better than very simple models.

- The burden of an accurate estimate is on the user, and the user must do extensive calibration and tuning of the models to minimize estimation error.

- Current cost estimation models are better able to satisfy needs of JLC early in acquisition life cycle.

- No one current cost estimation model satisfies all of the JLC cost estimation needs.

- Reliable historical data for model development or validation is almost non-existent.

- A software cost estimation methodology should be developed which covers life cycle phases completely by providing the required cost estimation information to the user based on data available at that time in the acquisition life cycle.

- Basic research should be conducted into techniques, determining parameters that characterize the software development environment, and the influence of application environment upon cost model accuracy.

- No more surveys appear to be required, additional evaluation of why the current models perform differently in different application and development environments would provide additional insight.

- An estimate cost and schedule should be associated with specific software updates.

- Model requirements should be developed in the areas of input and output parameters, and refined to correlate specific requirements with each area of model application.

- Additional research in the area of software metrics is needed to define software attributes which are the cost drivers over the life cycle.

- Data collection activities must be established using the metrics in an organized and standardized manner.

- Data collection should be automated and outputs provided in machine readable format.

- A central repository should be designated for storing and analyzing software cost estimation (SCE) data.

- MIL-STD-881A should be modified to permit a work breakdown structure that supports data collection and analysis.

- The SCE methodology requirement needs to be integrated into the Government's technical performance measurement system.

- The SCE methodologies should not be applied to progress payment determinations, which are relatively inaccurate and only predictive in nature.

- SCE methodology should not be employed in the fee determination process as it is predictive and cannot adequately support evaluations of predetermined baselines and criteria.

- SCE data should be collected at each formal program review point.

- The JLC should issue a policy that implements a SCE direction, making existing SCE technology usable by program managers.

The four recommendations developed by the panel were:

1) The JLC should not adapt any existing SCE model, as none of those existing are sufficient to adapt as an embedded computer system standard.

2) A judicious use of SCE models and methodologies can improve acquisition and management of software, and a guidebook should be developed for program offices to systematically qualify models and methodologies.

3) The JLC should sponsor a program to develop and implement an improved SCE methodology.

4) A SCE Data Base should be established to support improved software cost estimating.

The final panel, on Software Reusability, was convened to evaluate whether reusability represents a potentially valuable concept to reduce cost and elapsed time to develop embedded computer system software. Standardization in the hardware arena has yielded DoD gains in various aspects of life cycle costs. Three types of reusability were addressed:

1) Reuse of functional software systems across multiple configurations of the same basic system.

2) Reuse of generic software components for different applications.

3) Reuse of prototype and development software during the evolution of a system through its life cycle.

The panel defined reusable software to be "existing software, including specification, design, code, and/or documentation, which can be employed or adapted, in part or total, into a new end use." Four subpanels were organized to address the following issues:

1) How to design and build reusable software.

2) Managing a "ported" development.

3) How to employ existing software and what can be learned from past experience.

4) Implications on DoD policy and acquisition practice and strategy.

Language issues must be considered as a primary factor among the technical factors which influence the degree to which software is reusable. Despite the extent to which the use of Ada will alleviate the problem of proliferating languages, dialects and processors, "the language problem may continue to persist unless very strict controls are implemented and enforced with respect to both the Ada language and its compiler(s)." Dialects of Ada must not be permitted to occur.

There was a consensus among this group that attempts to reuse present-day existing software are likely to meet with failure. New software must be manufactured with a requirement that it is to be reused. Software unit packaging, modularity rules, coding standards and information binding concepts require exploration and standardization. Interface criteria, identification of functional units, specification languages and tools, support tools environments, and common language subsets must all be standardized. Strict configuration control must be maintained for the ultimate set of standard languages. Traceability methods/tools should be investigated and developed so that system requirements can be traceable down to a source statement, and a precise record of the module derivation can be kept.

The next subpanel recommended that:

1) CRM should define a technology transfer program which incorporates lessons learned in business applications to Weapon System Programs.

2) JLC should provide a government investment plan for reusable software technology development.

3) JLC should determine the feasibility of standardizing on a language and methodology for system and software requirement analyses.

4) Existing DoD guidebooks should be updated to address technical problems and remedies in the planning, award, and performance of contracts where the reuse of software is anticipated.

5) CRM should sponsor the formation of User Groups in areas considered candidates for reusability.

Other final recommendations of the panel include revision of military standards, instructions, directives, guidebooks and regulations to identify changes which must be made to include the concept of reusability. Incentives should be provided to DoD PM's and contractors for compliance with reusability concepts. Current documentation standards should be investigated to achieve reusability. Further attention is required for the transition to Ada from current languages. One suggestion is to have the Ada Program Support Environment (APSE) support multiple languages.

The Report of the Army Science Board (ASB) Ad Hoc Subgroup on Testing of Electronic Systems: 16 April 1982.

This report presents four conceptual plans addressing test planning for mission critical or embedded computer resources (ECR). These plans have been prepared by DARCOM (US Army Materiel Development and Readiness Command) in coordination with TRADOC (Training and Doctrine Command) and OTEA (Operational Test and Evaluation Agency), but are currently submitted as a DARCOM position only since full concurrence could not be obtained within the timeframe allowed.

The first group issue is to achieve more orderly design/software testing in early system development. Specific recommendations are for additional effort to be devoted to the concept definition/concept evaluation/advanced development phases of system development; the improvement of Army in-house capability as "wise builders"; employ a philosophy of incremental step-by-step design/testing during engineering development; place additional emphasis on hardware and software subsystem testing and on hardware/software integration; the performance of formal reviews for hardware and software design status throughout engineering development; and complete and evaluate development tests prior to the related phase of operational testing. Furthermore, it should be recognized that higher-than-normal funding in early program stages, with effective program management, can be expected to lead to reduced life-cycle costs and shortened time scales; additional emphasis should be given to early establishment and documentation of quantitative, "testable" system requirements; software designs should be required to be testable at module and subsystem levels as well as on an overall system basis; and software designs should be directly relatable to system requirements. "Program plans should include module, subsystem and system-level software tests in all phases of system design (with adequate funding provided)".

The history of weapon development has been that little lead-time was needed for successful completion of test programs; methods and testing facilities were in place at the proving grounds, the developer merely had to arrange for the item to be at the proving ground and the test could start with very little preparation. Complex electronic systems have changed these precedents, and require long lead-times for reviewing documentation and digesting tasks and functions that must be performed.

Testing is an integral part of every stage of the development cycle. The necessary functions of testing include testability of requirements/specifications, requirements traceability, software documentation completeness, and test planning for objectives and instrumentation and tools. The user needs to participate in the early system design and development process. The system Materiel Developer should produce the A-level specification in conjunction with the Combat Developer.

The government's software testing interests are rarely served when the Developing Contractor is hired to draft all specs and develop all system elements to include software. If more than one contractor is involved, the changing of contractor from drafting specs to writing code forces visibility and good documentation. "The developing contractor need not be the software producer."

The Independent Software Verification and Validation (V&V) Tester serves as a close observer and sometimes performs tests at the module, module integration and software system integration levels. "Much care must be used in drafting the prime software contractor to allow the Independent Software V&V Tester free access to all required information, documentation, test tools, and code."

Roles of the System DT Tester, the Combat Developer, Software Maintainer, DT Evaluator, Operational Tester, and the Operational Evaluator are described in detail. Summary recommendations include reviewing appropriate Army regulations with emphasis on early involvement of T&E agencies, incremental design/test philosophy, requirement of system simulation, use of suitably revised MIL-STDs-490 and 1679, delivery of software documentation, early identification of DT and OT test issues and criteria, and enhancement of the user's ability to participate in the early design/development process. It is also recommended that Army Regulations 1000-1, 70-1, and others, be rewritten to "front end load" funding to permit concentration of effort on early program stages.

Group 2 recommendations relate to parallel testing using computer-based test tools, and are as follows:

1) Automated, computer-based test tools should be developed to drive the engineering and initial production models of software-intensive systems;

2) Facilities such as MAINSITE should be designed and equipped for the special requirements of software testing; lessons learned by other Army testing agencies and other services, should be studied to assist in determining MAINSITE testing requirements;

3) To facilitate cost-effective software testing with results that can be uniformly interpreted and graded, a common library of software verification and validation tools should be developed and used on an Army-wide basis;

4) Consideration should be given to a radical change in the development/testing process, in recognition of the special characteristics of software-intensive systems, and computer-based test tools required to present the test environment should be provided by a contractor other than the system development contractor;

5) Test drivers (environment simulators) should be developed in accordance with the disciplines previously outlined for software development and testing.

These recommendations are discussed and interpreted in relation to 1) Simulators and Stimulators, 2) Large Instrumentation Facilities, and 3) a Library of Software V&V Tools.

The Group 3 Issue addresses the post DSARC III software testing. It has been found that software intensive systems have had major difficulties because testing is not conducted on true production prototypes; DT II/OT II data is often unrepresentative of production designs because DT II/OT II systems software is usually incomplete and immature; inadequate recognition of these points result in inadequate planning for design/testing follow-up between the Production Decision (DSARC III) and the Start of Production; and production testing appears to be scheduled "as needed" and is unfunded and limited in scope.

Recommendations related to these findings include the following:

1) Additional efforts should be devoted to detailed establishment of relationships between the hardware/software employed for DT II/OT II and the ultimate production designs;

2) After OT II, "visibility" relative to hardware/software status should be regarded as critically important; the need for continuation of integration tests should be recognized;

3) Follow-on evaluation on production hardware should be planned as a requirement, not on an "as needed" basis;

4) It should be recognized that software designs will be evolutionary, that integration testing will be necessary during the production phase, and that continuing visibility and adherence to design disciplines will be essential.

Inadequate emphasis on the earliest design effort is believed to be the root of most system acquisition problems. There is no substitute for a thoroughly thought-out design. This emphasis on careful design, although costing additional dollars and time initially, is justifiable, and furthermore, is in agreement with Carlucci initiative #12, Front-end Funding of Test Hardware (software). It is recommended that a design, when agreed upon, be frozen, and formal configuration management procedures begin.

Other improvements to consider include top-down software development precepts, modularization, uninterrupted dedication of personnel involved in system development, Government personnel participation in the lowest levels of testing at the contractor's facility, inclusion of all test agencies in all phases of system evaluation right from the beginning, establishment of a Traceability Matrix, DT II/OT II to Production, and establishment of standards for software languages, programming, and ancillary documentation.

The Group 4 Recommendations address the improvement of interoperability testing and coordination of planning for acquisition and operation of test facilities. The findings of this group include the following:

1) The Army does not adequately coordinate the development or use of its testing facilities.

2) The Army does not adequately plan for interoperability testing of its systems.

3) The Army does not maintain the technical continuity of personnel who are responsible for Army system developments.

It is recommended that Army testing facilities should coordinate and communicate better among themselves, as well as coordinate with other software agencies to ensure the unique requirements for software testing are met; steps should be taken to outline a plan for improving coordination of detailed facility planning and test system planning for the overall Army test community; more extensive planning for interoperability testing is needed; the Army must improve its in-house technical expertise and continuity of that expertise throughout a system's development; and, Federal Contract Research Centers (FCRC's) should be used to gain continuity, "corporate memory," and "transmittal of culture" from program to program.

Furthermore, the Army must accept the facts that personnel must be trained in the system development process on an ongoing basis; personnel must be acquired who can perform in these roles; it must realign its manpower allocations to allow for adequate personnel; and it must systematize its use of standards and traceability processes so that records and documentation are updated and maintained.

ESD-TR-78-141: Software Acquisition Management Guidebook: Series Overview. March 1978.

This guide is one in a series of Software Acquisition Management (SAM) guidebooks intended to help ESD Program Office personnel in the acquisition of embedded software for command, control and communications (C^3) systems. It serves as the introductory volume to the series, places it in the overall context of the Air Force Guidebook program, defines the intended audience, tells how the guidebooks were developed, provides a subject matter index, and identifies future guidebook requirements. Some of these SAM guidebooks are in the process of revision or being eliminated. AF ESD should be contacted to ascertain the current status of any specific guidebook.

The Command, Control and Communications (C^3) System SAM Guidebook series is one of three series designed to help Air Force Program Office (PO) personnel in the acquisition and management of embedded software procured under Air Force 800-series regulations. The other two series relate to Automated Test Equipment and Flight Simulators, and Avionics. C^3 software subsystems are large; customized and unique (normally state-of-the-art); on-line, real-time, closed-loop operations; interactive; employ a large data base component; contain significant user-tailored doctrinal software; use large, multi-site, ground-based computers; are capable of dealing with asynchronous, event-driven environments; and must be fault-tolerant with fail safe/soft and recovery attributes. The basic purpose of the C^3 series guidebooks is to communicate preferred procedures; instruct PO personnel in implementation of regulations, specifications, and standards (RSS); correlate sources of information concerning the software life-cycle; provide references, explanations and checklists to aid PO personnel; and to support training of new personnel.

The guidebooks provide three general classes of information:

- Explanations (lessons learned, common pitfalls, and mistaken assumptions);

- Checklists and descriptions of proven software acquisition management techniques which answer the questions: What are early symptoms of common SAM problems, what tools and techniques have proven effective in the past in monitoring status and re-creating and performing post-mortems on management decisions?

- References and index lists which can be used to access the formal RSS relevant to the topic presented.

Much of the guidance provided is applicable to less complex systems. In all cases it should be tailored to the needs of individual projects.

The audience of the guidebooks is intended to be Air Force PO management personnel and members of the Engineering Division that are referred to as the Software Director (SD), presumed to be responsible for managing software acquisition, with systems engineering experience ranging from basic to highly advanced.

The need for the guidebooks was motivated by dissatisfaction with software acquisition management in several areas:

- Today's RSS are written in a general manner which allows an experienced SD considerable latitude in tailoring the acquisition to the program.

- There is a lack of experienced SDs.

- Standards and regulations which are sometimes unclear or contradictory.

- Known differences between policy and practice.

- Software which is often the high risk item of C^3 procurements.

- Schedules and budget which prove to be unrealistic, or for other reasons, are not met.

"Although a large body of software development information is available in written form, it is not available as useful guidance for ready reference in concise or focused form." The experience of varied efforts within DoD to acquire defense system software is not transferred effectively to new personnel and the same mistakes tend to be repeated in software acquisition programs.

The first seven C^3 guidebooks were produced by the MITRE Corporation between FY75 and FY77, and System Development Corporation subsequently produced the rest of the guidebooks. Drafts of each guidebook were reviewed by Air Force PO personnel, the Air Force Contracting Offices, and the authors, face-to-face. Upon Air Force approval, each guidebook was disseminated to the field through Air Force channels as well as the facilities of the National Technical Information Service (NTIS).

Issues arose in the preparation of the guidebooks which were beyond the original intent and scope of the series, and are presented with the hope that they will be considered for future publications. These issues include: the system and software engineering processes, differing procurement strategies, software aspects of the system specification, the budgeting and scheduling dilemma, firmware acquisition management, and an Air Force glossary of terms ("many software-related terms are not consistently used throughout the guidebook series, throughout the RSS, or throughout the Air Force").

ESD-TR-75-85: An Air Force Guide for Monitoring and Reporting Software Development Status. September 1975.

This guide provides information for managers and technical personnel engaged in a software development project. Formal procedures, found in official regulations and manuals, and informal methods, from military, industrial, and academic experience, are discussed to provide a concise reference for the software manager. Using these procedures and methods, the manager should be able to identify the kinds of information relevant to his project, where to obtain it, how to use it to determine status, and what problems may be associated with using this information.

This guidebook is relevant to all Air Force activities responsible for planning, developing, and acquiring computer software in systems managed under the concepts of AFR 800-2. A major section of the guidebook is devoted to status-monitoring tools and status reporting. It discusses both formal and informal milestones, periodic status meetings, contractor and Government reports, interviewing, and project schedule representations.

A system acquisition life cycle will typically consist "of five major phases: Conceptual, Validation, Full-Scale Development, Production and Deployment." The major emphasis in this guidebook is providing management direction during the Validation and Full-Scale Development phases, through Formal Qualification Testing (FQT). Status monitoring is a means of providing managers with information so that areas of confidence and potential problem areas can be identified early and decisions can be made to correct deficiencies. Status is interpreted as the measure of progress toward a project goal in terms of quality and schedule.

Monitoring during the design phase is through design documentation and baselining. "The Reviewer must be capable of analyzing progress through evaluation of design and implementation documents, test plans and procedures, and test results."

Another section discusses contractual planning to ensure Government visibility, top-down and bottom-up implementation approaches, modularity, structured design, requirements traceability, structured programming, and functional organization of personnel.

There are limitations of software status monitoring and reporting. "Reliability must be designed in, it cannot be tested into a system." The impact of decisions on a software project are inversely proportional to the life cycle phase of a project, i.e. the most important decisions occur at software system requirements analysis and design time. This does not mean that testing is unimportant. Only through testing can the product be demonstrated to satisfy specification requirements. The test phase is more visible than the design phase, but because of schedule and cost, little can be done to redirect projects at test time without serious impact.

Finally, an appendix is included that provides summaries of selected status factors that can be used to evaluate project progress. Included in the appendix are discussions of documentation quality, stability of the requirements baseline, interfaces, programming languages, programming practices and standards, project complexity and operating systems. In addition, the appendix addresses data management, personnel, development facility quality, project management, and non-subjective data.

<u>ESD-TR-75-365</u>: An Air Force Guide to Contracting for Software Acquisition. January 1976.

This guidebook provides an introduction to the process of contracting for the development of computer programs acquired under Air Force 800-series regulations. It attempts to highlight those areas that deserve special attention when software is to be developed under contract. It describes the Armed Services Procurement Regulation (ASPR), "the bible for all contracting by the Department of Defense". This guidebook provides a list of references that identifies those parts of the ASPR that have a bearing on the Software Director's job, and identifies Air Force publications that may be useful. The guidebook specifically excludes commercial, ADP software from its areas of interest.

A major section is devoted to preliminary procurement planning, including selection of the basic procurement approach, formal procurement planning, and the bid package. Considerations relating to single or multiple contractor awards are described, and specifically those problems that may arise if one contractor on a multiple contract system fails to deliver on time. This section provides a checklist for the Software Director to use in planning the basic procurement approach and discusses the Advanced Procurement Plan, negotiated procurements, and Determinations and Findings. It is asserted that "ideally, the detailed design of the hardware, and preferably the hardware itself, should exist before the support software is designed," and that both should be completed before the applications software is designed.

Another section addresses contractor selection, including contracts, RFPs, proposal evaluation, contract negotiation, and contract award. The issue of proprietary documentation and data that the Air Force owns is addressed. The guide then discusses contract management, and summarizes the responsibilities of the Air Force Contracting Officer, the Procuring Contracting Officer, the Administrative Contracting Officer, the Defense Contract Administrative Services, the Air Force Plan Representative Offices, and the Software Director.

The Air Force Software Director is limited in what he can say to contractors while the advanced procurement planning is being done, but he can accept input from potential contractors on what has been accomplished on previous programs, what related developments are currently underway, and what the risk areas are in the planned program. When the RFP is released, the Air Force Software Director is directed to work with the Contracting Officer to provide answers to questions which deal with technical or management aspects of software development.

Evaluation criteria for a software source selection are based upon requirements in the RFP package. A sample set of criteria are tabulated in this guide, and address such issues as computer program design; programming and design standards; government furnished equipment; human factors, overall system design and performance; program, configuration and data management. Lastly, contract changes and contract termination are discussed.

The last section lists ASPR references that are directly applicable to this guidebook. It also abstracts regulations such as AFR 800-14 that are of particular relevance to contracting for software acquisition, and finally, an appendix provides a list of source selection do's and don'ts.

ESD-TR-76-159: An Air Force Guide to Software Documentation Requirements. June 1976.

This guide addresses those requirements for documentation of software developed in a large system acquisition, and use of that documentation. It emphasizes the determination of documentation needs, the preparation of the Contract Data Requirements List (CDRL), and the specification of Data Item Descriptions (DIDs).

A major section of this guidebook addresses software documentation requirements, and how to determine those requirements. The author describes the purpose of documentation and identifies regulations, standards, and specifications (RSS) which provide guidance on the determination of documentation requirements for software. In addition, he discusses specific factors which impact documentation requirements and presents a series of guidelines that can be used to determine documentation requirements. Factors which impact documentation requirements include: system magnitude, system complexity, duration of the software development process, multi-contractor/agency involvement, instability of system requirements, the relationship to schedule, procuring activity manning and expertise, and the cost of document preparation and updating.

Considerations in the determination of documentation requirements should follow the following steps:

- Identification of mandatory requirements;
- Solicitation of user requirements for the operation and support phase;
- Inclusion of applicable RSS;
- Incorporation of program-unique data requirements;
- Identification of any other basic documentation requirements;
- Verification that only the minimum essential documentation satisfying specific needs is acquired.

Another major section describes key software documents. Key software documents consist of "Data Items", a formal collection of information (data) acquired under the system acquisition process to support the management and technical objectives of the program. The specific content and organization of each data item is defined in a Data Item Description (DID). These DIDs include such items as the Computer Resources Integrated Support Plan (CRISP), the Computer Software Configuration Item (CSCI), the Computer Program Development Plan (CPDP), the Version Description Document (VDD), the Engineering Change Proposal (ECP), and other documentation required for design, development, integration, testing, fielding, and maintenance of software.

This section also discusses the tailoring of DIDs to meet the requirements of specific programs. In addition, it discusses alternative sets of software documentation (documents used by DoD agencies other than the Air Force) that may be applicable to some programs.

The next section of this guidebook is devoted to the contractual aspects of data item acquisition. It addresses contractual specification of the documents desired, including their content, format, delivery dates, numbers, and conditions of acceptance. This section emphasizes the relationships between the SOW, CDRL, and DIDs.

In the last section, general conclusions and recommendations regarding the determination of documentation requirements are presented, including the following conclusions:

1) There is a lack of formal guidance in the area of software documentation requirements;

2) There is no single source of guidance on 'software documentation;

3) There is a lack of guidance on requirements for and usage of documentation related to software and its acquisition;

4) Many applicable references lack definitions, and where definitions are provided or may be inferred, they introduce ambiguities and inconsistencies with respect to other references;

5) There is a general lack of detail in the DIDs;

6) The determination of the number of Computer Program Configuration Items (CPCIs) is a management decision which has a heavy impact on software documentation requirements; if a "sub-assembly" (functional area) is expected to have its own independent cycle of changes, it should be designated a separate CI. An approach to determining the most appropriate number of CPCIs must attempt to minimize CPCI interdependencies and interfaces.

Recommendations include:

1) Minimizing the number of documents identified for acquisition;

2) Defining terms and requiring references to specific sections and paragraphs in all RSS cited;

3) Acquiring documentation for all software used or developed in the system acquisition;

4) Specifying the acquisition of the Data Accession List/Internal Data to ensure access to all data generated by the contractor for his use in performing the contract;

5) Specifying the preparation and periodic updating of a glossary of terms associated with the requirements and design of the software or total system.

Three appendixes provide summary material on Data Items relevant to software acquisition.

ESD-TR-77-16: Software Acquisition Management Guidebook: Statement of Work Preparation. January 1977.

This guidebook explains the preparation of a software-related statement of work (SOW), and describes other components of Requests for Proposal (RFPs) for acquisition of electronic systems that comprise, or include, software. It contains a major section covering planning for SOW preparation and then presents model Full-Scale Development Phase SOW tasks.

The purpose of the guidebook is not to prescribe what must be done, but rather to identify issues and pitfalls, reference relevant sections of appropriate regulations, specifications and standards, and suggest alternative approaches to problems that arise for the Program Office or the Software Director.

The section on planning heavily emphasizes the use of work breakdown structures in SOW preparation. A work breakdown structure (WBS) is a hierarchical representation of the tasks and the products (e.g., equipment, software, data) that comprise an acquisition. A WBS depicts the chief order in which these tasks and products will be aggregated for purposes of cost accounting.

SOW preparation requirements, general suggestions for SOW preparation, and configuration item definition are discussed. SOW preparation requirements apply generally to SOWs for Validation Phase and Full-Scale Development Phase contracts, and on occasion to SOWs for other types of contracts. Examples of these requirements are:

- Each SOW must correspond in structure and substance to the planned contract's Preliminary Contract WBS (CWBS).

- Each SOW paragraph that prescribes contract effort must correspond to a Contract Line Item (CLIN) of the same name.

- Each Validation Phase or Full-Scale Development Phase SOW paragraph that prescribes contract effort must be identified by the Program Breakdown Code (PBC) of the corresponding Preliminary CWBS Element.

- A separate SOW paragraph must call for acquisition of each Computer Program Configuration Item (CPCI).

The section on model SOW tasks includes a table of contents and the software-related paragraphs of a hypothetical Full-Scale Development Phase SOW. The SOW presumes to prescribe the work desired from a single contractor (at the system level) to develop a postulated one-of-a-kind digital communications message switch. The SOW-prescribed tasks include interfacing the system with numerous local and remote digital data sources and sinks. The hypothetical planned contract covers site activation, support equipment, and administrative data, as well as software acquisition, computer equipment acquisition, and system engineering.

Three appendixes are included in the guidebook. The first describes the use of WBSs, and summarizes the various types of WBSs. The second discusses source selection plan requirements, and finally, the third appendix presents extensive guidance regarding the contents of RFPs.

"The Request for Proposal (RFP) is a formal document used by the Air Force to solicit proposals from a Source List of potential contractors." The RFP contains a statement, in general terms, of the criteria that the government plans to use to evaluate proposals, and the relative importance of each aspect of the proposal. A RFP for a software-related Validation Phase or Full-Scale Development Phase contract consists of three volumes. It includes contract forms and representations; solicitation instructions and conditions; evaluation factors for award; the schedule, including supplies, inspection and acceptance; general provisions, and a list of documents including the SOW, engineering drawings, security classification specification and a preliminary contract WBS.

<u>ESD-TR-77-22</u>: Software Acquisition Management Guidebook: Life Cycle Events. February 1977.

This guidebook explains the chief activities, events, products, and software-related efforts that normally occur during the life cycles of major electronic systems acquired within the framework of the 800-series of Air Force regulations and manuals.

The 800-series normally governs acquisition of computers and software which are embedded in weapons or C^3 systems. Some of this software (e.g., applications programs) may be built expressly for the weapons or C^3 system. Some (e.g., certain operational executives) may be modified versions of off-the-shelf software. The 800-series covers the research, design, development, engineering, testing, and production of tactical and strategic systems for the operational inventory. In contrast, the acquisition of off-the-shelf, commercially-marked data processing equipment and its associated support software for business type applications (payroll, logistics, personnel records, etc.) is normally governed by the 300-series of Air Force regulations and manuals. This guidebook does not address acquisition managed in accordance with the 300-series, although some of its principles may apply.

Uses anticipated for this guidebook include as a tutorial for the individual inexperienced in acquisition of large systems that include software, and as a summary of material relevant to software acqusition for those otherwise quite familiar with large system acquisition.

Major sections are devoted to the acquisition life cycle including an overview of the life cycle and one section each for the Conceptual Phase, the Validation Phase, the Full-Scale Development Phase, and the Production and Deployment Phases. Each section discusses phase objectives, initiating events, primary activities and related products, and terminating events. Another section addresses less elaborate acquisitions, and the last section summarizes the activities and products of the computer program life cycle. It distinguishes between this cycle and the system acquisition life cycle and relates the two. The computer program life cycle consists of six phases that occur sequentially for the most part, but have some overlap. These phases are identified as:

- Analysis,
- Design,
- Coding and Checkout,
- Test and Integration,
- Installation,
- Operation and Support.

AFR 800-14 defines the computer program life cycle and the goals of, activities in, and milestones for each phase. Examples of activities that occur in these phases are: requirements allocation to the computer program; the conducting of PDRs for the computer program's CPCIs; defining algorithms and computer program logic; test planning; critical design reviews (CDRs) for the computer program's CPCIs; coding; module and CPCI tests; DT&E, IOT&E, and FOT&E (defined elsewhere in this document).

The computer program life cycle may span more than one system acquisition life cycle, or occur in any one phase. The computer program life cycle for the T&E software might extend from the validation phase of the acquisition life cycle into the deployment phase.

An appendix summarizes the major specification provisions affecting software. This appendix should be compared with the similar discussions in the Configuration Management Guidebook.

ESD-TR-77-254: An Air Force Guide to Computer Program Configuration Management. August 1977.

This guide provides basic instructional and reference materials to support the application of Air Force/DoD prescribed configuration management techniques. The guidebook covers the following topics:

1. Background and introductory information, reviewing general concepts, principles, special terms, and the status of Air Force/DoD configuration management standards.

2. Requirements and criteria for selecting assemblies of computer program code to be identified as computer program configuration items (CPCIs), and includes a subsection summarizing the sources and coverage of standards for identification numbers and markings.

   Requirements and criteria "decisions should be based on experience, knowledge of the principles ... and the given system program", and not based on "'stylized' rules". The CI (originally "contract end item") is the level at which a program office exercises formal management control over the contractor in the areas of configuration management, procurement, program control, and monitoring of the contractor's technical progress. The following documents and actions are requirements that apply separately to each CPCI;

   - Specifications.

   - Proposed engineering changes, and reports of change implementation.

   - Management information reporting against the contract Work Breakdown Structure.

   - The performance of technical reviews and configuration audits -- Preliminary Design Review (PDR), Critical Design Review (CDR), Functional Configuration Audit (FCA), and Physical Configuration Audit (PCA).

- The preparation of operating and user manuals.
- Formal acceptance by the procuring activity.

Criteria are given as a shopping list, since the importance and applicability of considerations vary widely among different system programs. By definition, the CPCI must be "in a form suitable for insertion into a computer". Therefore, "computer programs to be designed for operation in different types or models of computers must be separate CPCIs". Computer programs scheduled for development, testing, or delivery at different times may be separate CPCIs. However, an earlier-developed CPCI may be expanded by an Engineering Change Proposal (ECP), or a later CPCI may be developed to incorporate and replace the earlier item. Computer programs for different system functions, such as mission, support, and off line diagnostic functions, should be separate CPCIs.

3. Specification types and forms, the specification tree, the system specification, computer program development and product specifications, other types and forms of specifications applicable to computer programs, and comparisons between software and hardware with respect to the roles of their specifications in the system acquisition cycle.

4. Requirements and procedures for processing changes to approved specifications. It identifies organizational factors, explains change classification, describes standard forms, and discusses procedures involved in the preparation and processing of change proposals, including a subsection dealing with concepts of interface control and the documentation of interfaces involving software.

5. Requirements and practices of document identification and maintenance which are significant to configuration management functions, and to formal reports/records of status for documents, change proposal s, and CPCIs.

6. Factors that arise after completion of development and initiation of the CPCI operations at a field location. Using a sample system DT&E situation for illustration, it identifies the nature of questions to be anticipated and shows how centralized controls and procedures described in preceding sections relate to that expanded framework.

7. Notes written in response to questions raised by reviewers of a draft version of the guidebook, pertaining to a few of the topics covered in preceding sections.

The bibliography and glossary list references, other guidebooks, abbreviations and standard Air Force/DoD terms as they apply to configuration management of computer programs.

ESD-TR-77-255: Acquisition Management Guidebook: Software Quality Assurance. August 1977.

This guide describes the scope of quality assurance (QA) to be (as in AFR 74-1): "A planned and systematic pattern of all actions necessary to provide adequate confidence that material, data, supplies and services conform to established technical requirements and achieve satisfactory performance."

Special attention is given to the relationship of QA to the other acquisition management disciplines, the integration of QA requirements within the system acquisition process, contractual aspects of QA, monitoring the implementation of QA requirements, common problems and proposed solutions, and pitfalls, risk areas, and danger signals as they occur during the System Acquisition Life Cycle.

Section One relates the Air Force QA program to the major milestones of the system acquisition cycle as they occur during the Conceptual, Validation, and Full-Scale Development Phases. It treats objectives, activities, and QA considerations for each phase. Discussions are supplemented by flow charts depicting major activities within each phase.

Section Two provides discussions designed to assist the PO in evaluating a contractor's proposal and the status of his software QA program. The contractor is responsible for controlling product quality and for offering to the Government for acceptance only those supplies and services that conform to contract requirements. The director of Computer Systems Engineering (MCI) at ESD is responsible for providing software support to the POs. MCI computer system personnel are assigned to the POs to assure that quality software is being developed by the responsible organizations.

Prior to the award of a Full-Scale Development Phase contract, QA activities must be conducted with the following objectives, to assure that:

- The technical and contractual requirements for the CPCI(s), data, and services are practical and enforceable.

- The delivered CPCI(s), data, and services conform to the specified technical and contractual requirements.

- The causes of user dissatisfaction and mission degradation are identified and corrected or eliminated.

Section Three is designed to assist the Software Director (SD) in evaluating a contractor's proposal and monitoring the status of a contractor's software QA program. It covers the evolving QA role within ESD and discusses specific QA aids, such as a source selection checklist used by the Computer Systems Evaluation Panel (CSEP) to assist in evaluating RFPs and proposals. The source selection checklist highlights areas which were not as thoroughly reviewed in past procurements. These areas include: schedule risks; technical risks; adequacy of the stated requirements; conformance with regula‘.ons, specifications, and standards (RSS); and computer program products that will be useful to the Air Force throughout the system life cycle.

An appendix discusses various software quality issues. It begins by discussing the term software quality, qualitatively and quantitatively. Software quality is a composite of many conflicting factors, e.g., efficiency, maintainability, reliability, testability, understandability, modifiability, and portability. Software quality is not yet measurable, in any practical sense. The subjects covered include quality vs. program delays, how much QA is enough, and independent support contractors. Risk analysis can be used to determine the amount of QA required in terms of the impact of the software on the overall system. For a system where it is difficult to design a thorough and realistic test program, an intensive software QA effort is merited.

ESD-TR-77-263: Air Force Electronic Systems Division Software Acquisition Management Guidebook: Verification. August 1977.

This guide provides direction for planning and managing the implementation of software verification concepts and requirements as they relatε to software acquisition management. It provides a review of the ver\fication practices and procedures employed by industry and set forth in relevant RSS and describes those Computer Program Configuratic̄n Item (CPCI)-oriented system engineering and test activities wnich lead to verification.

Verification is defined and distinguished from validation and certification. Verification is CPCI oriented. It begins with system and software engineering activities, which lead to CPCI definitions and to the CPCI Development Specification, and ends with the qualification of the CPCI. Validation is system oriented, begins with the System Specification and concludes at the end of System Development T&E. Certification is a user-oriented, system-level activity and occurs during Operational T&E.

One section addresses requirements verification from initial CPCI definition until authentication of the Development (Part I) Specification and verification of the contractor's CPCI DT&E plan. Contractor activities are discussed, as are PO verification activities during the Validation Phase, including determination of Validation Phase support products, SRR (System Requirements Review), and SDR (System Design Review).

The Requirements Verification section also discusses PO verification activities during the Full-Scale Development Phase, including evaluations of the contractor's Computer Program Development Plan (CPDP), authentification of the Development Specification, and review of the contractor's CPCI DT&E plan. Evaluation techniques have had "varying degrees of success in verifying performance requirements." Such techniques include:

- Simulation, the process of studying specific system characteristics by the use of models exercised over a period of time and a variety of conditions.

- Performance monitoring, the process of collecting data on the performance of an existing system for the purpose of evaluating or improving performance.

- Synthetic programs, a set of executable instructions, including I/O, files, etc., written for the purpose of representing various computer demands inherent in the system under study.

- Benchmarks, existing operational programs used for performance projection or selection evaluation of computer equipment.

- Kernels, programs written to evaluate timing information about a specific computer.

Design verification activities which occur during the Full-Scale Development Phase, contractor activities, PO activities, including Preliminary Design Review (PDR), Critical Design Review (CDR), and review of the contractor's CPCI DT&E procedures are addressed.

Computer Program Verification activities in terms of informal testing of the CPCI and its components as carried out by the contractor at his discretion, and formal testing of the CPCI as carried out by the contractor are discussed. Contractor-internal testing, including Computer Program Component (CPC) code and test, CPC incremental-integration testing, and CPCI testing, qualification testing, including Preliminary Qualification Tests (PQTs) and Formal Qualification Tests (FQT) are specifically addressed. Module and CPC-level testing uses traces, dumps, drivers, data reduction programs, and test-case generators for aids in helping the programmer locate an error in program code. Tools used for CPC integration testing include automatic execution analysis, and dynamic analysis of the system structure, a program which outputs listings of the CPCI.

An appendix describes selected commonly used support tools and techniques for computer program development and testing. The appendix stresses the applicability of these aids to distinct verification tasks. Design tools and techniques used to support definition of computer program components (CPC) performance requirements, interfaces, and data base definitions, include simulation, top-down design, use of a design language, and use of decision tables.

ESD-TR-77-326: Software Acquisition Management Guidebook: Validation and Certification. Air Force Electronics Systems Division. August 1977.

This guidebook summarizes the software acquisition implications of validation and certification. It recognizes and is compatible with Air Force 800-series regulations and related concepts. Validation is defined to be those evaluation, integration and test activities carried out at the system level to ensure that the final system satisfies the requirements of the System Specification. It is system oriented, begins with the System Specification and concludes at the end of System Development Test and Evaluation (DT&E). Software validation cannot be isolated from system validation since all evaluation and test activities that make up validation are focused at the system level.

Certification refers to the using command agreement, at the conclusion of OT&E, that the acquired system satisfies its intended operational mission. During OT&E, the system undergoes test and evaluation aimed at assuring operational effectiveness and suitability under operational conditions. Verification is the iterative process of determining whether the product of selected steps of the CPCI-development process fulfills the requirements levied by the previous step.

The guidebook describes the system engineering activities carried out to ensure that the requirements documented in the System Specification accurately respond to the operational needs called for in the Required Operational Capability. It then addresses the activities involved in integrating into the system the qualified CPCIs which were verified during FQT. At this point in the system acquisition cycle, the software has been tested and the individual CPCIs are now ready to be put together and checked out in preparation for System DT&E. The PO should have a high degree of confidence that each CPCI is functionally correct. The contractor must now demonstrate that the software performs correctly when assembled into the system in an environment which may differ markedly from that used for CPCI development and test.

Another section addresses the software-related activities involved in planning and executing a comprehensive System DT&E program. Although the objective of System DT&E is formal qualification of the system, there are unique aspects of planning and conduct which are software related and should be recognized at the beginning of the system acquisition cycle. The principal software-related items which should be included in the program management plan (PMP) and which affect DT&E planning are:

- The identification of software system validation expertise to be allocated to the PO for the management of the test program.
- Requirements for simulation capabilities to support System DT&E, if needed for system testing inputs.
- Requirements for a system test facility, if necessary, based on both system DT&E and planned system deployment support requirements.
- A realistic master schedule containing all the major milestones, key events, and critical actions related to software acquisition.
- An identification of required external interfaces to be accommodated by the system.
- A discussion of growth and spare capacity requirements.
- An identification of support required from outside agencies.

System certification starts the Deployment Phase and indicates the operational suitability of the system. While certification is the responsibility of the using command, the PO is involved in planning and preparing the Operational Test and Evaluation which concludes with certification, just as the operating command may support System DT&E with liason personnel, facilities, test data, and general assistance in evaluating test results.

The last section discusses the software-related requirements of system turn-over, transfer of management responsibility, and system certification. System turnover agreements must be formulated early in the acquisition cycle. In the turnover agreement a version description should be included, listing all system elements and computer resource elements, and details of all deficiencies and exceptions to be corrected and delivered. For the system to be certified, the using command must agree, at the conclusion of OT&E, that the acquired system satisfies its intended operational mission.

ESD-TR-77-327: Software Acquisition Management Guidebook: Software Maintenance. October 1977.

This guide has a scope that is limited to those acquisition and development activities, occuring throughout the Software Acquisition Management (SAM) cycle, which impact software maintenance. It includes discussions of system turnover to the using command and the transfer of program management responsibility to the supporting command. The computer program life cycle is also considered. Most of the information provided in this guidebook covers the implementing command's responsibilities during the SAM cycle. However, software maintenance during the Deployment Phase is also discussed to provide the background for proper planning. Concepts of quality computer program design and development are discussed, as well as Regulations, Specifications, and Standards (RSS). Quality computer program design and development should emphasize the following:

- A limited number of interfaces between modules;
- Communication between modules limited to the defined interfaces;
- Well documented, easy to understand design;
- Limited equipment interfaces;
- A controlled data base;
- Limited access to the data base by each module;
- Programming style for clarity of function and ease of verification;
- Separate modules for input, output, and computation of functions.

Additionally, the increasing cost of fixing software errors during the advance of the program life cycle is described.

This guidebook emphasizes the specification and procurement of maintainable software, including procurement of the facilities, support tools, and documentation necessary to support software maintenance activities. Major sections are devoted to the acquisition of maintainable software and to applicable RSS. The former addresses: the definition and specification of maintainable software; monitoring the evolving software and evaluating contractor effectiveness; design change and error correction during subsystem DT&E; and transfer and turnover. The latter discussed those RSS that impact software maintenance. In this guidebook, the definition of software maintenance includes the ability to modify the software. Therefore, this section relates some of the configuration management RSS to software maintenance.

The guidebook includes an appendix devoted to designing maintainable software, which describes the properties of maintainable software, including conceptual organization, modular design, self-monitoring computer programs, program hooks for further extensions, and design methodology. In addition, it covers specific techniques that facilitate software modification, including computer program legibility, parametric organization, stable code, and development methodology or structured programming.

ESD-TR-78-117: Software Acquisition Management Guidebook: Reviews and Audits. November 1977.

This guide provides detailed guidance concerning the use of engineering design reviews and configuration management audits as tools to monitor a developing organization's technical progress. The following formal reviews and audits are defined and described: System Requirements Review (SRR), System Design Review (SDR), Preliminary Design Review (PDR), Critical Design Review (CDR), Functional and Physical Configuration Audit (FCA & PCA), and the Formal Qualification Review (FQR).

Major sections are devoted to general requirements for reviews and audits, engineering design reviews, and configuration management audits. Each of these sections discusses such topics as location and scheduling, the responsibilities of participating organizations, and the conduct of reviews and audits. In addition, the materials to be reviewed or audited are listed and suggested evaluation criteria presented. The data required for CDR is detailed in the agenda for the review. The developer prepares the agenda and submits it to the PO for approval. The technical information is the same data contained in a CPCI product (PART II) specification, with the exception of the program listings and the results of software engineering studies that were conducted to arrive at the CPC-design decisions. Evaluation criteria for CPCI development specifications include assuring that the development specifications contain performance requirements rather than computer program design information, so that a non-programmer can evaluate the development specification; assuring that the operational CPCI development specification reflects an understanding of the operational mission; and assuring that the requirements are defined at a level of detail sufficient to initiate the CPCI design effort.

Another section presents modified sample forms from MIL-STD-1521A (USAF) which can be used to identify and record critical data during reviews and audits, and finally, a section on the more common reviews and audits problem areas is included, dealing with responsibility and authority, the CPCI (Part I) Development Specification, and the scheduling of PDRs and CDRs. This guidebook also has a glossary.

ESD-TR-78-178: Software Acquisition Management Guidebook: Regulations, Specifications, and Standards. November 1978.

This guide serves as an introduction to the plethora of military and Government documents pertaining to software acquisition management and development. It identifies the existing types of official documents and provides a table of guides, lists, catalogs, and indexes to the various forms of military and Government publications. It ends with two indexes, the first of which lists keywords with associated regulations, specifications, and standards (RSS). The second index reverses the first and lists RSS with associated key words.

The RSS guidebook applies to software, whether it is acquired as an entity or as a portion of a larger system. Therefore, even though many of the documents cited do not specifically refer to software management or development tasks, the software element of a system assumes the same measures of management control and development quality as does the system. Further, some referenced publications deal specifically with software while others apply to software on a broader scale (e.g., cost control systems, or work breakdown structures (WBSs)).

A major section of the RSS guidebook differentiates between the types of programs governed by Air Force 300-series regulations and those governed by Air Force 800-series regulations. This section provides lists of 300-series and 800-series regulations and manuals and identifies distinguishing characteristics between the two series.

Two other major sections list documents pertaining to software acquisition management and software development tasks, while an appendix presents abstracts of selected software acquisition RSS.

ESD-TR-81-128: An Air Force Guide to the System Specification. January 1981.

This guidebook differs from others in the series in that its topic relates more to the system as a whole than to the software elements of the system. There is a growing recognition that the prominence of software, especially in ground electronic systems, has implications for management at the system level.

The System Specification (Type A spec) is the designated source of basic requirements for the system software functions and performance, and many of the problems associated with software acquisition in systems have been traceable to inadequacies in those basic requirements. This guidebook's material is addressed primarily to members of system Program Offices who are responsible for software aspects of system programs, and in part to supporting contractor personnel.

Furthermore, since a PO's approach to the System Specification is constrained by basic program management policies that are determined at or above the Program Manager level, the discussions in this guidebook also touch on certain areas which merit attention by higher-level managers and decision makers.

The System Specification development process is described, with emphasis being placed on describing the levels and nature of system engineering studies that are normally needed, but not yet typical in practice, to develop comprehensive requirements information in signficiant areas. The manner in which the technical process can be planned and managed systematically within the framework of program management policies and milestones established in such documents as AFR 800-2 is outlined.

The section on issues and problem areas identifies those areas where problems have been encountered pertaining to development and uses of the system specification in electronic system programs, including the intended functions of the system specification; current problems associated with PO manpower and increasing prominence of commercial components; and those problems that have impacted factors of risk.

An appendix is provided as a preliminary basis for development of guidance pertaining to preparation of the System Specification as described in MIL-STD-490. Also provided is a sample system specification paragraph dealing with design and construction standards for computer programs developed at ESD, and sample functional flow block diagrams. The System Specification paragraph provides requirements for computer programs that are comparable to the types of design and construction standards provided in other parts of the specification as a whole for items of system equipment. Functional flow block diagrams are a prominent form of system engineering documentation which is normally contained or referenced in a system specification. These diagrams show the flow of functions required to carry out the system mission.

Air Force Electronic Systems Division Pamphlet 800-4: Acquisition Management: Statement of Work Preparation Guide.
15 January 1979.

The Statement of Work (SOW) is a vital management tool and an important contractual instrument. This document provides detailed guidance for each functional/technical task involved in preparing the SOW. Each task has been aligned with a prescribed Work Breakdown Structure (WBS) code to facilitate cost proposal preparation, analysis, and tracking. The guidance in each task addresses each of the areas of instructions to offerors, system/equipment specifications and data requirements.

Sample or model SOWs are provided in this pamphlet as guides. ESD policies and procedures for SOW preparation are given, with itemization of applicable documents. "Specifications and Standards are the heart of the SOW. Minimum application and tailoring to program needs are required per DoDD 4120.21."

Test and Evaluation concepts are addressed, and DT&E and OT&E are defined. "Test and Evaluation shall be applied on all programs in accordance with the individual program requirements." It is emphasized that the program office (PO) must provide for early test planning and get the operating and supporting commands and the Air Force Test and Evaluation Center (AFTEC) involved with test planning early in the program. A T&E working group should be organized to accomplish test planning. Testing must be specified on the life cycle phases of conceptualization, validation, full-scale development, and production.

Statements of design requirements are specified, as are methods of verifying the design requirements. Sample system specifications are given. Methods of verification include quality conformance verification, inspection, analysis, demonstration and test.

Design requirements for Computer Programs should not be included in the SOW. The System Specification addresses such issues as general requirements, computer program structure, top down design, structured coding, programming languages, firmware requirements, program generation, and character set standards. General instructions to the contractor shall include the development of computer programs required to satisfy the design and performance requirements delineated in the system specification. The contractor's approach to computer program development shall conform to the government approved Computer Program Development Plan. "Care must be taken, when preparing the SOW, to ensure that the level of management and detail does not constrain the contractor to the extent that cost/efficiency of the computer programs will be adversely impacted." Data Items most frequently used for the management of computer programs are listed. A sample SOW is given for guidance only, and includes such issues as sizing and timing analysis, data base architecture, software design criteria and decisions, algorithm description, and results of the investigations to be delivered by the contractor in accordance with the CDRL.

<u>Model Statement of Work Task for Software Development:</u> U.S. Air Force, Electronic Systems Division (ESD). March 1979.

This document specifies general and specific requirements for the SOW, and gives samples of how it should be worded. "The contractor shall develop the project software to satisfy the design and performance requirements established in Specification No. XXX."

Topics addressed include Software Development Technologies/Management Practices, Life Cycle Activities, Analyses, Sizing/Timing Analysis, Data Base Architecture Analysis, and Risk Analysis. Algorithms must be described. Design criteria and decisions, hardware selection criteria and decisions, CPCI organization and decomposition, system integration testing, documentation for design, QA, delivery, installation, operation, etc. must be included. Support software must be identified. Reviews, both formal and informal, of software development must be conducted.

Management Guide For Independent Verification and Validation: Air Force Space Division (SD), Directorate of Logistics and Acquisition Support. August 1980.

This guide was prepared to provide an overview of IV&V as performed by SD program offices. Its purpose is to provide insight into how the need for an IV&V contractor is established, the scope of the IV&V effort relative to the size of the project itself, how the RFP should be written, what CDRL items to call for, what to look for in source selections, and how to manage the IV&V contract. IV&V spans activities throughout the system acquisition life cycle. The purpose of these activities is not to avoid the occurrence of all software bugs, but to eliminate programming errors which can lead to catastrophic results such as loss of life or mission failure, or to less compelling but still serious consequences, such as equipment damage and negative economic impacts.

Some SD programs that have used IV&V are: Space Defense, Defense Meteorological Satellite, Space Shuttle (Interim Upper Stage), and Global Positioning System. These programs are recommended as sources for first-hand, lessons learned experience.

The consequences of computer programming errors establishes whether or not IV&V is indicated for a project. If there is some chance that an undetected error could cause loss of life or personnel injury, jeopardize mission success, damage equipment, or lead to waste of economic resources, IV&V is required on the project. A numerical IV&V value can be determined by summing criticality values from the following decision table.

CRITICALITY CLASS	ASSIGNED VALUE	PROBABILITY OF OCCURRENCE	ASSIGNED VALUE
Negligible	1	Impossible	0
Marginal	2	Improbable	1
Critical	3	Probable	2
Catastrophic	4	Frequent	3

For every factor/subsystem combination, a criticality value can be obtained from the product of the software error criticality class value times the probability of occurrence, summed over all factors, then divided by the number of factors. The level of IV&V effort is determined from that number through the use of the following selection chart.

IV&V Level Selection Chart

IV&V VALUE	SUGGESTED IV&V LEVEL
0-2	None - C
2-3	C
3-6	B
6-12	A

Where:

Level C: Constructively critique developer's documentation, participate in milestone reviews; monitor development.

Level B: Same effort as in Level C. In addition, using appropriate tools as necessary: Analyze selected critical functions, spot check design performance, conduct limited testing, perform selected audits.

Level A: Same effort as in Level B. In addition, using appropriate tools as necessary: Independently analyze requirements and design, rederive key algorithms, confirm technical adequacy, independently test and evaluate operational software, conduct stress tests and special studies, and support configuration and data management.

Cost of IV&V is addressed, as well as what to put in the RFP, how to identify a good IV&V Contractor, how to get the desired IV&V evaluation data, and how to evaluate the IV&V proposal data. Pointers on establishing and maintaining typical IV&V/Software Developer/SPO Interactions are provided for the SD IV&V manager. A chapter is devoted to lessons learned, potential IV&V management problems and suggested actions.

Guide to the Management of Embedded Computer Resources: Air Force Space Division (SD), Directorate of Logistics and Acquisition Support. September 1982.

This publication provides guidance to SD project officers on the acquisition of embedded computer resources, defined in AFR 800-14 as "the totality of computer equipment, computer programs, associated documentation, contractual services, personnel, and supplies". Technical and management activities which lead to a contract award for the development of a computer resource capability are highlighted. Pre-contract award activities are seen as particularly critical for software development, and must lead to a professional business environment wherein the software development requirements are effectively negotiated and made legally binding on the signed contract.

General guidelines for construction of the statement of work (SOW) are presented, with an emphasis on the SOW being well researched and as explicit as possible. Techniques that can be used to enhance software reliability are recommended as specifics to look for in evaluating a vendors qualifications, such as good requirements definition, programming standards, all-branch testing, internal independent verification and validation, and other "equally obvious reliability enhancers". Areas to be considered in a pre-award survey of contractor qualifications are presented in detailed tables.

In computer resources acquisition, the life cycle model is a key concept, but more important is that the SD program office be concerned with the adequacy of how a phase was conducted rather than just adhering to the form of the life cycle phasing. "SPO personnel should be aware of the fact that the life cycle was not invented to burden contractors or to increase the cost of development." The life cycle has evolved because of cost overruns, schedule delays, and sometimes failure.

The remaining chapters address the issues of planning and management control, technical analysis and requirements development, application of standards and specifications, software cost estimation and measurement, configuration management, technical data, reviews and audits, quality assurance, programming languages, and IV&V. "The most important phase of software cost estimating from the Air Force standpoint is the pre-contract activity which will dictate the visibility we will get on the cost/schedule during the life of a contract." The work breakdown structure is one of the most important mechanisms for getting visibility in software cost estimating.

The IV&V concept was originated in the Air Force during the early days of the Ballistic Missile and Space Systems Division, and applied to missile systems software that was connected to activation and control of nuclear weapons or the launching of space vehicles. IV&V is defined to be the application of a variety of techniques often supported with automated software tools to evaluate critical/complex software. These techniques are applied by an experienced contractor who is completely independent.

In the final chapter, critical issues and periods in computer resource management are summarized. The critical periods are primarily in the conceptual and design validation phases. Some key considerations are:

- A detailed study of requirements must be developed by the SPO in concert with the user.

- The investment strategy is to spend as much time and money in this phase as is necessary to do a professional job, as you get the maximum engineering benefit for each dollar spent in this phase.

- A technical and management strategy for the acquisition relating the management aspects of the program to the technical and procurement considerations should be developed.

- A general conceptual phase scenario should include:

    a. Prepare system concept;
    b. Review system concept;
    c. Determine systems approach;
    d. Determine major functions, i/o, processing, estimated storage requirements;
    e. Prepare requirements definition and system specifications;
    f. Prepare master development schedule and cost estimate;
    g. Factor in transition and maintenance concepts.

Probably the most critical activity is to assign the management responsibility and to ensure that the Air Force has the expertise allocated to do the computer resource job.

APPENDIX A

DATA GATHERING GUIDES

I. OVERVIEW OF DATA GATHERING GUIDE FOR HQ AND DEVELOPMENT COMMAND VISITS

OVERVIEW:

A. Background Information

B. Regulations and Standards, Etc.
Regulations and standards, controls and waivers for software T&E and procedures for coordinating multiservice T&E effort.

C. Industry Testing Standards vs. DoD Practices
Contractor selection process, and regulations and standards.

D. New Technology Trends
New technologies related to embedded computer resources and special validation tasks required.

E. Other
Suggested programs and contacts, and general comments.

NOTES:

1. The primary purpose of these interviews was to determine what guidance currently exists and the effectiveness of that guidance, the involvement of the Headquarters and Development Commands with the individual project offices, what the future holds for embedded computer resources, and what programs would be useful for survey purposes.

2. Interviews conducted using this guide had an average duration of one hour.

I. DATA GATHERING GUIDE FOR HQ AND DEVELOPMENT COMMAND VISITS

A. Background Information

1. Name, Organization, Address, Phone Number.

B. Regulations and Standards, Etc.

1. Regulations, etc. that exist with respect to:
   a) Development testing and evaluation.
   b) Operational testing and evaluation.
   c) Acceptance testing.
   d) Test documentation.
   e) Quality assurance.
   f) Independent verification and validation.
   g) Risk assessment.
   h) Other.

2. Controls of software testing and evaluation (reports, etc.).

3. Circumstances under which a program may be exempted from any governing regulations, etc. concerning software.

4. Percentage of the major programs that actually receive waivers.

5. Tailoring of regulations, etc.

6. Strengths of regulations, etc.

7. Weaknesses of regulations, etc.

8. Enhancement efforts with respect to regulations, etc.

9. Procedures for coordinating multiservice software testing and evaluation.

10. Comments on Regulations and Standards, etc.

C. Industry Testing Standards vs. DoD Practices

1. Role in contractor selection process.

2. Guidelines followed with respect to the amount of importance given to a potential contractor's internal policies, etc. and past performance regarding software testing and evaluation when letting a contract.

3. Requirements placed upon contractors with respect to:
   a) Development testing and evaluation.
   b) Operational testing and evaluation.
   c) Test documentation.
   d) Quality assurance.
   e) Independent verification and validation.
   f) Other.

4. Strengths of requirements, etc.

5. Weaknesses of requirements, etc.

6. Enchancement efforts with respect to requirements, etc.

7. Comments on Industry Testing Standards vs. DoD Practices.

D. New Technology Trends

1. New technology trends that relate to embedded computer resources.

E. Other

1. Suggested programs and contracts.

2. General comments.

II. OVERVIEW OF DATA GATHERING GUIDE FOR PROGRAM OFFICE VISITS

OVERVIEW:

A. Background Information
Description of organization and program, current status, applications, development groups, staffing, programming languages, operating systems, existing software, and hardware.

B. Regulations and Standards, Etc.
Regulations and Standards, controls and waivers for software T&E, and procedures for coordinating multiservice T&E effort.

C. Industry Testing Standards vs. DoD Practices
Contractor selection process, and regulations and standards.

D. Pre-Testing Activities
Programming standards and conventions, documentation, requirements analysis, design analysis and reviews, design-to-test procedures, metrics, prototypes, baselining, and code inspections and walkthroughs.

E. Development Testing and Evaluation
Test plans and procedures, testing strategies, testing techniques, and evaluation techniques.

F. Integration Testing
Software and hardware/software integration and testing procedures.

G. Acceptance Testing
Organization, interfaces, acceptance testing procedures, and quantification of requirements and thresholds for acceptance.

H. Testing and Evaluation Tools
Testing and evaluation tools and metrics.

I. Test Documentation Procedures and Regression Testing
Test documentation procedures, maintenance of documentation and test media, tools, documentation of errors, and regression testing techniques.

J. Quality Assurance Program
Scope and organization, QA procedures and standards, QA reviews and evaluations, and QA activities.

K. Independent Verification and Validation
Scope, organization, requirements analysis, design analysis, code analysis, and independent testing.

L. Operational Testing and Evaluation
   Objectives, organization, interfaces, methodology, environment, test plans and procedures, test data analysis, in-plant testing, on-site testing, effectiveness-related testing, full system/casualty mode testing, suitability-related testing, metrics, and regression testing.

M. Risk Assessment
   Risk assessment procedures with respect to success of mission, lethality of failure, and system production, and relationship between risk and testing effort required.

N. New Technology Trends
   New technologies related to the program's embedded computer resources and special validation tasks required.

O. Other
   Lessons learned, suggested contacts, and general comments.

NOTES:

1. The primary purpose of these interviews was to get as much information as possible on all aspects of the software development process as seen by the Program Office. An unexpected result of the interviews was the discovery that the Program Offices, in general, have little detailed knowledge of the activities performed by the contractors or the OT&E agencies.

2. Interviews conducted using this guide had an average duration of four hours.

II. DATA GATHERING GUIDE FOR PROGRAM OFFICE VISITS

A. Background Information

1. Name, organization, address, phone number.

2. Description of organization.

3. Description of program.

4. Current status of program's software.

5. Software applications that exist in program:
   a) Tracking.
   b) Guidance and Control.
   c) Navigation.
   d) Digital Filtering/Image Processing.
   e) Computation.
   f) Communications Systems.
   g) Command and Control/Information Management.
   h) Applications Support.
   i) General Automatic Data Processing.
   j) Built-in-Test Software.
   k) Other.

6. Government furnished software applications. Organization responsible.

7. Contractor furnished software applications. Contractor responsible.

8. Software development lifecycle for program.

9. Programming language(s) used. Percentage of code using each.

10. Operating system used (timesharing vs. batch).

11. Existing operational application software to be used.

12. Hardware used (host and target).

13. Special hardware devices used for testing.

B. Regulations and Standards, Etc.

1. Regulations, etc. being applied to this program with respect to:
   a) Development testing and evaluation.
   b) Operational testing and evaluation.
   c) Acceptance testing.
   d) Test documentation.
   e) Quality assurance.
   f) Independent verification and validation.
   g) Risk assessment.
   h) Other.

2. Controls of software testing and evaluation (reports, etc.).

3. Circumstances under which a program may be exempted from any governing regulations, etc. concerning software.

4. Waivers approved for program's software.

5. Tailored regulations related to software testing and evaluation for program.

6. Strengths of regulations, etc.

7. Weaknesses of regulations, etc.

8. Enhancement efforts with respect to regulations, etc.

9. Procedures for coordinating multiservice software testing and evaluation effort.

10. Comments on Regulations and Standards, etc.

C. Industry Testing Standards vs. DoD Practices

1. Role in the contractor selection process.

2. Guidelines followed with respect to the amount of importance given to a potential contractor's internal policies, etc. and past performance regarding software testing and evaluation when letting a contract.

3. Requirements placed upon contractors with respect to:
   a) Development testing and evaluation.
   b) Operational testing and evaluation.
   c) Acceptance testing.
   d) Test documentation.
   e) Quality assurance.
   f) Independent verification and validation.
   g) Risk assessment.
   h) Other.

4. Strengths of requirements, etc.

5. Weaknesses of requirements, etc.

6. Enchancement efforts with respect to requirements, etc.

7. Comments on Industry Testing Standards vs. DoD Practices.

D. Pre-testing Activities

1. Programming standards and conventions used in program. Enforcement procedures.

2. Procedures to assure software architecture modularity.

3. Procedures to assure that the "Top-down" software development methodology and structured programming are used.

4. Documentation items for software as it proceeds from concept to design to the final operational system.

5. Validation of software requirements/specifications prior to implementation (requirements analysis).

6. Validation of software design prior to implementation (design analysis).

7. Software design reviews conducted for program.

8. Organization responsible for conducting software design reviews.

9. Standards that software design reviews are conducted in accordance with (MIL-STD 1521A?).

10. Software design review participants.

11. Procedures for analysis of units consistency.

12. Design-to-test procedures.

13. Procedures for quantitative assessment of software design maturity and supportability.

14. Metrics used to evaluate software design quality in terms of cohesiveness, coupling, scope of effect/control, modularity, etc.

15. Software prototypes built to refine system requirements/specifications and design prior to implementation.

16. Software design baselining procedures.

17. Verification that code conforms to the original software design (program analysis).

18. Code inspections and walkthroughs conducted.

19. Error checklists used for inspections.

20. Other pre-testing activities.

21. Differences in pre-testing activities between application areas.

22. Major likes and dislikes of pre-testing activities.

23. Reasons for use/non-use of pre-testing activities.

24. Strengths of pre-testing activities.

25. Weaknesses of pre-testing activities.

26. Enhancement efforts with respect to pre-testing activities.

27. Comments on Pre-testing Activities.

E. Development Testing and Evaluation

1. Software testing and evaluation process for program, covering the entire software development life cycle.

2. Software tests performed. Levels of testing employed.

3. Time of initial preparations for software testing.

4. Software module test plans and procedures development process.

5. Testing strategies used:
   a) Black-box Methodology.
      Input space partitioning.
      Cause-effect graphing.
      Random testing.
      Automated generation of test data.
   b) White-box Methodology.
      Logic coverage testing.
      - statement coverage.
      - branch coverage.
      - condition coverage.
      Domain testing.
   c) Top-down.
   d) Bottom-up.
   e) Thread testing.
   f) Other.

6. Lines of code/test case ratio (planned and actual).

7. Procedures to ensure that test data is representative of the total range of data and operational conditions that the software might encounter.

8. Test execution procedures - scenario fashion simulating "real world" situations vs. testing all inputs, displays, processing, etc. in generic groups.

9. Types of calibration bugs used to test the test data.

10. Differences in testing strategies between application areas.

11. Major likes and dislikes of these testing strategies.

12. Reasons for use/non-use of these testing strategies.

13. Testing techniques used:
    a) Symbolic testing.
    b) Program instrumentation.
    c) Mutation.
    d) Input space partitioning.
    e) Functional program testing.
    f) Algebraic program testing.
    g) Random testing.
    h) Grammar-based testing.
    i) Data-flow guided testing.
    j) Other.

14. Differences in use of testing techniques between application areas.

15. Major likes and dislikes of testing techniques.

16. Reasons for use/non-use of Testing Techniques.

17. Software module interface testing procedures.

18. Validation procedures for critical software computational and decision algorithms and their timing assumptions.

19. Formal proofs of correctness attempts for program's software.

20. Measurements of software maturity (versus design maturity) during development.

21. Procedures for quantitatively demonstrating the completion of software development.

22. Evaluation techniques used.

23. Software verification and validation procedures. Organization responsible.

24. Organization responsible for preparation of test data for software validation.

25. Mechanism to make an independent assessment of the software.

26. Differences in the use of evaluation techniques between application areas.

27. Major likes and dislikes of these evaluation techniques.

28. Strengths of development testing and evaluation process.

29. Weaknesses of development testing and evaluation process.

30. Enhancement efforts with respect to development testing and evaluation process.

31. Comments on Development Testing and Evaluation.

F. Integration Testing

1. Software integration and testing procedures.

2. Organization responsible for software integration when there is a mix of government furnished software and contractor furnished software.

3. Procedures for quantitatively demonstrating the completion of software integration and testing.

4. Major likes and dislikes of software integration testing methodology.

5. Software/hardware integration and testing procedures.

6. Procedures for quantitatively demonstrating the completion of software/hardware integration and testing.

7. Major likes and dislikes of software/hardware integration testing methodology.

8. Strengths of integration testing.

9. Weaknesses of integration testing.

10. Enhancement efforts with respect to integration testing.

11. Comments on Integration Testing.

G. Acceptance Testing

1. Software acceptance testing process for program.

2. Organization responsible for the acquisition of the program's software.

3. Organization responsible for conducting the software acceptance testing for program. Interfacing process.

4. Process used to develop the software acceptance test plans and procedures.

5. Procedures to use acceptance testing to establish the proper execution of each software function.

6. Procedures to use acceptance testing to demonstrate that the integrated software operates correctly in the user environment.

7. Procedures to use acceptance testing to demonstrate the compliance of the integrated software with general performance requirements.

8. Procedures for the quantification of requirements for software and threshold values for acceptance.

9. Differences in the software acceptance testing process between application areas.

10. Organization responsible for the final decision to accept/reject software. Information decision is based on.

11. Strengths of the software acceptance testing process.

12. Weaknesses of the software acceptance testing process.

13. Enchancement efforts with respect to the software acceptance testing process.

14. Comments on Acceptance Testing.

H. Testing and Evaluation Tools

1. Testing and evaluation tools used:
   a) Static analyzers.
   b) Symbolic evaluators.
   c) Test data generators.
   d) Program instrumenters.
   e) Mutation testing tools.
   f) Automatic test drivers.
   g) Comparators.
   h) Others.

2. Metrics used to evaluate the software.

3. Differences in use of testing and evaluation tools between application areas.

4. Major likes and dislikes of testing and evaluation tools.

5. Reasons for use/non-use of testing and evaluation tools.

6. Testing and evaluation tools available but not used. Justification for non-use.

7. Strengths of testing and evaluation tools.

8. Weaknesses of testing and evaluation tools.

9. Enhancement efforts with respect to testing and evaluation tools.

10. Comments on Testing and Evaluation Tools.

I. Test Documentation Procedures and Regression Testing

1. Software test documentation procedures for program.

2. Procedures to maintain test-related documentation and media to allow repeatability of tests.

3. Procedures used to ensure that changes in the requirements and/or specifications trigger changes in the test documentation.

4. Tools used to maintain and control test case library.

5. Procedures to define, collect, analyze, and report software error data.

6. Regression testing procedures for software.

7. Procedures to determine the amount of regression testing to be performed given an arbitrary change to the software.

8. Strengths of the software test documentation and regression testing procedures.

9. Weaknesses of the software test documentation and regression testing procedures.

10. Enhancement efforts with respect to the software test documentation and regression testing procedures.

11. Comments on Test Documentation Procedures and Regression Testing.

J. Quality Assurance Program

1. Software quality assurance program.

2. Scope and organization of software quality assurance program (separate function?).

3. Software development standards and procedures required by quality assurance plan.

4. Process used by quality assurance to ensure that standards and procedures are being followed.

5. Quality assurance organization reviews and evaluations of software documentation:
   a) Requirements specifications.
   b) Design specifications.
   c) Test plans and procedures.
   d) User manuals.
   e) Implementer's Guide.
   f) Other.

6. Quality assurance activities:
   a) Software design reviews and audits.
   b) Code walk-throughs.
   c) Software acceptance testing witnessing.
   d) Final software configuration audit prior to its installation in the operational environment.
   e) Other.

7. Strengths of the software quality assurance program.

8. Weaknesses of software quality assurance program.

9. Enhancement efforts with respect to the software quality assurance program.

10. Comments on Quality Assurance Program.

K. Independent Verification and Validation

1. Scope of the independent verification and validation effort of the software for this program.

2. Organization responsible for the independent verification and validation of software for program.

3. Organization to whom the independent verification and validation organization reports.

4. Involvement and responsibilities with respect to the independent verification and validation of software for program.

5. Time of initial involvement of the independent verification and validation organization with program's software.

6. Independent verification and validation activities:
   a) Analysis of software requirements for completeness, correctness, consistency, traceability, and testability.
   b) Analysis of software design for correctness and satisfaction of requirements.
   c) Analysis of code to verify correct implementation of the design.
   d) Independent test of software (Using nominal scenarios? Using worst-case scenarios?).
   e) Other.

7. Procedures used by the independent verification and validation organization to ensure that the software does not fail by unintentionally performing an undesirable function.

8. Strengths of independent verification and validation.

9. Weaknesses of independent verification and validation.

10. Enhancement efforts with respect to independent verification and validation.

11. Comments on Independent Verification and Validation.

L. Operational Testing and Evaluation

1. Objectives of operational testing and evaluation of software.

2. Organization responsible for conducting the operational testing and evaluation of software for program.

3. Process used to interface with the software operational testing and evaluation organization.

4. Involvement and responsibilities with respect to the operational testing and evaluation of software for program.

5. Extent of operational testing and evaluation organization's participation in software development, covering the entire lifecycle:
   a) Requirements analysis.
   b) Design analysis.
   c) Program analysis.
   d) Development testing and evaluation.
   e) Other.

6. Software operational testing and evaluation methodology.

7. Operational testing and evaluation environment:
   a) Special hardware.
   b) Monitoring devices.
   c) Patches allowed during testing.
   d) Number of prototype devices dedicated to OT&E.
   e) Other.

8. Time of initial preparations for software operational testing.

9. Software operational test plans and procedures development process.

10. Procedures used by the operational testing and evaluation organization to ensure that the test scenarios are representative of the total range of data and operational conditions that the software might encounter.

11. In-plant testing.

12. On-site testing.

13. Procedures to ensure that the software meets its stated operational requirements.

14. Effectiveness-related testing procedures for software with respect to:
    a) Performance.
    b) Machine-machine interface.
    c) Operator-machine interface.
    d) Other.

15. Validation of critical software computational and decision algorithms and their timing assumptions during operational testing and evaluation.

16. Full system/casualty mode testing.

17. Procedures to use testing to clearly identify deficiencies as software or hardware related.

18. Suitability-related testing procedures for software with respect to:
    a) Maintainability.
        - source code.
        - documentation.
        - computer support resources.
    b) Usability.
    c) Other.

19. Metrics used by the operational testing and evaluation organization to evaluate the software.

20. Regression testing procedures used by the operational testing and evaluation organization.

21. Procedure used to determine the necessary amount of regression testing given a set of changes to the software during operational testing and evaluation.

22. Procedures to quantitatively demonstrate the completion of software operational testing and evaluation.

23. Differences in the operational testing and evaluation methodology for software between application areas.

24. Strengths of the operational testing and evaluation process for software.

25. Weaknesses of the operational testing and evaluation process for software.

26. Enhancement efforts with respect to the operational testing and evaluation process for software.

27. Comments on Operational Testing and Evaluation.

M. Risk Assessment

1. Software risk assessment procedures for program.

2. Software risk assessment procedures with respect to:
   a) Success of the mission.
   b) Lethality of failure.
   c) System production.
   d) Other.

3. Relationship between risk posed by various casualty modes of software failure and testing effort required in the development-procurement-maintenance phases of the software lifecycle.

4. Strengths of software risk assessment procedures.

5. Weaknesses of software risk assessment procedures.

6. Enhancement efforts with respect to the software risk assessment procedures.

7. Comments on Risk Assessment.

N. New Technology Trends

1. New technologies to be developed or utilized for program's embedded computer resources.

2. Special tasks to be performed to validate new technologies.

O. Other

1. "Lessons learned" from program.

2. Suggested contacts (contractors, IV&V contractors, OT&E personnel, etc.).

3. General comments.

III. OVERVIEW OF DATA GATHERING GUIDE FOR SOFTWARE DEVELOPMENT SHOP VISITS

OVERVIEW:

A. Background Information
Description of organization and program, current status, applications, staffing, programming languages, operating systems, existing software, and hardware.

B. Industry Testing Standards vs. DoD Practices
Regulations and standards, and controls.

C. Pre-Testing Activities
Programming standards and conventions, documentation, requirements analysis, design analysis and reviews, design-to-test procedures, metrics, prototypes, baselining, and code inspections and walkthroughs.

D. Development Testing and Evaluation
Test plans and procedures, testing strategies, testing techniques, and evaluation techniques.

E. Integration Testing
Software and software/hardware integration and testing procedures.

F. Acceptance Testing
Organization, interfaces, acceptance testing procedures, and quantification of requirements and thresholds for acceptance.

G. Testing and Evaluation Tools
Testing and evaluation tools and metrics.

H. Test Documentation Procedures and Regression Testing
Test documentation procedures, maintenance of documentation and test media, tools, documentation of errors, and regression testing.

I. Quality Assurance Program
Scope and organization, QA procedures and standards, QA reviews and evaluations, and QA activities.

J. Independent Verification and Validation
Scope, organization, requirements analysis, design analysis, code analysis, and independent testing.

K. Operational Testing and Evaluation
Organization, interfaces, involvement, and responsibilities.

L. Risk Assessment

Risk assessment procedures with respect to success of mission, lethality of failure, and system production, and relationship between risk and testing effort required.

M. New Technology Trends

New technologies related to program's embedded computer resources and special validation tasks required.

N. Other

Lessons learned, suggested contacts, and general comments.

Notes:

1. The primary purpose of these interviews was to get as much information as possible on all aspects of the software development process.
2. Interviews conducted using this guide had an average duration of four hours.

III. DATA GATHERING GUIDE FOR SOFTWARE DEVELOPMENT SHOP VISITS

A. Background Information

1. Name, organization, address, and phone number.

2. Description of organization.

3. Description of program.

4. Current status of software.

5. Software applications responsible for:
   a) Tracking.
   b) Guidance and Control.
   c) Navigation.
   d) Digital Filtering/Image Processing.
   e) Computation.
   f) Communications Systems.
   g) Command and Control/Information Management.
   h) Applications Support.
   i) General Automatic Data Processing.
   j) Built-in-Test Software.
   k) Other.

6. Software development lifecycle for program.

7. Programming language(s) used. Percentage of code using each.

8. Operating system used (timesharing vs. batch).

9. Existing operational application software to be used.

10. Hardware used (host and target).

11. Special hardware devices used for testing.

B. Industry Testing Standards vs. DoD Practices

1. Requirements placed upon contractors with respect to:
   a) Development testing and evaluation.
   b) Operational testing and evaluation.
   c) Acceptance testing.
   d) Test documentation.
   e) Quality assurance.
   f) Independent verification and validation.
   g) Risk assessment.
   h) Other.

2. Controls of software testing and evaluation (reports, etc.).

3. Strengths of requirements, etc.

4. Weaknesses of requirements, etc.

5. Enhancement efforts with respect to requirements, etc.

6. Comments on Industry Testing Standards vs. DoD Practices.

C. Pre-Testing Activities

1. Programming standards and conventions used in program. Enforcement procedures.

2. Procedures to assure software architecture modularity.

3. Procedures to assure that the "Top-down" software development methodology and structured programming are used.

4. Documentation items for software as it proceeds from concept to design to the final operational system.

5. Validation of software requirements/specifications prior to implementation (requirements analysis).

6. Validation of software design prior to implementation (design analysis).

7. Software design reviews conducted for program.

8. Organization responsibile for conducting software design reviews.

9. Standards that software design reviews are conducted in accordance with (MIL-STD-1521A?).

10. Software design review participants.

11. Procedures for an analysis of units consistency.

12. Design-to-Test procedures.

13. Procedures for quantitative assessment of software design maturity and supportability.

14. Metrics used to evaluate software design quality in terms of cohesiveness, coupling, scope of effect/control, parsimony, modularity, etc.

15. Software prototypes built to refine system requirements/ specifications and design prior to implementation.

16. Software design baselining procedures.

17. Verification that code conforms to the original software design (Program analysis).

18. Code inspections and walkthroughs conducted.

19. Error checklists used for inspections.

20. Other pre-testing activities.

21. Differences in pre-testing activities between application areas.

22. Major likes and dislikes of pre-testing activities.

23. Reasons for use/non-use of pre-testing activities.

24. Strengths of pre-testing activities.

25. Weaknesses of pre-testing activities.

26. Enhancement efforts with respect to pre-testing activities.

27. Comments on Pre-testing Activities.

D. Development Testing and Evaluation

1. Software testing and evaluation process for program, covering the entire software development life cycle.

2. Software tests performed. Levels of testing employed.

3. Time of initial preparations for software testing.

4. Software module test plans and procedures development process.

5. Testing strategies used:
   a) Black-box Methodology.
      Input space partitioning.
      Cause-effect graphing.
      Error guessing.
      Automated generation of test cases.
   b) White-box Methodology.
      Logic coverage testing.
      - Statement coverage.
      - Branch coverage.
      - Condition coverage.
      Domain testing.

c) Top-down.
d) Bottom-up.
e) Thread testing.
f) Other.

6. Lines of code/test case ratio (planned and actual).

7. Procedures to ensure that test data is representative of the total range of data and operational conditions that the software might encounter.

8. Test execution procedures - scenario fashion simulating "real world" situations vs. testing all inputs, displays, processing, etc. in generic groups.

9. Types of calibration bugs used to test the test data.

10. Differences in testing strategies between application areas.

11. Major likes and dislikes of these testing strategies.

12. Reasons for use/non-use of these testing strategies.

13. Testing Techniques used:
a) Symbolic testing.
b) Program instrumentation.
c) Mutation.
d) Input space partitioning.
e) Functional program testing.
f) Algebraic program testing.
g) Random testing.
h) Grammar-based testing.
i) Data-flow guided testing.
j) Other.

14. Differences in use of testing techniques between application areas.

15. Major likes and dislikes of testing techniques.

16. Reasons for use/non-use of testing techniques.

17. Software module interface testing procedures.

18. Validation procedures for critical software computational and decision algorithms and their timing assumptions.

19. Formal proofs of correctness attempts for program's software.

20. Measurements of software maturity (versus design maturity) during development.

21. Procedures for quantitatively demonstrating the completion of software development.

22. Evaluation techniques used.

23. Software verification and validation procedures. Organization responsible.

24. Organization responsible for preparation of test data for software validation.

25. Mechanism to make an independent assessment of the software.

26. Differences in use of evaluation techniques between application areas.

27. Major likes and dislikes of these evaluation techniques.

28. Strengths of development testing and evaluation process.

29. Weaknesses of development testing and evaluation process.

30. Enhancement efforts with respect to development testing and evaluation process.

31. Comments on Development Testing and Evaluation.

E. Integration Testing

1. Software integration and testing procedures.

2. Organization responsible for software integration when there is a mix of government furnished software and contractor furnished software.

3. Procedures for quantitatively demonstrating the completion of software integration and testing.

4. Major likes and dislikes of software integration testing methodology.

5. Software/hardware integration and testing procedures.

6. Procedures for quantitatively demonstrating the completion of software/hardware integration and testing.

7. Major likes and dislikes of software/hardware integration testing methodology.

8. Strengths of integration testing.

9. Weaknesses of integration testing.

10. Enhancement efforts with respect to integration testing.

11. Comments on Integration Testing.

F. Acceptance Testing

1. Software acceptance testing process for program.

2. Organization responsible for the acquisition of the program's software.

3. Organization responsible for conducting the software acceptance testing. Interfacing process.

4. Involvement and responsibilities with respect to the acceptance testing of software.

5. Process used to develop the software acceptance test plans and procedures.

6. Procedures to use acceptance testing to establish the proper execution of each software function.

7. Procedures to use acceptance testing to demonstrate that the integrated software operates correctly in the user environment.

8. Procedures to use acceptance testing to demonstrate the compliance of the integrated software with general performance requirements.

9. Procedures for the quantification of requirements for software and threshold values for acceptance.

10. Differences in the software acceptance testing process between application areas.

11. Organization responsible for the final decision to accept/reject software. Information decision is based on.

12. Strengths of the software acceptance testing process.

13. Weaknesses of the software acceptance testing process.

14. Enhancement efforts with respect to the software acceptance testing process.

15. Comments on Acceptance Testing.

G. Testing and Evaluation Tools

1. Testing and evaluation tools used:
   a) Static analyzers.
   b) Symbolic evaluators.
   c) Test data generators.
   d) Program instrumenters.
   e) Mutation testing tools.
   f) Automatic test drivers.
   g) Comparators.
   h) Others.

2. Metrics used to evaluate the software.

3. Differences in use of testing and evaluation tools between application areas.

4. Major likes and dislikes of testing and evaluation tools.

5. Reasons for use/non-use of testing and evaluation tools.

6. Testing and evaluation tools available but not used. Justification for non-use.

7. Strengths of testing and evaluation tools.

8. Weaknesses of testing and evaluation tools.

9. Enhancement efforts with respect to testing and evaluation tools.

10. Comments on Testing and Evaluation Tools.

H. Test Documentation Procedures and Regression Testing

1. Software test documentation procedures.

2. Procedures to maintain test-related documentation and media to allow repeatability of tests.

3. Procedures used to ensure that changes in the requirements and/or specifications trigger changes in the test documentation.

4. Tools used to maintain and control test case library.

5. Procedures to define, collect, analyze, and report software error data.

6. Regression testing procedures for software.

7. Procedures to determine the amount of regression testing to be performed given an arbitrary change to the software.

8. Strengths of the software test documentation and regression testing procedures.

9. Weaknesses of the software test documentation and regression testing procedures.

10. Enhancement efforts with respect to the software test documentation and regression testing procedures.

11. Comments on Test Documentation Procedures and Regression Testing.

I. Quality Assurance Program

1. Software quality assurance program.

2. Scope and organization of software quality assurance program (separate function?).

3. Software development procedures and standards required by quality assurance plan.

4. Process used by quality assurance to ensure that standards and procedures are being followed.

5. Quality assurance organization reviews and evaluations of software documentation:
   a) Requirements specifications.
   b) Design specifications.
   c) Test plans and procedures.
   d) User manuals.
   e) Implementer's guide.
   f) Other.

6. Quality assurance activities:
   a. Software design reviews and audits.
   b. Code walk-throughs.
   c. Software acceptance testing witnessing.
   d. Final software configuration audit prior to its installation in the operational environment.
   e. Other.

7. Strengths of the software quality assurance program.

8. Weaknesses of software quality assurance program.

9. Enhancement efforts with respect to the software quality assurance program.

10. Comments on Quality Assurance Program.

J. Independent Verification and Validation

1. Scope of the independent verification and validation effort of the software for this program.

2. Organization responsible for the independent verification and validation of software for program.

3. Organization to whom the independent verification and validation organization reports.

4. Involvement and responsibilities with respect to the independent verification and validation of software for program.

5. Time of initial involvement of the independent verification and validation organization with program's software.

6. Independent verification and validation activities:
   a) Analysis of software requirements for completeness, correctness, consistency, traceability, and testability.
   b) Analysis of software design for correctness and satisfaction of requirements.
   c) Analysis of code to verify correct implementation of the design.
   d) Independent test of software (Using nominal scenarios? Using worst-case scenarios?).
   e. Other.

7. Procedures used by the independent verification and validation organization to ensure that the software does not fail by unintentionally performing an undesirable function.

8. Strengths of independent verification and validation.

9. Weaknesses of independent verification and validation.

10. Enhancement efforts with respect to independent verification and validation.

11. Comments on Independent Verification and Validation.

K. Operational Testing and Evaluation

1. Objectives of operational testing and evaluation of software.

2. Organization responsible for conducting the operational testing and evaluation of software.

3. Process used to interface with the software operational testing and evaluation organization.

4. Involvement and responsibilities with respect to the operational testing and evaluation of software.

5. Time of initial involvement of the operational testing and evaluation organization with software.

6. Extent of operational testing and evaluation organization's participation in software development, covering the entire lifecycle:
   a) Requirements analysis.
   b) Design analysis.
   c) Program analysis.
   d) Development testing and evaluation.
   e) Other.

7. Differences in the operational testing and evaluation methodology for software between application areas.

8. Strengths of the operational testing and evaluation process for software.

9. Weaknesses of the operational testing and evaluation process for software.

10. Enhancement efforts with respect to the operational testing and evaluation process for software.

11. Comments on Operational Testing and Evaluation.

L. Risk Assessment

1. Software risk assessment procedures for program.

2. Software risk assessment procedures with respect to:
   a) Success of the mission.
   b) Lethality of failure.
   c) System production.
   d) Other.

3. Relationship between risk posed by various casualty modes of software failure and testing effort required in the development-procurement-maintenance phases of the software lifecycle.

4. Strengths of software risk assessment procedures.

5. Weaknesses of software risk assessment procedures.

6. Enhancement efforts with respect to the software risk assessment procedures.

7. Comments on Risk Assessment.

M. New Technology Trends

1. New technologies to be developed or utilized for program's embedded computer resources.

2. Special tasks to be performed to validate new technologies.

N. Other

1. "Lessons learned" from program.

2. Suggested contacts (IV&V contractors, OT&E personnel, etc.).

3. General comments.

IV. OVERVIEW OF DATA GATHERING GUIDE FOR IV&V ORGANIZATION VISITS

OVERVIEW:

A. Background Information
Description of organization and program, current status, applications, development groups, staffing, programming languages, operating systems, existing software, and hardware.

B. Industry IV&V Standards vs. DoD Practices
Requirements for IV&V and controls of IV&V.

C. IV&V Pre-Testing Activities
Requirements analysis, design analysis and reviews, metrics, and code inspections and walkthroughs.

D. IV&V Development Testing
Test plans and procedures, testing strategies, testing techniques, and evaluation techniques.

E. IV&V Integration Testing
Software and software/hardware integration and testing procedures.

F. IV&V Involvement in Acceptance Testing
Organization, interfaces, acceptance testing procedures, and quantification of requirements and thresholds for acceptance.

G. IV&V Testing and Evaluation Tools
Testing and evaluation tools and metrics.

H. IV&V Test Documentation Procedures and Regression Testing
Test documentation procedures, maintenance of documentation and test media, tools, and documentation of errors, and regression testing.

I. IV&V Involvement in Operational Testing and Evaluation
Organization, interfaces, involvement, and responsibilities.

J. New Technology Trends
New technologies related to program's embedded computer resources and special validation tasks required.

K. Other
Lessons learned, suggested contacts, and general comments.

NOTES:

1. These interviews had a primary purpose of getting all available information on all aspects of IV&V involvement in the software development process.

2. Interviews conducted using this guide had an average duration of two hours.

IV. DATA GATHERING GUIDE FOR IV&V ORGANIZATION VISITS

A. Background Information

1. Name, organization, address, and phone number.

2. Description of organization.

3. Description of program.

4. Current status of program's software.

5. Scope of the independent verification and validation effort for the software for this program.

6. Organization to whom the independent verification and validation organization reports.

7. Time of initial involvement of the independent verification and validation organization with the program's software.

8. Software applications that exist in program:
   a) Tracking.
   b) Guidance and Control.
   c) Navigation.
   d) Digital Filtering/Image Processing.
   e) Computation.
   f) Communications Systems.
   g) Command and Control/Information Management.
   h) Applications Support.
   i) General Automatic Data Processing.
   j) Built-in-Test Software.
   k) Other.

9. Government furnished software applications. Organization responsible.

10. Contractor furnished software applications. Contractor responsible.

11. Software development lifecycle for program.

12. Programming language(s) used. Percentage of code using each.

13. Operating system used (timesharing vs. batch).

14. Existing operational application software to be used.

15. Hardware used (host and target).

16. Special hardware devices used for testing.

B. Industry IV&V Standards vs. DoD Practices

1. Requirements, etc. related to IV&V of software.

2. Controls of software IV&V (reports, etc.).

3. Strengths of requirements, etc.

4. Weaknesses of requirements, etc.

5. Enhancement efforts with respect to requirements, etc.

6. Comments on Industry IV&V Standards vs. DoD Practices.

C. IV&V Pre-Testing Activities

1. Documentation items for software as it proceeds from concept to design to the final operational system.

2. Analysis of software requirements for completeness, correctness, consistency, traceability, and testability.

3. Analysis of software design for correctness and satisfaction of requirements.

4. Software design reviews conducted for program.

5. Organization responsibile for conducting software design reviews.

6. Standards that software design reviews are conducted in accordance with (MIL-STD-1521A?).

7. Software design review participants.

8. Procedures for analysis of units consistency.

9. Procedures for quantitative assessment of software design maturity and supportability.

10. Metrics used to evaluate software design quality in terms of cohesiveness, coupling, scope of effect/control, modularity, etc.

11. Analysis of code to verify correct implementation of the design.

12. Code inspections and walkthroughs conducted.

13. Error checklists used for inspections.

14. Other IV&V pre-testing activities.

15. Differences in pre-testing activities between application areas.

16. Major likes and dislikes of IV&V pre-testing activities.

17. Reasons for use/non-use of IV&V pre-testing activities.

18. Strengths of IV&V pre-testing activities.

19. Weaknesses of IV&V pre-testing activities.

20. Enhancement efforts with respect to IV&V pre-testing activities.

21. Comments on IV&V Pre-Testing Activities.

D. IV&V Development Testing

1. IV&V independent test of software (Using nominal scenarios? Using worst-case scenarios?).

2. Software tests performed. Levels of testing employed.

3. Time of initial preparations for software testing.

4. Software module test plans and procedures development process.

5. Testing strategies used:
   a) Black-box Methodology.
      Input space partitioning.
      Cause-effect graphing.
      Error guessing.
      Random testing.
      Automated generation of test cases.
   b) White-box Methodology.
      Logic coverage testing.
         - Statement coverage.
         - Branch coverage.
         - Condition coverage.
      Domain testing.
   c) Top-down.
   d) Bottom-up.
   e) Threat testing.
   f) Other.

6. Lines of code/test case ratio (planned and actual).

7. Procedures to ensure that test data is representative of the total range of data and operational conditions that the software might encounter.

8. Test execution procedures - scenario fashion simulating "real world" situations vs. testing all inputs, displays, processing, etc. in generic groups.

9. Procedures used to ensure that the software does not fail by unintentionally performing an undesirable function.

10. Types of calibration bugs used to test the test data.

11. Differences in testing strategies between application areas.

12. Major likes and dislikes of these testing strategies.

13. Reasons for use/non-use of these testing strategies.

14. Testing techniques used:
    a) Symbolic testing.
    b) Program instrumentation.
    c) Mutation.
    d) Input space partitioning.
    e) Functional program testing.
    f) Algebraic program testing.
    g) Random testing.
    h) Grammar-based testing.
    i) Data-flow guided testing.
    j) Other.

15. Differences in use of testing techniques between application areas.

16. Major likes and dislikes of testing techniques.

17. Reasons for use/non-use of testing techniques.

18. Software module interface testing procedures.

19. Validation procedures for critical software computational and decision algorithms and their timing assumptions.

20. Formal proofs of correctness attempts for program's software.

21. Measurements of software maturity (versus design maturity) during development.

22. Procedures for quantitatively demonstrating the completion of software development.

23. Evaluation techniques used.

24. Differences in the use of evaluation techniques between application areas.

25. Major likes and dislikes of these evaluation techniques.

26. Strengths of IV&V development testing and evaluation process.

27. Weaknesses of IV&V development testing and evaluation process.

28. Enhancement efforts with respect to IV&V development testing and evaluation process.

29. Comments on IV&V Development Testing and Evaluation.

E. IV&V Integration Testing

1. IV&V software integration and testing procedures.

2. Procedures for quantitatively demonstrating the completion of IV&V software integration and testing.

3. Major likes and dislikes of IV&V software integration testing methodology.

4. IV&V software/hardware integration and testing procedures.

5. Procedures for quantitatively demonstrating the completion of IV&V software/hardware integration and testing.

6. Major likes and dislikes of IV&V software/hardware integration testing methodology.

7. Strengths of IV&V integration testing.

8. Weaknesses of IV&V integration testing.

9. Enhancement efforts with respect to IV&V integration testing.

10. Comments on IV&V Integration Testing.

F. IV&V Involvement in Acceptance Testing

1. Software acceptance testing process for program.

2. Organization responsible for the acquisition of the program's software.

3. Organization responsible for conducting the software acceptance testing for program. Interfacing process.

4. Involvement and responsibilities with respect to the acceptance testing of software.

5. Process used to develop the software acceptance test plans and procedures.

6. Procedures to use acceptance testing to establish the proper execution of each software function.

7. Procedures to use acceptance testing to demonstrate that the integrated software operates correctly in the user environment.

8. Procedures to use acceptance to demonstrate the compliance of the integrated software with general performance requirements.

9. Procedures for the quantification of requirements for software and threshold values for acceptance.

10. Differences in the software acceptance testing process between application areas.

11. Organization responsible for the final decision to accept/reject software. Information decision is based on.

12. Strengths of the IV&V involvement in the software acceptance testing process.

13. Weaknesses of the IV&V involvement in the software acceptance testing process.

14. Enhancement efforts with respect to the IV&V involvement in the software acceptance testing process.

15. Comments on IV&V Involvement in Acceptance Testing.

G. IV&V Testing and Evaluation Tools

1. Testing and evaluation tools used:
   a) Static analyzers.
   b) Symbolic evaluators.
   c) Test data generators.
   d) Program instrumenters.
   e) Mutation testing tools.
   f) Automatic test drivers.
   g) Comparators.
   h) Other.

2. Metrics used to evaluate the software.

3. Differences in use of testing and evaluation tools between application areas.

4. Major likes and dislikes of testing and evaluation tools.

5. Reasons for use/non-use of testing and evaluation tools.

6. Testing and evaluation tools available but not used. Justification for non-use.

7. Strengths of testing and evaluation tools.

8. Weaknesses of testing and evaluation tools.

9. Enhancement efforts with respect to testing and evaluation tools.

10. Comments on IV&V Testing and Evaluation Tools.

H. IV&V Test Documentation Procedures and Regression Testing

1. IV&V software test documentation procedures.

2. Procedures to maintain test-related documentation and media to allow repeatability of tests.

3. Procedures used to ensure that changes in the requirements and/or specifications trigger changes in the test documentation.

4. Tools used to maintain and control test case library.

5. Procedures to define, collect, analyze, and report software error data.

6. IV&V regression testing procedures for software.

7. Procedures to determine the amount of regression testing to be performed given an arbitrary change to the software.

8. Strengths of IV&V software test documentation and regression testing procedures.

9. Weaknesses of IV&V software test documentation and regression testing procedures.

10. Enhancement efforts with respect to IV&V software test documentation and regression testing procedures.

11. Comments on IV&V Test Documentation Procedures and Regression Testing.

I. IV&V Involvement in Operational Testing and Evaluation

1. Objectives of operational testing and evaluation of software.

2. Organization responsible for conducting the operational testing and evaluation of software for program.

3. Process used to interface with the software operational testing and evaluation organization.

4. Involvement and responsibilities with respect to the operational testing and evaluation of software for program.

5. Differences in the IV&V involvement in operational testing and evaluation for software between application areas.

6. Strengths of the IV&V involvement in the operational testing and evaluation process for software.

7. Weaknesses of the IV&V involvement in the operational testing and evaluation process for software.

8. Enhancement efforts with respect to the IV&V involvement in the operational testing and evaluation process for software.

9. Comments on IV&V Involvement in Operational Testing and Evaluation.

J. New Technology Trends

1. New technologies to be developed or utilized for program's embedded computer resources.

2. Special tasks to be performed to validate new technologies.

K. Other

1. "Lessons learned" from program.

2. Suggested contacts.

3. General comments.

APPENDIX B

BIBLIOGRAPHY

DoD DIRECTIVES AND INSTRUCTIONS

DoD Directives and Instructions are issued to DoD Components. They provide guidance and uniformity of thrust, which the separate services may tailor, supplement or amplify for their own specific applications, including more detail when appropriate. Those listed below were chosen based on their applicability to the topics addressed by STEP.

DoDD 5000.1: Major Systems Acquisitions. 19 March 1980.

DoDI 5000.2: Major System Acquisition Procedures. 19 March 1980.

DoDD 5000.3: Test and Evaluation. 26 December 1979.

DoDD 5000.29: Management of Computer Resources in Major Defense Systems. 26 April 1976.

DoDI 7920.2: Major Automated Information Systems Approval Process. 20 October 1978.

MILITARY STANDARDS

Military Standards and Data Item Descriptions may be applied on contracts or used as guidelines by Project Managers. Those listed below were chosen based on their applicability to the topics addressed by STEP.

MIL-STD-483 (USAF): Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs. 1 June 1971.

MIL-STD-490: Specification Practices. 1 February 1969.

MIL-STD-1679 (NAVY): Weapon System Software Development. 1 December 1978.

MIL-S-52779A: Software Quality Assurance Program Requirements. 1 August 1979.

DI-S-30567A: Computer Program Development Plan. 2 February 1978. Air Force DID.

DI-T-3703A: Computer Program Configuration Item Test Plans/Procedures. 18 May 1977. Air Force DID.

DI-T-3717A: Computer Program Configuration Item Development T&E Test Report. 18 May 1977. Air Force DID.

MIL-HDBK-255(AS): Nuclear Weapons Systems, Safety, Design and Evaluation Criteria For. 5 May 1978.

MIL-STD-SDS: Defense System Software Development (Working Papers). 15 April 1982.

A Comparison of MIL-STD-SDS and MIL-STD-1679 (Navy). 15 April 1982.

Proposed Revisions, MIL-STD-483 (USAF). 15 April 1982.

MIL-STD-490: Proposed Revisions. 15 April 1982.

MIL-STD-1521 (USAF): Technical Reviews and Audits for Systems, Equipments, and Computer Programs. Proposed Revisions. 15 April 1982.

AIR FORCE REGULATIONS

These Air Force Regulations address areas of concern to STEP.

AFR 80-14: Research and Development, Test and Evaluation. 12 September 1980.

AFR 800-14: Acquisition Management, Management of Computer Resources in Systems (Volumes I & II). 12 September 1975.

AFR 122-9: Nuclear Safety Cross-Check Analysis. 1 July 1974.

AFR 122-10: Nuclear Weapon Systems Safety Design and Evaluation Criteria. 27 November 1978.

ARMY REGULATIONS

These Army Regulations address areas of concern to STEP. DARCOM Regulations are used by the Army's Materiel Development and Readiness Command to supplement the Army Regulations.

AR 70-1: Army Research, Development and Acquisition. 15 February 1977.

AR 70-10: Research and Development Test and Evaluation During Development and Acquisition of Materiel. 29 August 1975.

AR-702-9: Product Assurance - Production Acceptance Testing and Evaluation. No Date.

AR 1000-1: Utilization, Basic Policies for Systems Acquisition. 1 June 1981.

DARCOM 70-16: Management of Computer Resources in Battlefield Automated Systems. 16 July 1979.

DARCOM 702-6: Quality Assurance and Product Quality Management. 13 March 1979.

DARCOM 702-10: Quality Assurance Provisions for Army Materiel. 22 May 1979.

NAVY REGULATIONS

The following Navy Regulations and Standards address areas of concern to STEP. The TADSTAND's are Standards for Tactical Digital Systems.

TADSTAND 9: Software Quality Testing Criteria Standard for Tactical Digital Systems. 18 August 1978.

TADSTAND A: Standard Definitions for Embedded Computer Resources in Tactical Digital Systems. 2 July 1980.

TADSTAND B: Standard Embedded Computers, Computer Peripherals, and Input/Output Interfaces. 2 July 1980.

TADSTAND C: Computer Programming Language Standardization Policy for Tactical Digital Systems. 2 July 1980.

TADSTAND D: Reserve Capacity Requirements for Tactical Digital Systems. 2 July 1980.

TADSTAND E: Software Development, Documentation, and Testing Policy for Navy Mission Critical Systems. 25 May 1982.

COMOPTEVFOR NOTICE 3960: Operational Test and Evaluation of Software Intensive Systems Computer Software Subsystems. 6 July 1979.

OPNAV 3960.10: Test and Evaluation. 22 October 1975.

MISCELLANEOUS DOCUMENTS

The following documents describe current initiatives, processes, and activities which address areas of concern to STEP. Also included are numerous guidebooks related to software acquisition, development, and testing.

DoD Acquisition Improvement Program (Carlucci's Initiatives). 1 January 1982.

Strategy for a DoD Software Initiative, Draft. August 1982.

Embedded Computer Resources and the DSARC Process. 30 April 1981.

Proceedings of the Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management - Computer Software Management Subgroup/Second Software Workshop. 1 November 1981.

Report of the Army Science Board Ad Hoc Subgroup on Testing of Electronic Systems. 16 April 1982.

ESD-TR-78-141: Air Force Electronic Systems Division Software Acquisition Management Guidebook: Series Overview. March 1978.

ESD-TR-75-85: Air Force ESD Software Acquisition Management Guidebook: An Air Force Guide for Monitoring and Reporting Software Development Status. September 1975.

ESD-TR-75-365: An Air Force Guide to Contracting for Software Acquisition. Electronics Systems Division, USAF. January 1976.

ESD-TR-76-159: An Air Force Guide to Software Documentation Requirements. June 1976.

ESD-TR-77-16: Software Acquisition Management Guidebook: Statement of Work Preparation. Prepared for ESD by the Mitre Corporation, January 1977.

ESD-TR-77-22: Software Acquisition Management Guidebook: Lifecycle Events. February 1977.

ESD-TR-77-254: Air Force Guide to Computer Program Configuration Management. August 1977.

ESD-TR-77-255: Air Force ESD Software Acquisition Management Guidebook: Software Quality Assurance. August 1977.

ESD-TR-77-263: Air Force ESD Software Acquisition Management Guidebook: Verification. August 1977.

ESD-TR-77-326: Software Acquisition Management Guidebook: Validation and Certification. August 1977.

ESD-TR-78-327: Software Acquisition Management Guidebook: Software Maintenance. October 1977.

ESD-TR-78-117: Air Force ESD Software Acquisition Management Guidebook: Reviews and Audits. November 1977.

ESD-TR-78-178: Software Acquisition Management Guidebook: Regulations, Specifications, and Standards. November 1978.

ESD-TR-81-128: An Air Force Guide to the System Specification. January 1981.

USAF ESD 800-4: Acquisition Management: Statement of Work Preparation Guide. 15 January 1979.

Model Statement of Work Task for Software Development. USAF, ESD. March 1979.

Management Guide for Independent Verification and Validation. Air Force Space Division. August 1980.

Guide to the Management of Embedded Computer Resources. Air Force Space Division. September 1982.

Software OT&E Guidelines. Volume I. Software Test Manager's Handbook. February 1981. Air Force Test and Evaluation Center (AFTEC), Kirtland Air Force Base, New Mexico 87117.

Software OT&E Guidelines. Volume II, Handbook for Deputy for Software Evaluation. Air Force Test and Evaluation Center (AFTEC), Kirtland Air Force Base, New Mexico 87117.

Software OT&E Guidelines. Volume III. Software Maintainability Evaluator's Handbook. April 1980. Air Force Test and Evaluation Center (AFTEC), Kirtland Air Force Base, New Mexico 87117.

Software OT&E Guidelines. Volume IV. Software Operator-Machine Interface Evaluator's Handbook. July 1980. Air Force Test and Evaluation Center (AFTEC), Kirtland Air Force Base, New Mexico 87117.

Software OT&E Guidelines. Volume V. Computer Support Resources Evaluator's Handbook. July 1980. Air Force Test and Evaluation Center (AFTEC), Kirtland Air Force Base, New Mexico 87117.

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

VOLUME 4
TRANSCRIPT OF STEP WORKSHOP
MARCH, 1982

SUBMITTED TO
THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION
AND
THE OFFICE OF NAVAL RESEARCH
ONR CONTRACT NUMBER N00014-79-C-0231

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

Volume 4
Transcript of STEP Workshop, March 1982

SUBMITTED BY
GEORGIA INSTITUTE OF TECHNOLOGY

TO

THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION

AND

THE OFFICE OF NAVAL RESEARCH

FOR

ONR CONTRACT NO. N00014-79-C-0231
Subcontract 2G36661

FOREWORD

This volume is one of a set of reports on Software Test and Evaluation prepared by the Georgia Institute of Technology for The Office of the Secretary of Defense/Director Defense Test and Evaluation under Office of Naval Research Contract N00014-79-C-0231.

Comments should be directed to: Director Defense Test and Evaluation (Strategic, Naval, and C^3I Systems), OSD/OUSDRE, The Pentagon, Washington, D.C. 20301.

Volumes in this set include:

Volume 1: Final Report and Recommendations
Volume 2: Software Test and Evaluation: State-of-the-Art Overview
Volume 3: Software Test and Evaluation: Current Defense Practices Overview
Volume 4: Transcript of STEP Workshop, March 1982
Volume 5: Report of Expert Panel on Software Test and Evaluation
Volume 6: Tactical Computer System Applicability Study

Software Test and Evaluation Project Workshop

PARTICIPANTS

Cdr. Mike Anderson	Operational Test and Evaluation Force (Navy)
Lt. Col. M. A. Blackledge	HQ Air Force Test and Evaluation Center
Mr. T. R. Browning	Naval Electronic Systems Command
Lt. Col. Robert L. Christopher	Office of the Director Defense Test and Evaluation
Dr. Richard A. DeMillo	Georgia Institute of Technology
Mr. John A. Devlin (NSIA)	Automation Industries, Inc.
Dr. Kurt Fischer	Computer Sciences Corporation
Mr. Stephen French	Operational Test and Evaluation Agency (Army)
Mrs. Caral Giammo	Defense Communications Agency
Lt. Col. Chuck Gordon	HQ US Air Force
Dr. Robert B. Grafton	Office of Naval Research
Mr. Donald R. Greenlee	Office of the Director Defense Test and Evaluation
Mr. H. Mark Grove	Director, Embedded Computer Resources ODUSD (AM)
Maj. David A. Hammond	HQ Air Force Systems Command
Mr. James Hess	US Army Materiel Development and Readiness Command
Mr. Ed Kennedy	Defense Communications Agency
Dr. J. F. Leathrum	Clemson University

Lt. Col. Vance Mall	Ada Joint Program Office
Dr. Edith W. Martin	Deputy Under Secretary of Defense (Research and Advanced Technology)
Ms. R. J. Martin	Control Data
Mr. Donald W. Miller	Control Data
Mr. Owen L. McOmber	HQ Naval Material Command
Lt. Col. Kenneth Nidiffer	Defense Systems Management College (DSMC)
BG. B. J. Pellegrini	Commandant, DSMC
Lt. Col. Kenny Rahn	Office of the Director Defense Test and Evaluation
Dr. Frederick Sayward	Consultant
Lt. Col. Harry Sherlock	HQ US Air Force
Mr. William Smith	Office of the Assistant Secretary of the Navy
Mr. Jerry Thingelstad	Naval Electronic Systems Command
Mr. Patrick J. Ward	US Army Materiel Systems Analysis Activities (AMSAA)
Mr. Charles K. Watt	Deptuy Director Defense Test and Evaluation

Volume 4

Transcript of STEP Workshop, March 1982*

TABLE OF CONTENTS

Page

Participants 1

Administrative Items and Introduction (Mr. Greenlee, ODDTE). . 3

Keynote Address (Mr. Watt, Deputy DDTE) 12

DSB Study on Embedded Computer Resources
(Mr. Grove, OUSDRE/AM) 17

Software Technology Program (Dr. Martin, DUSD/R&AT) 28

Ada (Lt. Col. Mall, Ada Joint Program Office) 38

DSMC Perspective (Lt. Col. Nidiffer, DSMC) 63

Industry Perspective (Dr. Fischer, NSIA) 81

Army Presentation (Mr. French, OTEA) 103

Navy Presentation (Cdr. Anderson, OPTEVFOR). 120

Air Force Presentation (Lt. Col. Blackledge, AFTEC). 137

DCA Presentation (Mrs. Giammo, DCA) 182

Software T&E Project Status & Plans
(Dr. DeMillo, GIT) 195
(Mr. Miller, CDC) 212
(Ms. Martin, CDC) 216

Summation & Discussion (Mr. Greenlee, ODDTE) 225

*The following is a transcript of a Workshop on Software Test and Evaluation which was sponsored by the Director Defense Test and Evaluation and held at the Defense Systems Management College, Ft. Belvoir, VA on 18 March 1982. Participants were given the opportunity to edit their comments prior to distribution of this document; however, it should still be kept in mind that this is a transcript and not a collection of formal papers from a symposium.

Workshop on Software Test & Evaluation
Sponsored by the Director Defense Test & Evaluation

18 March 1982
Ft. Belvoir, VA

MR. DON GREENLEE: OFFICE OF THE DIRECTOR DEFENSE TEST & EVALUATION

Mr. Greenlee: I would like to say just a couple of introductory words having to do with the Software T&E Program. Many of you have differing degrees of familiarity with the Program in the large. I think most of you, by virtue of your professional activities, are aware of the general state of things in the embedded computer software arena, but let me tell you just very briefly one aspect of it that we observe in the Office of Director Defense T&E. Very frequently, programs that reach OSD for DSARC or other milestone reviews come encumbered with issues relating to the adequacy of testing of the embedded software. It seems that between software and RAM, much of our review time is spent in assessing the degree of operational effectiveness and suitability. For this reason, Mr. Watt, Deputy Director Defense Test and Evaluation, has proposed an initiative which would lead to the development of improved guidelines for software testing within the Department of Defense. This, in basic terms, would provide a basis upon which the review community and the developing community could agree on standards which would relate to the satisfactory completion of software testing prior to milestone decision points. This status has been highlighted in some of our senior documents.

Let me quote just briefly from the Secretary of Defense's Consolidated Guidance, Test and Evaluation Section: "Services should establish cost-effective readiness objectives. Realistic estimates of the readiness levels to be achieved at the time of early operational employment and at maturity should be made. In conjunction with this planning, the service test and resource planning should program the procurement of adequate standard test hardware to support early maturation of reliability, growth, and proof of maintenance design. System test beds, simulation techniques, and evaluation of software performance should be used in the assessment of system operational capability." And later, in the section on Unresolved Problems, i.e., those areas to which the Secretary wishes the Services to devote special attention, "All Services need to give priority to development of tools and techniques for testing of embedded computers and software. Testing of software should be sufficient to achieve a balanced risk with the hardware of the same system."

In the SECDEF's Posture Statement and Annual Report to the Congress, he says, "In support of testing technology advancement, considerable attention is being given to the effective utilization of system test beds and simulation techniques and software performance evaluation. These advances are required if the activities are to provide realistic assessment of system operational capability." Finally, in the Annual Report of the Under Secretary of Defense (Research and Engineering), Dr. DeLauer, under Objectives, states, "Specifically, I will ensure the effective utilization of system test beds, simulation techniques, and the evaluation of software performance in the assessment of system operational capability." So you see that the state of affairs has received attention from the senior management of the Department of Defense. The Software T&E Project is intended to assist in resolving some of the issues and problems in this area.

Our objective, basically, is to attempt to develop policy guidance which would embrace all DoD components, and yet be specific enough to be of use and value in the evaluation of embedded computer software. Additionally, we hope to stimulate the development of improved tools and techniques for software testing, support the development of guidelines and criteria, and promote uniform standards as appropriate in software T&E. We are definitely not in the "standards for standards sake" business here. The Software T&E Project will be primarily composed of the indicated participants. The management and control will come out of the Director Defense T&E. A panel will be established drawing from the Services and military departments. Defense Systems Management College is obviously intimately involved. Industry will be represented primarily by the good offices of the National Security Industrial Association. Dr. Fischer from NSIA will talk to us a little bit later about that. Contractual support in expert areas is being provided by Georgia Tech and Control Data.

That picture is intended to indicate roughly our approach to the Software T&E Project. Our initial phase is principally data gathering. We seek inputs from military, and government, industry and the academic world. We are looking at two sides of the coin; first, the practices and procedures and tools which are actually in current use at this time, as well as the management doctrine and constraints, standards and guidelines under which the military and industry are operating. In cognizance of the rapid changes in this area, we also wish to look ahead and try to predict what trends in both hardware and software will affect software testing. In Phase II, we hope to be able to assess the value and identify deficiencies in the tools area, as well as in the standards area. Phase III is essentially a decision box, at which point we will attempt to determine what, if any, guidance is practicable to be developed. Phase IV will consist of the issuance of that guidance in a suitable format, perhaps as a modification to DoD Directive 5000.3 on test and evaluation. In any event, the various Phases of the Project will be documented and provided for use by the community. That, in a nutshell, is the overall program, and we will welcome your interest and involvement in the program as it proceeds over the next 9 months to a year.

SOFTWARE TEST & EVALUATION ACTIVITY

OFFICE OF THE DIRECTOR

DEFENSE TEST & EVALUATION

OCTOBER 1981

SOFTWARE T&E ACTIVITY

PRIMARY OBJECTIVE

DEVELOP AND PROMULGATE POLICY FOR DoD COMPONENTS IN THE TEST AND EVALUATION OF COMPUTER SOFTWARE

SOFTWARE T&E ACTIVITY

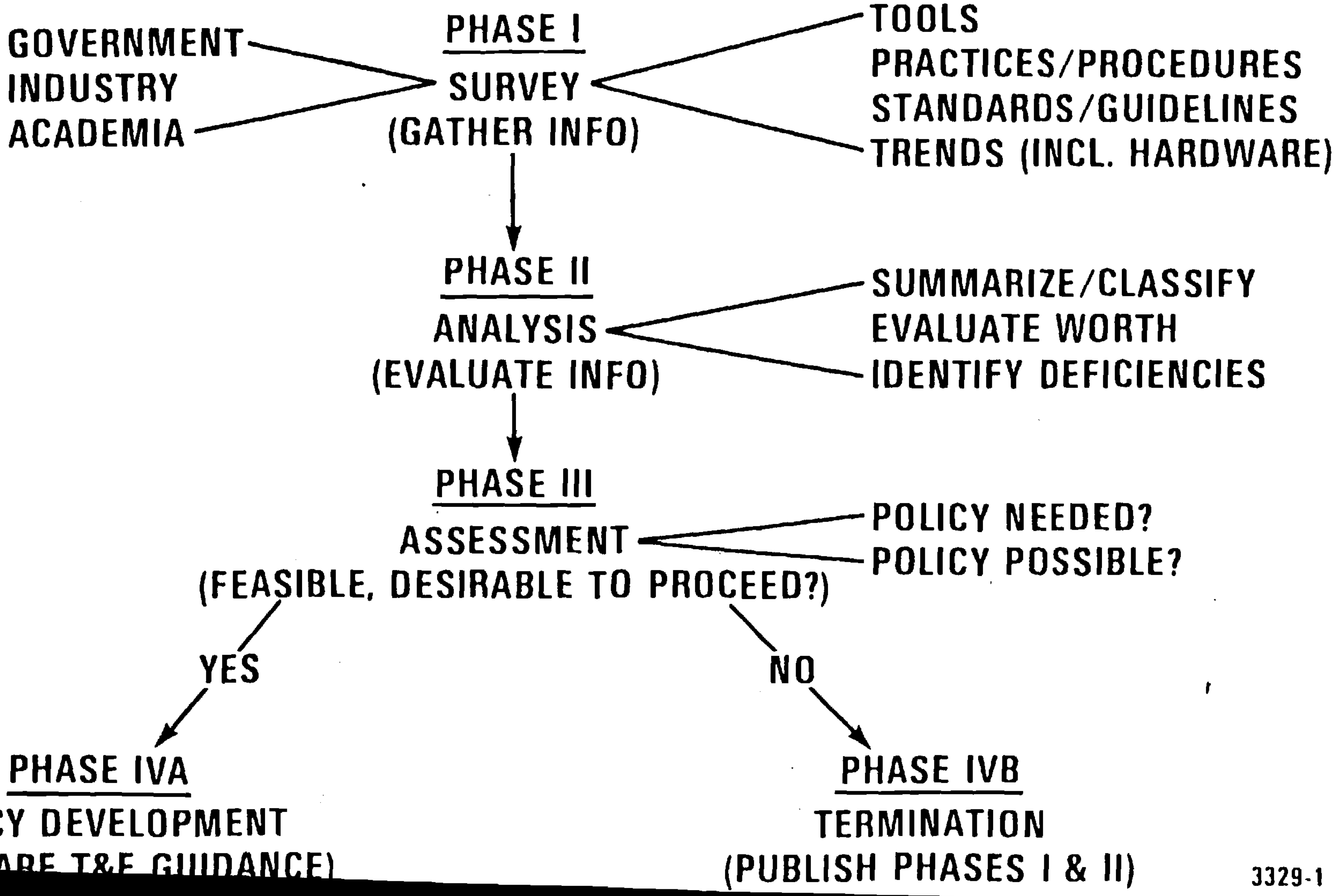
ADDITIONAL OBJECTIVES

- STIMULATE THE CREATION AND APPLICATION OF IMPROVED TOOLS & TECHNIQUES FOR SOFTWARE T&E

- SUPPORT THE DEVELOPMENT OF GUIDELINES AND CRITERIA FOR USE IN SOFTWARE T&E

- PROMOTE UNIFORM AND CONSISTENT DoD STANDARDS, WHERE APPROPRIATE, IN THE T&E OF SOFTWARE

SOFTWARE T&E ACTIVITY



3329-1

SOFTWARE T&E ACTIVITY

PARTICIPATION

DDTE STAFF

DoD PANEL

DSMC

NSIA

CONTRACTOR

Mr. Greenlee: I would like now to introduce Mr. Watt, who will stimulate our thinking and probably issue a challenge or two. Mr. Watt is the Deputy Director Defense Test & Evaluation, with responsibilities for T&E of all strategic and space, naval and C^3I systems. He is an alumunus of Bell Labs and the Naval Electronics Systems Command, among many other things. He was most recently the Technical Director at the NAVELEX Systems Engineering Center in Charleston, SC. Mr. Watt.

MR. CHARLES WATT: DEPUTY DIRECTOR DEFENSE TEST AND EVALUATION

Mr. Watt: Permit me also to welcome you to this most important workshop and to express my appreciation for an opportunity to share some thoughts with you on computing, or computer software. Perhaps I will talk more about what I would phrase computing than I will software. I will not be presumptuous and claim to have the in-depth knowledge of software that I know all of you have who are truly experts in this explosive field of technology. Nor will I belabor the importance of this subject in that such is already well known and documented. All of you are familiar with the pie charts and have seen the tremendous software and embedded computer system investments. We know that in looking to the next decade, software will be the dominant issue in most systems.

Having said that, I would like to talk about two basic subjects. First is "attitudes" or "mind-sets," that we have established about computing in this nation, and second is "opportunities," which I will interchange from time to time with "technological advances." A few months ago, when my staff and I first started discussing the subjects you are addressing today, our primary concern was how do we wrap our minds around such an illusive, complex issue as software. Even narrowing it somewhat to software testing did not solve our problem. Just to structure the task so that we could begin to make a contribution to test and evaluation was a considerable effort. I know that your efforts today will provide ideas and recommendations, and hopefully, they will outline some progressive actions in software test and evaluation. They should result in tools, techniques, and guidelines for a more effective "STEP" in defense. Similar to Neil Armstrong's famous quote, we are taking a giant step for mankind. Certainly for the issues we are addressing and discussing relative to how to do a better job in software testing, no one to my knowledge has any major solutions. From a mental standpoint, we are challenged to find and develop new frontiers. As I alluded to before, our computing effort is aided considerably today by mind sets and by technological advances. I'm gratified to read the numerous documents that suggest, and in many cases even dare to demand, that engineering disciplines and principles be applied to software. Some would even go so far as to state that control should be taken out of the hands of the programmers, the systems analysts and placed in the hands of systems engineers. We are even finding youngsters today that are getting a good grasp on pulling together engineering theory in one hand and computing in the other as they prepare for the future. Some universities are even condoning this by offering degrees in computer engineering. Now, I say this because I believe that it is time that we put more science into computing, into development, and into the software in particular that dominates system performance.

Now, you might ask, "why have I stressed the importance of attitude?" I believe that the attitude we see in this country today is very healthy. Certainly the attitude of our youngsters is most positive as they begin to deal with the next generation of computing. But, I am equally concerned that unless we solve some of the problems, and they are very complex and very pervasive, that we might well find the attitudes of this nation, and in particular, the attitudes of the users, beginning to inhibit rather than contribute to the solutions we all seek. I believe we presently have a mutually coupled effort in attitudes and technology that can considerably aid the process of finding scientific solutions to deal with computing and computer software.

Let me give you an example of the difference attitude can make in computer applications. I had dinner with a group of scientists from Europe not too long ago, and we were discussing the subject of computing and its rather widespread application in the United States. It was concluded that the growth in Europe may be slower than in the United States and that part of this delay may be attributed to the limited exposure and somewhat negative attitude of the more traditional user generation. Perhaps this is a Catch-22 situation. In either event, it is an iterative process and attitudes in this country are an important element in our successful utilization of computers.

Let me address another aspect of attitude. I am somewhat concerned that the application of chip technology in the commercial end of our business is oftentimes much more effective than in the Department of Defense. This point was brought home to me when I was working for the U.S. Congress in assessment of future telecommunication technologies. If you look at new networks being introduced by major corporations in the commercial marketplace, the wideband connectivities that are now possible in both space and terrestrial systems begin to stagger one's imagination. We are beginning to apply similar technologies in the defense establishment, but we must deal with a different mind set and commitment to achieve success. As we get more complex, users must be able to interact on a realtime basis with systems. They must not have a long trail of "programmers and support personnel" to make sure that the system is working. Such situations are likely to cause users to take on negative tones. Technology and scientific processes must be allowed to provide answers instead of creating problems.

In this month's IEEE Spectrum, Dr. Graham presented an article on the software crisis created by the large manpower costs of programming new information systems. His contention was that breaking the bottleneck requires a visionary approach. He stated that we need to actively seek a fully integrated programming support system and that solutions include computer advances that offer advanced programming aids. Project information management that exploits the benefits of computers in interpersonal communications, documentation and recordkeeping is described as being among the major items that are essential for software development projects. Perhaps the foremost requirement is for direct access to computers by end users. In particular, they must not require programmers or professional support personnel for routine data processing.

I believe that the bottom line of most of these concerns and attitude indicators gets back to cost. In fact most people, when all is said and done, respond to the typical question of, "What is your major problem?", --- with a simple statement of cost. But, I don't believe that cost is the root cause of the dilemma. The root cause of most system failures is inadequate software, which is a result of an ineffective application of technology. In all too many instances, we have not properly applied technological scientific processes and that is the significance of why we are seeing humongous cost increases and difficulties. In order to build on this statement, let me go back to the point that I made earlier when I said I was going to talk about attitudes, as well as opportunities or technological advances. Never before have we experienced or observed such a blurring of systems as is being experienced today. I contend that the technological blurring is caused somewhat by systems engineers who are effectively doing their job. They are utilizing computers in solving a multiple of large complex problems. A more integrated approach is being utilized in the application of computers, and we are finally getting a handle on end-to-end system engineering. The embedded computers in defense systems is a familiar example of technological blurring. The reference that I made to communications or teleprocessing is another example of networks that are certainly becoming transparent information pipes, as we provide wide band connectivity and alternate information channels which are rapidly available on a realtime basis for those who would like to interconnect. Some of the blurring of issues is even entering into our court systems, as we recently have read about some of the decisions concerning AT&T and IBM. We are now finding it difficult to accurately define old lines of demarcation. This is a result, I believe, of good systems engineering. Now, I'm not condoning the fact that we don't properly control; I'm just saying that where computing ends and other processes begin is no longer as clear in anyone's mind as it was a few years ago. The impacts of such systems are explosive, and I believe that they also have many economic implications. Needless to say, there are also social, demographic and political implications. Those things that were thought possible tomorrow, suddenly are becoming reality today, even yesterday. Integrated circuits with an extremely high density of 1 million components, 2 micron line widths on chips requiring only a few milliwatts of power, continue to stretch our minds. Powerful new 32-bit microprocessors, termed micro-mainframes, now are suddenly coming on the market. Three such systems were recently announced in Spectrum. We are finding that these systems with solid state devices and nonvolatile memories of 1 million bits are complemented by 20 megabit secondary devices. These are but a few examples of what is happening in the area of computing. That is why when I started out I said I would talk more perhaps about computing than I would just the software aspect of it.

I believe that technology is and will continue to be very explosive. This trend may further readjust our attitude to acceptance of even cheaper hardware in place of costly software. The takeover of many software functions by hardware may help programmers to develop high level languages that are shorter, more efficient, easier to write, compute, and hopefully, to debug. So, we may begin to see many of the functions that perhaps historically have been those of the programmer--the software and the logic--begin to be converted into hardware. Now, you have probably heard this before and considered that it is nothing new. What amazes me is that recent forecasts on similar technological advancements, estimated to require a decade, have occurred within a short time span of 3 years. Therefore, many of the problems we grapple with in software today could well be solved with hardware tomorrow.

I could continue to talk about the maze of technologies that are impacting computing, but I will hold these for another place, another time. I believe that my point of technologies, opportunities, attitudes is either well made by now, or my teleprocessing system isn't working very well. I did want to challenge you with these thoughts as you begin to start your workshop.

Now, in closing, I would not have you think that computing or developing computer software has become a science or is even close to it. It certainly is not; but, I believe that your being here today is certainly indicative that we are making progress and the trend is in the right direction. Your tasks are made somewhat easier by the technological advancements and attitudes I've touched on this morning. I firmly believe that the time and place for "STEP" is here and now -- thank you very much for your time and attention.

Mr. Greenlee: Thanks very much, Mr. Watt, for that stimulating and far-reaching keynote. Since this is being taped, I must correct one statement in that presentation. Mr. Watt suggested that he was not a computer software expert, and I ask you not to believe that and correct your notes accordingly. He is an expert in that, as in many other areas of software and systems engineering.

Our next speaker will be Mr. Mark Grove, who works in the Office of the Under Secretary of Defense (Research and Engineering) and is the Director for Embedded Computer Resources. I expect that any of us who have had any contact, even tangential, with the ECR field are well aware of his name. The latest major activity with which he has been associated is, of course, the recently completed Defense Science Board Study on Embedded Computer Resources, which was just recently reported out. As you are undoubtedly aware, one of the thrusts of this ECR report by the DSB is the proposal for standardization not only of high-order languages, but also of instruction set architectures. We feel this is very key in its impact on software testing since obviously the difficulties in testing are in some way proportional to the complexity of the universe that one must deal with, and conversely, limiting and standardizing the HOL's and ISA's can very materially ease the job of the software tester. For this reason, I've asked and Mr. Grove has kindly consented to talk to us a little bit about embedded computer resources, the findings of the recent DSB study, and how they devolve upon computer software testing. Mark.

MR. MARK GROVE: DIRECTOR, EMBEDDED COMPUTER RESOURCES - ODUSD(AM)

Mr. Grove: It is a pleasure to be here this morning. Charlie's remarks remind me that we did not give sufficient attention to the test and evaluation aspects of computer software and hardware during our Defense Science Board Review. That is something we probably should correct the next time around. I think it would have strengthened considerably some of the conclusions that were reached and probably would have helped us order the discussion. But, we will try to repair that; I don't believe we have done anything inconsistent with what your needs are. Clearly, test and evaluation are central to the policy decisions that we need make with respect to language and to architecture. I certainly agree with the discussion about the cost and how that must be capped by an approach that levers on the ability to evaluate the systems themselves during the T&E phases. We haven't done a good job of that. There are some other points that I may, if we have a little time, discuss relative to the hardware/software tradeoff.

This was the charter that Dr. DeLauer gave the Defense Science Board in establishing the Task Force on Embedded Computer Resources Acquisition and Management. A principal motivation for this study was the extensive disagreement both within the Department and between us and our colleagues in industry about whether of not this type of management is appropriate, or whether we would be better off with a more general statement of requirement and to let industry do their thing in meeting those requirements with whatever technology is on the shelf and in their plant. But, at any rate, he asked us take this look to review and make recommendations on the acquisition, management, and utilization of digital technology. We did not, therefore, get deeply into some of the other things that need to be done in providing a better technology base for software/hardware engineering with respect to "computering."

Those four questions each abstract about a paragraph which describes the concerns about management policy, some of the key programs that are going on in the Department now and those that are planned for the near future. A key point had to do with the process of management and oversight in the Department of Defense particularly at the OSD staff level --- a process that is somewhat emulated down through the Military Departments. But, it's a rag-tag kind of operation and almost every OSD staff office has some interaction in the management and oversight of programs to provide a computer capability for the Department of Defense. At the time this charter was written, and this was August the 20th, I believe, of last year, there were some legislative changes in the mill that had not been really completely acted upon, and so there was some uncertainty about that. It had to do with exempting the DoD from the Brooks Act under which much of the commercial equipment has been purchased over time. That has some implications that we'll go into.

The Task Force was a fairly broad-based Task Force. We did have people from the current industry, both producers and integraters; we had the academic community represented; the not-for-profits. We had an outstanding group of senior military participants who really did participate. They attended most of the meetings and participated intensively. It is sad that last Saturday Admiral Lewis died of a heart attack; he was quite a gentlemen who contributed strongly to the acquisition part of this.

We use the term "embedded computers" extensively. Let me remind you of what we meant by it in the Task Force and what we generally mean by it in the Acquisition Policy part of DoD. It doesn't have anything to do with the source of that equipment; whether it is commercial, off-the-shelf; whether it is large; whether it's small; but it has to do with the applications. And, we're tending to move from the term "embedded" toward something more general, "Mission Critical Computer Resources" (MCCR), which is a better descriptor of what we really need. It describes better that we need access to the commercial market place as well as to the specially designed equipment that has traditionaly been thought of as "embedded computers." This also immediately brings up another problem in that some of our policies have been put in place with the assumption that the materiel that we were trying to control was specially designed, militarized computers, and particularly those that are deeply embedded within subsystems. As a result, the policy we are proposing to standarize Instruction Set Architectures (ISAs) needs to be re-evaluated, since we do have a better access now to the commercial market place.

I want to make clear that we are not saying that we want to force this small set of instruction set architectures on those applications where the commercial market place can actually fulfill the requirement.

But, it's key that these five areas are, with respect to the Department of Defense, exempt, and that is in capital letters, from all of the acquisition process that is covered within the Brooks Act that you may be familiar with. The exemption does not include routine administrative and business applications. It also does not include "routine" logistics applications. So, the gray area has moved from the region between specially designed and commercial off-the-shelf to a gray area in application. Certainly, on the battlefield, we have administrative and business applications that need to be fulfilled. In some cases, it will be necessary to do that with a militarized piece of equipment. In those cases, the policies we are talking about would apply. Where commercial off-the-shelf can meet those requirements, then these policies would probably not apply.

Again, I might bore some of you with this, but we need to know what the Task Force meant by instruction set architecture, and it was basically this. It's key that we are talking about the interface between software and hardware, and that interface is an important one to the test and evaluation process. It's clearly an important one also to the hardware and software design processes, but it does not imply a specific hardware implementation or instantiation of the interface. There are things included in that interface which do impinge upon the design process and are particularized in a given piece of hardware, but that is not what we are trying to control at this level. That is another problem, and on the battlefield, you may want to do that. There are some exemptions even in the militarized environment where we feel we need to control this in order to manage the logistics and maintenance.

There are some applications where we don't expect the software to change. And, if that is the case, it's not quite so important how that's done. You need to know about it, and you probably need to configuration-manage it, but not in the same way. Let me remind you that DoD Directive 5000.29 was issued in 1976, and it basically made these points about computer resources used within systems. They really should be considered a subsystem of major importance and treated so throughout the development and lifecycle support of a system. Try to emphasize that you lay requirements on the computer subsystem then the same way as you would lay requirements on a radar subsystem. There are some important things that need to be done, and those requirements need to be validated the same as any other system requirement. Perhaps, this was ill stated of the configuration management process because immediately the bureacracy said you apply the same kinds of configuration management that you do to hardware, and that may be inappropriate. Indeed, software must be able to change, else you don't need software. But, what it must do is change in a controlled way, and you must be knowledgeable about what the state of a given system is at a given time. I'm not so sure we have solved that problem yet.

DoDD 5000.29 said also, and here's a charge observed more in the breach, that any unique software required to support a system throughout its life shall be deliverable. That's not an option. Turns out to be ignored, but it's not supposed to be an option legally. The other key thing that caused action was the policy that software languages should be standardized after recognizing that applications code should wherever possible be developed in a high-order language. Somebody needs to control those languages. Generation of dialects is one of things that causes that software cost curve to exponentiate over time. Following that up, we set up these seven languages as interim standards that should be followed and that worked pretty well.

What was next proposed and what caused grief with the industry was their misunderstanding of what this meant, and they don't want to understand that ISA doesn't mean hardware, because in their parlance it does mean hardware, and they control producer economics by controlling ISA's. We propose to adopt a small number, not 1, not 2, but a small practical number and manage pretty much the way we said we ought to with the software. Each Military Department would be given the assignment for controlling and being able to evaluate whether an ISA was actually properly implemented, such that software and other things could be transferred.

We suggested that a waiver process would be necessary; there is no way we can forecast what all the requirements will be, so waivers will be appropriate, particularly in the early times; but we want to know what that process is in advance, rather than after the fact. The principal criteria for not using the ISA, assuming the list is good, ought to be that a particular application has technical requirements or the economics of a given application don't allow it, not just because we don't want to or our selected contractor says he would rather use his proprietary approach. We also said this is a dynamic situation so the list ought to be continuously monitored to assure that we aren't making a mistake by implanting this policy. And, if it does turn out to be a mistake, we ought to get rid of it, not just let it go along and then be ignored, which is a rather embarrassing way to operate. We also need to be able to add or delete ISA's in order to meet the requirements of the systems. Here, again, is where the industry people got a little bit upset with us. They said that in order for us to do this, we must have if not unlimited rights and data, we must have clear rights so that at the beginning of a program, we can explain that and make it a basis for a fair competition, that there aren't surprises coming up later. Clearly, that says you don't start with a commercial ISA and get a license for it, because immediately there comes a contractor who is doing militarized and commercial product promulgation. He has an unfair advantage from the start. Anybody who wants to enter that process has to capitalize quickly in order to be able to provide product, and if he is only covered in a license for those things that are militarized, he can't write that off against the commercial market place, and he also has to worry about the leakage from the military to the commercial market place. The risk is significant. So, our policy is founded on the precept that it must be a government owned architecture that anybody, the first time around at least, can enter fairly and equitably.

The Task Force then identified these 7 principal issues that should be concentrated upon, and there is no need to go through those in any detail; we have touched upon most of them. These two were added for emphasis. The Joint Chiefs of Staff asked us to say a few words of advice to them about whether or not Ada, the new high-order language, common language, for DoD, is sufficiently far along that they ought to consider it for the WWMCCS systems upgrade. And further, should they consider the hardware that is expected to come out of the military computer family, where applications in those transportable parts of WIS are expected. I mentioned we needed to take a look at how the management process was working.

We did do something a bit differently in advertising to the public in the Commerce Business Daily. We received 20 responses ranging from some of the largest companies down to individual responses. That may turn out to be a contribution to the total DSB process in the future.

We concentrated principally on this idea of managing instruction set architectures so one of the first things we did was to look at the comments from industry and try to evaluate them as to what impact they were going to have. Clearly, industry and we are interested in competitiveness. And, such a standardization policy may well eliminate some suppliers if they don't have the products at the time that a given RFP hits the street. That can't be considered positive if we are eliminating anybody from what we would like to say is fair and open competition. On the other hand, since people can enter on an equal footing, this surely will add some other contractors who do have something to bring to the table in the computer business, that certainly is a "pro". We have been locked in to a rather small set for a long time, and that is part of this cost escalation problem. There is always the fear of stultifying technology insertion if you standardize. The word standardize usually runs a chill up the spine of the technologist. And, certainly, if the standardization is done irrationally, it could impede the injection of technology. However, if it is done properly, which I think we've demonstrated in many cases, it can actually accelerate the injection of technology. It gives you a stablilized base of requirements so that you can plan your technology programs, and therefore, make them more coherent and therein supportive. No doubt, the software development process would be improved if there are less targeted machines that you have to worry about. The degree of that depends on many things, but the main reason I think that we came out with the conclusions that we did is life cycle support. This is the driver in the computer business or any other equipment for militarized application. Everybody concentrates on the acquisition phase, the early acquisition of the original equipment, but there is a minimum of three times, and it probably ranges up to 10-15 times that investment required just for spares for complex electronic equipment. The Army, when they're fielding equipment, when you consider all the war reserve spares involved, the pipelines that are involved, 7-9 times the numbers of boxes are required just to keep that logistic pipeline going almost independently of the reliability and maintainability of that equipment.

We really make our decision based on the wrong data, generally. And, it is this kind of recognition that drives us to a standardization policy, some kind of policy or limitation of variance. And, it is that one that the people who normally work in the commercial marketplace, selling to universities and individuals don't understand or don't want to understand. Everybody knows what the personnel training problem is and by limiting the variance there, we can't hurt that, I think we will help it. Our operational people are very concerned about the flexibility that they have when the shells and bullets begin to whistle around their ears. And, this is, again, something that doesn't happen even in the most ruggedized commercial equipment in a steel mill. However, if you are in the middle of some sort of a firefight and a critical piece of hardware goes down, namely a computer, to be able to trade-off quickly with something else that's in that same general location is very important. Again, it is something that doesn't have an analog in the commercial world.

There were findings from the DSB panel on these seven topics. The principal one, the one where we spent most of our time was on the proposed 5000.5X instruction set architecture standardization process. The next most important probably is the management and organizational aspects. The Task Force did recommend after a lot of discussion that the arguments were compelling in favor of standardizing ISA's. That action is a necessary thing for us to do, but not sufficient. There are a lot of other things that must be done in order to bring that cost curve you are all familiar with down. So, they did recommend that we issue the instruction expeditiously, but that we qualify its scope, as I tried to early on. It does not cover everything. It covers those things for which we need to provide organic support and for which the off-the-shelf product is not appropriate. We want to limit the number of ad hoc designs that are necessary. And, that management cannot be passive, it needs to be very active. Technology is moving fast, which means advantages will be overcome if we just put a policy in place and then walk away.

The key programs that we looked at which would implement this policy, if it were indeed a policy, are already in place, at least for this coming generation, and these are the representatives of that. There are three separate approaches to this problem because the three Military Departments face, in general, different problems. But, more than that, these programs which grew up by themselves without the need of an OSD thing to tie it together came about in different time periods. The Navy had long since decided that they could not support shipboard operations with a random selection of hardware. So they put the policy into place many years ago, and what they are doing is iterating that with technology over time. They have a large software investment, and that needs to be conserved simply because we can't find the money to replace it, even though we would like to. The Army had something like 50 separate computers in some 70 systems. They were using 44 languages at the time. That was just eating them alive. So, they decided that they needed to bring this under control. MIL-STD 1750 was the Air Force's approach to get out from under some of the sole-source acquisitions that they been forced to.

Now, from all those implementation programs, the findings and recommendations are these. Those programs indeed did meet the policy intent, and that's not surprising because the policy was developed after the programs. We would suggest however that the idea of having only one producer initially could very well be a mistake, and we have experienced that in trying to establish second sources after the fact. Almost impossible to do that. So, what we were asking, particularly of the Army and Navy, is to look up front at carrying multiple producers rather than try to inject them later. There was also some idea that the regulations under which acquisition is carried out were setting some of the timetables. The maximum length of a contract being five years with the law being interpreted as that was causing them to do technology injection programs on a five year cycle. We want to try to make sure that is not the driver, that there is some flexibility.

The approved list of high-order languages, I've already talked about and mentioned what they were. We do have a Joint Program Office that is trying to bring along the common language, Ada, and Vance Mall is with you to talk about that sometime during the forum.

Everybody is giving accolades to the Ada program, so your work's cut out for you to make sure that it works. I say "you", but that's an inclusive "you". Our work's cut out for us, since we have oversight of that program. The Task Force did say that it's time to update DoD Inst. 5000.31 and get a couple of the old languages off so the people could take the policy seriously. We're going to do that within the next very few weeks. Again, the Task Force as well as others said; "Let's make sure the Ada Joint Program Office is properly supported". We felt that since Ada is such a critical part of the military computer family program, there would be a little more attention there in that the Army could accelerate this whole process if they would put a little bit more emphasis behind it.

From a test and evaluation standpoint, if you are going to put tools and processes in place to evaluate the software, then certainly we would like to see those tools centered around these languages when language specific issues are at hand. And, of course, many of the issues may not be language specific, although the tools very well may. Even though we are pushing Ada for the broadest possible use, there is no way that existing languages are going to go away in the near term. Another issue is that all of the software that you ought to be worried about is not the applications code and equipment --- there is probably an equal amount that is dedicated to the automatic testing business from factory to field. From that standpoint, a little attention needs to be given to the Atlas type program.

Reasonably simple changes recommended to the overridding or capstone Directive 5000.29 are these. It is necessary because of the sunset clause that it has to be reviewed in 1982, or it goes away, and then we would basically have no policy to be followed in software acquisition, and therefore in the computer acquisition, and the observation was made that there still are problems. They haven't all gone away. And, another key recommendation of the DSB Task Force is that if the acquisition process can be made to accomodate it, the "software first" approach should be emphasized. But ofttimes, the hardware is chosen simply because of the leadtime and the comfort feeling of having the target hardware available upon which to develop software. As a result, we make some serious mistakes, and it seems that, since the hardware technology is changing so fast, the software can be developed in a different way, through emulation or some other manner, and largely matured before the hardware decision has to be made IF the leadtime for delivery can be handled. Another thing that should be done more, and we have been fairly poor about this I think, is to have specific software considerations in the source selection process. Again, that wraps into the T&E.

We are not consistent in the way we address industry from the three Military Departments and the other Agencies. As a result, they have to diffuse their talents; I think that tends to degrade quality, so we are asking for a consistent set of DoD-wide specs and standards. There's activity in process to get that through the Joint Logistics Commanders and others.

Now, the last thing I want to say a couple of words about is our management approach. I mentioned that we don't have a very consistent and coherent management process. Each program manager does pretty much his own thing. Each staff element in OSD approaches it from a different point of view, and we are not helping each other very much. We tried to coordinate these activities with the Management Steering Committee for Embedded Computer Resources (MSC-ECR), and clearly, the problem has outgrown that kind of process. The DSB Task Force has recommended that there be an explicit designation of a policy official, a senior official, responsible for all acquisition management and management of computers in the DoD. And that has been done. The first of February, Secretary Carlucci designated the Under Secretary for Research and Engineering as that policy official and he's responsible for all computer acquisition and management that is not under the ambit of the Brooks Act. On the 16th of February, that responsibility was delegated to me. We are going to try to centralize some of the activities. That doesn't mean we are going to bring everybody into one big office, but we are going to try to establish a set of communications channels across OSD and work for a little more friendly communication. And so the face that we put toward the industry and the policies that we put forth for the military departments to follow are at least coherent.

Then, if they are wrong, there is no problem determining what's wrong, and we can change it. We are going to be a little more specific about our oversight function on the R&D and acquisition processes in the Military Departments and that is not counter to Carlucci's decentralization policy. That means you must pay attention to the words "controlled decentralization", that Dr. DeLauer insisted upon, and it means we are going to insist on having visibility into the programs. When we can see things that could help one program going on in another, we want to make sure those people get together and that there is some advantage taken of that.

The terminology issue needs to be clarified, and when the DSB report comes, it will have an annex in it that consolidates, to the best we can now, some of the different definitions of terms. Perhaps we can then spend less time at future meetings and at future forums worrying about what the words mean and worrying a bit more about what needs to be done.

That report should be out soon. If in the interim, you have a need for more details on the findings and recommendations, we can work that out.

I didn't say a lot about software, or much about T&E, but I think those recommendations will have an impact on how easy that job is for you and us to accomplish.

Thank you.

VIEWGRAPHS

USED BY

MR. GROVE

FOR THE

DSB STUDY ON EMBEDDED COMPUTER RESOURCES

PRESENTATION

"NOT AVAILABLE"

Mr. Greenlee: Thanks very much, Mark, for that excellent overview of ECR.

Our next presenter was to have been Mr. Joe Batz from OUSDRE/R&AT, who is the proprietor of an interesting new initiative called the Software Technology Program or STP, not to be confused with our acronym. This is another initiative like the initiatives which have come out of the DSB study, which I believe all software testers must be aware of. The intent, as I understand it, is basically to improve the efficiency of the software development process. Some time ago, ideas and thoughts were solicited from industry and academia, as well as within the government, on notions which might lead to the improvement of software development. How can we get to debugged computer programs better and easier? That generated about three forklifts full of responses from the community, which Joe and his people have sifted through, and the Software Technology Program, as it is presently defined, as I understand it, has a couple of very interesting thrusts, most of which are really key to software testing, one being the emphasis on development of a program support environment and the other having to do with the notion of reusable software. Joe, unfortunately, could not be here this morning. He was presented with a conflict at the last moment, so we are very fortunate to have participating today Dr. Edith Martin, who has been involved with the STEP program since its inception. She is currently the Executive Director of Control Data Corporation's Atlanta R&D Center for Government Systems. However, she is about to assume the position of Deputy Under Secretary for Research & Engineering (Research and Advanced Technology), in which capacity she will have management responsibilities for implementation of the Software Technology Program. This may be the first forum in which you have been introduced in your new capacity. So, we are very pleased and honored to have Dr. Edith Martin talking about the Software Technology Program.

DR. EDITH MARTIN: DEPUTY UNDER SECRETARY OF DEFENSE (RESEARCH & ADVANCED TECHNOLOGY)

Dr. Martin: As he had indicated, the software technology initiative is within R&AT, Research and Advanced Technology, and within the Electronics and Physical Sciences Directorate, and Joe Batz is heading it. He's been coordinating extensively with the Ada Joint Program Office and with Larry Druffel. That coordination has been very good, however, we do anticipate that the AJPO will come over under R&AT in the near future. This move will formalize that coordination and among other things consolidate the computer related activities. The software technology initiative, even with its several forklifts of materials, is still seeking input from the community. Those of you who have suggestions to make, certainly have an opportunity remaining to embellish that repository. We would like very much to have by September 1982 an implementable plan, one that we can hand off and say, now it's your turn and go do it. We did suggest that the software technology initiative be a subject for the Defense Science Board Summer Study that would be in August. That is approximately a two week long concentrated activity. Subsequent to the DSB and prior to that September deadline, we would look for a quick turnaround review from industry and from other segments of DoD and the Services. Hopefully, you'll be participating in that, but in any event, look for something to come out in August, and mark your calendar to respond to it by September.

Now the role of test and evaluation in a program such as the software technology initiative, a program that is directed toward developing better software, cheaper software, software that can be developed faster and is going to last longer is obviously immense. We can see ourselves addressing in the final software technology plan some of these test and evaluation topics. Now, we need a lot of input on how they should be constituted. Listed here are some possible topics.

Reusable software. The idea of reusable software has been hanging around for a long time. We have all heard in other domains, "if it ain't broke, don't fix it", and this is the philosophy of "if it works, use it". I think that we need to pursue what it means to have reusable software; how do we describe it; how do we disseminate it; how do we make it available to people. You know, there are a lot of mechanics that surround that concept that must be flushed out in order for us to actually implement it and use it in the development of new systems.

Designing to test. We will focus a good deal of attention on how we go about developing systems that are easily tested. So often today, testing is done after the system has been completed. All of us know that what you have there is a Gordian knot. Trying to disentangle that and say that it does everything that you intended it to do correctly all of the time is very, very difficult, if not impossible. So, we have to look very hard at designing systems that can be tested in components and then as integrated systems. These should be cleanly put together and cleanly testable.

Development of evaluation methodologies and metrics. There are many methodologies and a few metrics available. They're not in common use across DoD, certainly. Part of what this T&E study is about is finding out exactly what the state of the practice is in DoD and in industry. I don't think we can say there are no voids. This T&E Study will be very helpful in identifying what those voids are. Research and Advanced Technology is the place from which the funding and support will come to see that we fill those gaps. There will be very strong endorsement for anyone who is pursuing research that will help in the definition and development of these methodologies and metrics. Developing a laboratory for experimenting, for the development of metrics is one area in which the Services could jump in and help out.

Develop consistent testing tools and incorporation of these tools as part of a standard environment. Obviously, the Ada integrated environment is what we are looking toward as the home of these tools. It is very important that if we develop new capabilities that we make these available. That we have the capability to evaluate or qualify one system relative to another and say that it is .5 on the Richter scale is important. We cannot do that today. Having a number of tools available, maybe different tools for different type of systems, having consistent metrics, having the capability to share those tools and not reinvent them would be very, very useful to us.

These six topics are sort of out of order, but as I was making up this "impromptu list", I realized that I had left off something very important and that is prototypes or throw-away software. We really do need to get a better handle on what that means. What do we have to provide to the developing community to allow them to prototype a system. I'm sure that we can develop systems for prototyping. Generic simulation capabilities ought to be established.

Those are the thoughts that I have. I have not discussed some of these with Joe Batz, but I will. I'm sure from the discussions that we've had he would concur with all of these things. Now he can go write them up. Are there any questions?

Software T&E, I agree with Mark, did not get satisfactory attention in our DSB study. We were focused very much on hardware. Software T&E certainly is one of the most important areas that we have to address in the software technology initiative. And, so, your involvement in that, any input that you might have is going to be very well received. Don't be bashful, don't be voluminous, but get us the information. Or, if you think there are people that we ought to be talking to, make that known. If this was all that the software technology initiative was about, it would be a very, very full program. There is much more to it. We will endeavor to keep you informed on the full scope of that program. We've got years of work here, and we need all the resources we can find.

Mr. Greenlee: Thanks very much, Edie, for stepping in there very quickly. Questions? Surely. Dr. Leathrum from Clemson University.

Dr. Leathrum: The previous talk and this one brought to focus in my mind, the problem that we that call ourselves software engineers need to give a bit more attention to. We tend to focus in on a software lifecycle and problems that arise in that lifecycle. But I think a lot of our problems arise particularly with embedded systems, where the system lifecycle is out of phase with the software development lifecycle and maybe the procurement cycle gets out of phase as well. This came to mind as I was reviewing the applications software for an Army computer system last night, and again, the procurement lifecycle seems to have reached a climax, and lo and behold, the software tools were primitive. The keeping in phase of these lifecycles, I think, is something that needs to be given some attention as well.

Dr. Martin: Well, I'm not sure that I understand how you can say that the software lifecycle is out of phase with the system lifecycle. You mean in hardware/software entities. The procurement problems are going to be there no matter what we do. That is just a fact of life.

Dr. Fischer: Kurt Fischer from Computer Sciences Corporation, representing NSIA. I know one experience that I have in that regard is that in our major weapons systems, we have a phase called production. But, for those of us who deal mainly in software, the production phase of the system lifecycle is always almost nothing in software because all of the software is developed during the development phase.

Dr. Martin: Are you saying that hardware production is preceding software production?

Dr. Fischer: No. Just the opposite. That software production is done during the development phase. With hardware, during development phase, we might make one or two prototypes of the hardware and put the software in. Where, insofar as putting that software in production, all you do is make copies of it. So, there is no software production done during system production phase.

Dr. Martin: OK, and your point is ...

Dr. Fischer: I'm just backing up what the gentleman from Clemson said, that sometimes during the system lifecycle the phases are out of sync. We produce our software during the development phase, we produce our hardware during the production phase.

Dr. Martin: I wish I had a couple of view graphs that I use for another presentation. I wholeheartedly agree. The view graph, if I could describe it, is one that shows the PERT chart of the development of a hardware system and it's very, very detailed. Essentially, that is because we understand it. We have a software PERT chart and it has two nodes, development and test. And the earliest possible start date on development of software, is sometime before completion of the hardware. You know, we do go off and fake it. The latest possible start is at the completion of the hardware. The earliest possible start date for testing is at completion of the total software system today. The latest possible start date is after the system has been deployed! Or never! Testing whether or not a system does what you thought it was going to when it was out in the field, is an observation, and that's all it is. That's pathetic. I think that some of the things that are happening as a result of the Ada activities and as a result of the embedded computer resources standardization activities will help. One, if we use standard hardware, we've got the hardware on which the software is supposedly going to run available for the total time in which the system is being developed. That's going to be a very big help. In the past, we have been guessing on how the hardware was going to operate. We will still guess at some of the interfaces, however, having a standard language and having standard tools will certainly be helpful in developing software because previously we have not had those aids. We are moving the capability to develop software closer to the outset of the total system development activity.

There is no reason why we can't do that with testing too. In fact, I don't think that with good conscience, we can continue the way we are now. We all know that we have to test the system upon completion, and we all know it is supposed to work. Part of your program plan really should ask "how are you going to go about testing this system along the way to give some assurance to those people who are dumping millions of dollars into its development, that when it is completed, it will work?". To say that after it is complete, we'll stick in a thermometer, and if it is past 105, we'll call it sick, is not good enough. We can move that up. I think that in the research community, there are techniques being developed that say that it's straightforward, it's not going to be that hard. If we can't break down the problem to a testable form early on, what makes you think we can do it later? So, it's a matter of how we manage what we do in a lot of instances rather than whether or not there is a technology there that permits us to do it. We have not been managing the software/hardware system as an integrated system from the outset. We have developed our hardware, we know that the software is supposed to run on it, we go off and we develop our software, and at the end, voila, comes a marriage, and hopefully, it works out. It's just unnecessary to continue doing that. I guess, what I'm saying is that part of the problem you are observing, although I think it is real, there's no technical reason for it. It's a management problem, and it is one that is resolvable if we simply bring it to the attention of the system designers early on and place a requirement that the test plan be there in advance of initiation of development of the system. There is no reason why we can't resolve that part of the problem.

The procurement problem is still going to be there. There are efforts underway to resolve some of that in the hardware area. I think that we have made great strides by having the procurement decision making power for embedded systems removed from under the Brooks Act almost in total. That means that the general purpose machines that are going to be purchased for embedded computer systems are not going to fall under Brooks, and that, procurement-wise, is a major step because it is probably going to buy you 2-3 years. So, I guess, I shouldn't be too harsh on the procurement problem, but it is going to be a difficult one to work with. In the area of software, we've got an advantage because they don't understand it, maybe we don't either, but at least, we are working with something that we can say we have a clean slate to start out with. We are not trying to reverse a trend that already exists as we were in the procurement of general purpose hardware. Trying to make an exception to the Brook's Law with a case by case instance was very, very grueling. So, we do have a better opportunity in software. Does that answer your question?

Dr. Leathrum: My point was that the development of the software should be controlled such that it climaxes at the same time as the rest of the system.

Dr. Martin: Agreed. Part of the draft papers that I would presume that most of you have not seen are components that will be in the software technology initiative that talk about the methodology or the management approach to the software system lifecycle. Those have been sufficiently rough, that they have not been put out for review except by a very few people. But, that problem is understood or recognized at least, if not understood, and is being addressed. Any suggestions you might have on that are very much welcome, and I'll be glad to get you a copy of the draft methodology or management paper, whatever we end up calling it, once that's ready for review.

Mr. Watt: The lack of synchronization between the hardware and the software is an issue that needs to be addressed in depth as a part of this study. The immaturity of the software relative to the hardware, the lack of synchronization, that entire issue, I think, is one that any ideas on would be welcome, and as we continue this effort, we need to address the alternatives for solving that problem.

Dr. Martin: I think that one of the major problems that comes about is a lack of advanced planning or at least not living by the plan once it has been established. Because there has been a hardware first approach, as we all know, the deadline doesn't decrease; the time to develop the software simply condenses, right? There is a point at which you can no further condense the development time for the software. What does happen as a result of hardware delays is that the software that is going out is immature. If you don't have mature software, then it is hard to say that the integrated system is truly going to work together. The other side effect is that the dollars also don't change, so as you start incurring overruns in the hardware, the software is simply going to cost less. That's ridiculous, but it happens. So, you're working with condensed dollars and less time, and the Mythical Man Month says that won't work. You know, as you condense the time, the dollars go up astronomically.

You get to a point when you can't do the job no matter how many dollars you put into it. So, you see, there are some parameters that are not being manipulated as the system synchronization gets screwed up that, again from a management standpoint, have to be factored in. We have to say the deadline moves or the dollars go up or something has to give. But, traditionally, there has been no room for that giving. So what you ask, is that for software people the meantime between miracles has to be short. We've pulled off a few, but again, that's unreasonable, and it is resolvable. I do think that it is getting attention at the right level right now because we do have multimillion dollar systems out there that don't work. And the reason they don't work is because of a bug in some tiny little piece of software that no one thought was terribly important. At least, they didn't think it was important enough to get started on early. So, it is getting attention because people are running out of excuses in program reviews.

Mr. Devlin: One of the things that may explain this nonsynchronous flow is that the thrust changes. Another possibility is the fact that the dollars change.

Dr. Martin: With changing missions, usually, the dollars are going to change. As the mission requirements change while the system is under development, it often means some retro-fitting, retro-designing, and that is going to cost more money. So, there is some justification of those things. Experience says that all of the requirements for every system change before the system is deployed from drawing board to deployment. Yet, very little is factored in for the cost change, that we all know is going to be there. How many systems have cost less. None. How many changes in the system have had zero impact on dollars. Probably none.

Lt. Col. Blackledge: Mike Blackledge, Air Force Test & Evaluation Center. The concept of reusable software is a beautiful one, and especially in a large mainframe CDC-type environment. Would you speak a little bit to its applicability in the embedded computer.

Dr. Martin: I don't see why you say it is so different in the large mainframe area. What you are looking at on the host computers, obviously, is an environment in which the tools that are used for the work that is done are very often the same. Editors and some debuggers and that type of thing certainly can be reused. But, how about operating systems for the field, multi-tasking operating systems. I can think of having developed, on six different projects, six different operating systems, the core of which was very, very similar.

Lt. Col. Blackledge: So, you are speaking of reusable software from the support software viewpoint.

Dr. Martin: Yes, but we ought to be able to do some of it in the application area. Fast Fourier transforms and things, routines that you would use in many applications should be packaged in such a way that someone else who has to process the same type of thing can pick it up and use it. Today that is usually not the case. Now, I know from work at the Engineering Experiment Station that we made stabs at cataloging what software had already been developed and making this available to other people. And, yes, there was a lot of ownership, and there was also a lot of "gee, if I do what I think, I'll have a greater confidence that it works".

All that says is that we have a lot of ignorance about qualifying software because you're qualifying software with a prejudice you can't substantiate. Developing tools and objective measures is critical to us in qualifying that software. Once we have something that works, and we can certify that even though Joe developed it, and I don't like Joe, I can certify that it works. That's important. I think that is part of the problem that has surrounded reusability in the past. The other part of the problem that we had was that it was very difficult to understand what somebody had when they were trying to tell you what they had in a piece of software. There is no consistent way of describing what that software routine was. There was a lot of difficulty in understanding how one would take that and incorporate it into another piece of software. So, trying to reuse a routine that somebody else had developed was fraught with sufficient integration problems in just trying to understand what it would take to integrate it that it was easier to start over and reinvent it. All that says is that we have to find a way to describe these things. We have to find a way to catalog them so that somebody else who might want to use them could find them. We have to have a way of describing how they interact with the other components of the program. It is going to take some deep study to understand this, because you are trying to develop a general interface to other people's application systems. There is a lot of research activity that needs to take place. I do think it is a good idea that has not been realized. But I do think there is payoff there and I think that we are smart enough to find out how to make it pay off. But, to say that somebody has the answer today, I don't think they do. But if you give someone a few dollars and the responsibility to get that job done, I think that it can be done. It's a matter of software documentation. Anything else? Thank you very much.

VIEWGRAPHS

USED BY

DR. MARTIN

FOR THE

SOFTWARE TECHNOLOGY PROGRAM

PRESENTATION

SOFTWARE TECHNOLOGY INITIATIVE

Possible Topics

- Reusable Software
- Designing to Test
- Evaluation Methodologies
- Metrics
- Testing Tools
- Prototypes

Mr. Greenlee: Thanks very much, Edie, for your presentation and your eloquent responses to questions and comments.

We've talked a little bit about the ECR DSB initiative and the software technology initiative. One of the other major thrusts, obviously, is the new high-order programming language, Ada, which will pervade the Department of Defense. Ada is a very successful program, I think most will agree, even though cynics said it would never happen back when. As testers, and as developers and other people involved with computer software, we will all have to be aware of and conversant in Ada. For this reason, I've asked Lt. Col. Vance Mall from the Ada JPO to talk to us a little bit about the language and his reflections on how the use of Ada as a high-order language throughout DoD will impact our T&E considerations. Vance.

LT. COL. VANCE MALL: ADA JOINT PROGRAM OFFICE

Lt. Col. Mall: Thank you, Don. I am from the Ada Joint Program Office. This briefly is the way we are constituted. We are attached to the Deputy Under Secretary of Defense for Research and Advanced Technology. We have representation from OSD and three Services. The director is Larry Druffel from OSD. I'm the Air Force representative. This is our purpose (drawn from the AJPO charter): to manage the DoD effort to implement, introduce and provide lifecycle support for Ada.

Now, this is what Ada is in one sentence. The key words are modern high-order computer programming language. Ada embodies most of the programming language concepts that have been developed over the last 15 or 20 years. Ada will become the standard language for writing software for DoD's embedded computer applications. We hold a registered trademark on the name Ada. That has been a very useful mechanism. We exercise the trademark by saying that anyone who uses the name Ada in connection with computer programming languages must acknowledge the trademark and furthermore, anyone who claims to be developing an Ada compiler, must be developing a full Ada compiler (no subsets, no supersets) or he must announce publicly that he intends to complete that development. At the moment, no one has delivered a full Ada compiler. That is reasonable since there hasn't been time, but people who are marketing such software must say, if they use the name Ada, that they intend to develop the full compiler. This turns out to be a very useful loop closing mechanism. There are apparently thousands of people in the land who know this rule and who read ads, and when they spot an ad that fails to recognize the trademark, they call us or send us a copy. We then get in touch with the company and brief him on the situation. It turns out that there are many people who are active in the business who are now at less risk than they were because we were able to close that loop.

This is what we hope Ada and the associated programming support environment will do: Reduce cost and improve quality by facilitating the application of modern software engineering practices. From the beginning, in the mid-70's, the studies all said that the way to solve these horrendous software problems is to apply good software engineering techniques. One of the things you need is a common high-order language. Ada is that common high-order language, but the fact remains that without the modern software engineering practices, we won't get there. So, we always emphasize that it is really software engineering we're after.

The solution to our software problems has two major facilitators: a single modern high-order language (Ada) and the Ada Programming Support Environment.

This is the EIA slide that shows an estimate of the cost that we are talking about. I'm sure that most people have seen this slide. This shows that by 1990, we will be spending $32 billion (1980 dollars) on software for DoD embedded computers. From $2.8 billion in 1980. A tremendous growth, if we don't do something.

And, this is what we are trying to do. The Ada program is considerably more than just a language. This is the view of the world we are trying to create. This represents the host machine on which development takes place. This is the target machine in the aircraft, missile, or other system. These are seen as different pieces of hardware. In the host machine, you have all kinds of development tools. We see these as being candidates for porting from one system to another. Candidates as reusable software. Over here, you actually have the same kind of thing. This is a target machine. It also has an operating system and a runtime support library. This is a computer, and there's all kinds of stuff over there that can be architected similiarly to the way the host is done. It turns out that we haven't done that very well. We are now beginning a program in conjunction with AIRMICS down at Georgia Tech to try to articulate just what this runtime support environment architecture should be, with the hope that we can get some commonality of tools, so that at least some of that software can be transported easily. Those are the two computer based environments. This is the extra-computer environment. The socio-political structure that all of us live in. This is clearly very important, and it includes ... it doesn't include test and evaluation, but, obviously, it should. The language itself (MIL-STD 1815) is in the process of being approved as an ANSI standard. We hope to have the ANSI standard by fall. We have certification procedures for the compilers under development, and in fact, the Ada Compiler Validation Capability is in very good shape. It will consist of about 1500 Ada programs. The scheme is to compile all those 1500 programs on your compiler and, if the compiler does what it is supposed to do, then it is certified correct. The capability also includes a batch of software support tools to make running those 1500 tests automatic and, in addition, a book called the Implementer's Guide which lists pitfalls and suggestions of how we might go about writing the compilers. This whole system has already been very useful. It was in place in large part before work started on compilers, and people who have been writing compilers have been using those tests and the Implementer's Guide. It surely has saved some money and effort, and we hope that it will save more in the future. One of the advantages that that system offers in being available before compilers are ready, is that we avoid having a bunch of compilers which are effectively dialect compilers. The test facility was in place first, so the conversion of compilers to the standard should be much easier.

There are two, perhaps three, bullets on this slide that are of particular interest to this group. We hope that by offering a single language that many people use, and the single programming support environment that many people use, that people will be encouraged to invest in software support technology. High quality tools, such as editors and compilers and debuggers, and a great list of other development tools, including test and evaluation tools. The programming support environment is there to host anything that might be useful in the whole lifecycle management process in software. We hope that by focusing attention, and thus resources, on one language and a small number of programming support environments, that we will be able to get very high quality out of the system. Reusable software is also subject to the same argument.

As has been mentioned, the Ada program seems to be proceeding very successfully, and it is clear that the reason for that is the basically political nature of the program from the beginning. This is a list of the requirements documents which culminated in the Steelman document, which is a 22 page paper describing, in extremely readable terms, just what requirements this language must have. The RFP for the Ada design was based on Steelman. There was a succession of documents circulated, increasingly widely as time went on. Comments were collected on each document and incorporated into the next document. The result is that there are hundreds of people in the world who have contributed significantly to the language. That is excellent from a technical point of view. We collected lots of good technical information. And it's excellent from a political point of view. There are now many, many supporters of the language in industry, government, and academia. I think that accounts for the success of the program.

These are the technical requirements that Steelman lists, and some of these requirements are associated with the T&E effort in that they either solve the problem of errors in software in the first place or they offer a handle to get T&E onto the project. Strong typing turns out to be a method for catching an awful lot of things like typing errors in the program. Encapsulation enforces modularity so that a module which is tested can continue to merit confidence even though some other module has changed. This module is not going to depend on the other module. Machine dependencies are isolated in a properly written Ada program. Machine dependencies are of course a place where breakdowns in reliability as you port the system from one piece of hardware to another are very likely. In a properly done Ada program, those are well isolated so you can focus attention on them. The generic facility is another feature which should contribute to reliability. A generic is a piece of code which can be instantiated with parameters of different types. You can write the generic package once, test it thoroughly, and it should then be usable in more instances than a typical procedure or subroutine is. It is a further effort to factor code into isolated pieces.

In addition to the language itself, the Ada Programming Support Environment is a very important aspect. Its purpose is as stated. One very important thing to be contributed by the Ada Programming Support Environment is a common user interface, common programmer interface. A programmer can move from one installation to another. If they both have the Ada Programming Support Environment, then his interface with the computer will be basically unchanged from place to place, so the training problem is reduced. This is the way the Ada Programming Support Environment is constructed. In here, we have the Kernel Ada Programing Support Environment. One way to describe it is as a wrapper around the hardware or the hardware plus the underlying operating system. It hides that from the tools that are outside. The interface of the tools to the kernel is uniform from one system to another. That means that if you want to transport your Ada Programming Support Environment from one piece of hardware to another, all you have to rewrite is the kernel. This stuff out here does not have to be redeveloped, does not have to be retested, it's simply transportable, at extremely low cost. Free would be nice, but that's probably pressing it.

This next ring is called the Minimal Ada Programming Support Environment. That is the minimal set of tools that you need in order to develop and test programs. It doesn't contain very many tools. Compilers, editors, linker/loaders, and so forth. Necessary and sufficient toolsets. Outside that is the full blown Ada Programming Support Environment with all the other tools anybody can think of, including test and evaluation tools. Now, if this thing works, it should be possible to plug in well-designed, well-exercised test and evaluation tools on any Ada Programming Support Environment irrespective of which hardware it happens to run on.

This is our responsibility in the Ada Joint Program Office. We have these three objectives. The ANSI standard we hope to have in September. We're in the process of getting the first systems up so that by 1983, we hope the Army will be using Ada for production systems. That's their plan. Providing support systems seems to fall into two groups. One is providing lifecycle support for Ada itself, that's configuration control of the language. We anticipate a review at about the 5 year point, say, to see whether everything is fine, to accommodate comments that may have come up and perhaps to adjust the language a little bit. That has already been done, of course, at great length during the ANSI process. So, I would anticipate not too many changes, and certainly none for several years. And to provide support systems such as the Ada Programming Support Environment.

I wrote down several thoughts about where I thought the software test and evaluation project and the Ada program had interfaces, and I think they have been pretty well taken care of already. But, let me go through them again. First of all, it seems to me that the main advantage is the advantage of a standard. Given the Ada program and the extent to which the language and the programming support environment are going to be used, there is very high leverage for anything that is inserted into that program. Test and evaluation stuff that is based on Ada and the programming support environment will have very broad applicability and should enjoy, it seems to me, the fairly sizeable resources that are going to be attracted by Ada. Lots of people are going to be building things in Ada and for Ada, so there are going to be lots of resources. If some of those can be attracted to test and evaluation activity, that's all for the best.

Let me talk a little about this. We are at the very beginning stages of the development of a methodology. Dr. Martin mentioned this earlier. We issued a letter just the other day for publication inviting suggestions for methodologies, inviting them to be sent to Professor Freeman at the University of California at Irvine. The objective is to collect what people know about methodologies and try to sort out that issue. It occurred to me that the test and evaluation community needs to be represented there, so I would encourage you to make that contact with Professor Freeman, and be sure your concerns are represented. Things Dr. Martin was talking about, designing to test and that kind of stuff, are a very important part of the methodology, and we have to be certain that they don't get overlooked. Are there any questions?

Dr. Fischer: Can you tell us please what currently identified DoD programs are planning on using Ada?

Lt. Col. Mall: I cannot. I had that question the other day.

Dr. Fischer: The one that I know of is RTACS, the Realtime Adaptive Control System that the Army is doing for the DCA, but that's the only one that I know of.

Lt. Col. Mall: Another one in the Air Force is the MEECN, Minimum Essential Emergency Communications Network. Ada has been suggested by the contractor and is being evaluated. So, that is another candidate. The Army is pressing very, very hard to get their Ada Language System fielded in early 1983, so that they can deliver it to program managers, but, I don't know what program managers they are talking about.

Dr. Fischer: The Army has the ALS, and the Air Force, the AIE, what are the Navy's plans?

Lt. Col. Mall: The Navy's plans are to choose either the ALS or the AIE, and then build on it. Bob Converse at PMS408 is putting together a Navy strategy, and their basic strategy is to choose one or the other and build on it. Since we have these two efforts, there is a lot of concern. One of the things you want to do with an Ada Programming Support Environment is be able to transport tools from one to the other. Well now, we have two environments, the AIE and the ALS, what are we going to do about that? What we are doing about that is the KAPSE Interface Team (KIT), which is being managed by Trish Oberndorf at Naval Oceans Systems Command in San Diego. There is a tri-Service committee and in addition, there is an industry committee. They are looking at both the ALS and AIE trying to identify interface issues so there can be some commonality of interfaces so that tools can be transported from one system to another.

Dr. Leathrum: You are right in observing that the support for Ada is wide, but there are also some very vocal detractors, partly along the lines of wishing for a smaller language, whatever that means. Do you feel that any of these methodologies might ultimately make the language smaller?

Lt. Col. Mall: Some of this may migrate into hardware. One of the features in Ada is the package. You write a package to provide some services. You define it very well so that it can be ported. It is entirely possible that when one of those packages is well shaken down and extremely useful, that package could be one of the things that is implemented in hardware. If so, it is possible that the language could be simplified, but the issue of simplifying the language has been looked at very, very hard for at least a year, and it turned out that the language is so tightly integrated that you just can't carve. The conclusion was you can't just carve off some pieces because then the house of cards collapses. So, making a smaller language turns out to be very, very difficult to do from a technical point of view.

Unidentified person: The subsetting rule only applies to compilers. The Program Manager can choose his own subset of the language for a given program.

Lt. Col. Mall: That's correct. The subsetting rule has entirely to do with transportability of software, suppression of dialect. But, both from a training point of view and from the individual program manager's point of view if you don't want to use the whole language, you don't have to.

VIEWGRAPHS

USED BY

LT. COL. MALL

FOR THE

ADA

PRESENTATION

Ada®

PROGRAMMING LANGUAGE

®Ada is a Registered Trademark of the Department of Defense (Ada Joint Program Office)

ADA®

ADA IS A MODERN HIGH ORDER COMPUTER PROGRAMMING LANGUAGE WHICH WILL BECOME THE STANDARD LANGUAGE FOR WRITING SOFTWARE FOR DOD EMBEDDED COMPUTER APPLICATIONS.

*ADA IS A TRADEMARK OF THE U.S. DEPARTMENT OF DEFENSE.

ADA

ADA AND PROGRAMMING SUPPORT ENVIRONMENTS WILL HELP US REDUCE THE COST AND IMPROVE THE QUALITY OF SOFTWARE BY FACILITATING THE APPLICATION OF MODERN SOFTWARE ENGINEERING PRACTICES.

ADA JOINT PROGRAM OFFICE (AJPO)

- ATTACHED TO DUSD (ACQUISITION MANAGEMENT)
- REPRESENTATION FROM OSD, NAVY, ARMY, AF
- PURPOSE — TO MANAGE THE DOD EFFORT; TO IMPLEMENT, INTRODUCE, AND PROVIDE LIFE-CYCLE SUPPORT FOR ADA

THE NAME "ADA" HONORS:

AUGUSTA ADA BYRON
COUNTESS OF LOVELACE
1815-1852

- SHE WROTE A DETAILED DESCRIPTION OF AN ALGORITHM TO BE EXECUTED ON CHARLES BABBAGE'S MACHINE. IT WAS, IN EFFECT, THE FIRST COMPUTER PROGRAM.

- SHE WAS THE DAUGHTER OF LORD BYRON.

ADA PROGRAM

□ PROBLEM

— EMBEDDED COMPUTER SYSTEM SOFTWARE

-- LIFE CYCLE COST

-- RELIABILITY

□ SOLUTION

— MODERN SOFTWARE ENGINEERING PRACTICES TWO MAJOR FACILITATORS

-- SINGLE MODERN HIGH ORDER LANGUAGE

-- PROGRAMMING SUPPORT ENVIRONMENT

ADA PROGRAM

DOD INITIATIVES TO IMPLEMENT SOLUTION

- DODD, DODI 5000 SERIES
  - CONSTRAIN LANGUAGE PROLIFERATION
- HOLWG
  - DEVELOP REQUIREMENTS, LANGUAGE, STONEMAN
- AJPO
  - INTRODUCE AND MANAGE LANGUAGE AND ENVIRONMENT
- COMPONENTS
  - USE ADA FOR EMBEDDED SYSTEMS

ADA PROGRAM

OBJECTIVES:

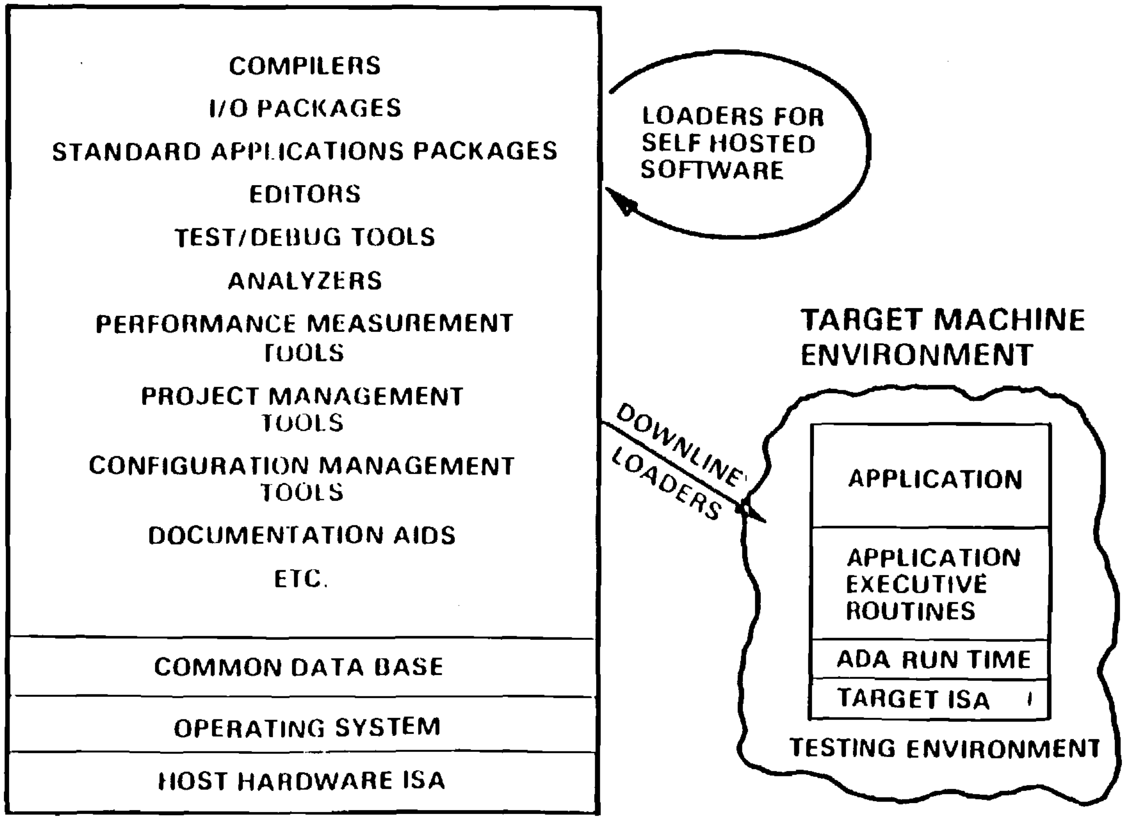
- IMPLEMENT ADA AS A STANDARD
- FOSTER EARLY INTRODUCTION AND ACCEPTANCE
- PROVIDE SUPPORT SYSTEMS

ADA LANGUAGE ENVIRONMENT

ORGANIZATIONAL INFRASTRUCTURE

- LANGUAGE STANDARD
- COMPILER CERTIFICATION PROCEDURES
- STANDARDS & GUIDELINES FOR HOST AND TARGET COMPUTER ENVIRONMENTS
- TOOL DISTRIBUTION MECHANISMS
- DOCUMENTATION
- TRAINING
- COMPUTER RESOURCE POLICIES

HOST (PROGRAMMING) MACHINE ENVIRONMENT



COMPILERS

I/O PACKAGES

STANDARD APPLICATIONS PACKAGES

EDITORS

TEST/DEBUG TOOLS

ANALYZERS

PERFORMANCE MEASUREMENT TOOLS

PROJECT MANAGEMENT TOOLS

CONFIGURATION MANAGEMENT TOOLS

DOCUMENTATION AIDS

ETC.

COMMON DATA BASE

OPERATING SYSTEM

HOST HARDWARE ISA

LOADERS FOR SELF HOSTED SOFTWARE

DOWNLINE LOADERS

TARGET MACHINE ENVIRONMENT

APPLICATION

APPLICATION EXECUTIVE ROUTINES

ADA RUN TIME

TARGET ISA

TESTING ENVIRONMENT

EMBEDDED COMPUTER SYSTEMS

- WEAPON SYSTEMS
- COMMUNICATIONS
- COMMAND AND CONTROL
- AVIONICS
- SIMULATORS

NOT

- FINANCIAL MANAGEMENT, INVENTORY, PAYROLL (COBOL)
- LARGE SCIENTIFIC COMPUTATION (FORTRAN)

EMBEDDED COMPUTER SYSTEMS APPLICATIONS CHARACTERISTICS

- REAL TIME CONSTRAINTS
- AUTOMATIC ERROR RECOVERY
- CONCURRENT CONTROL
- NON-STANDARD INPUT-OUTPUT

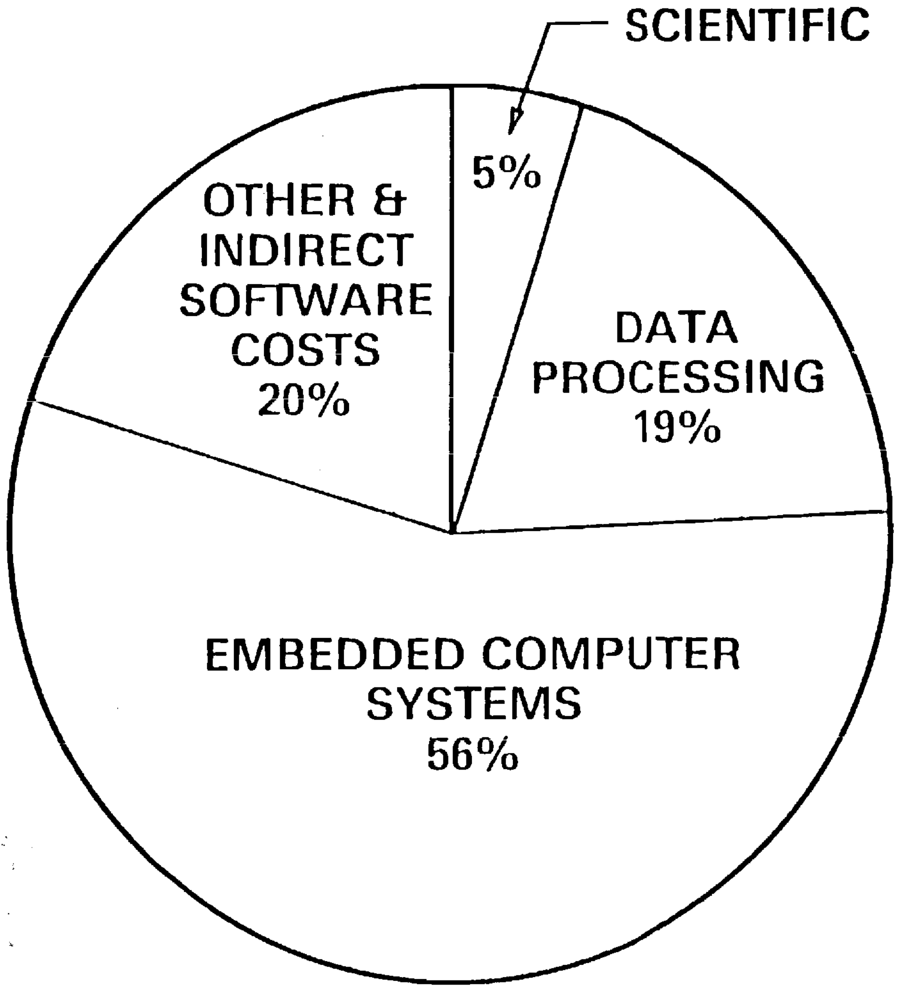
EMBEDDED COMPUTER SYSTEMS SOFTWARE CHARACTERISTICS

□ LARGE

□ LONG LIVED

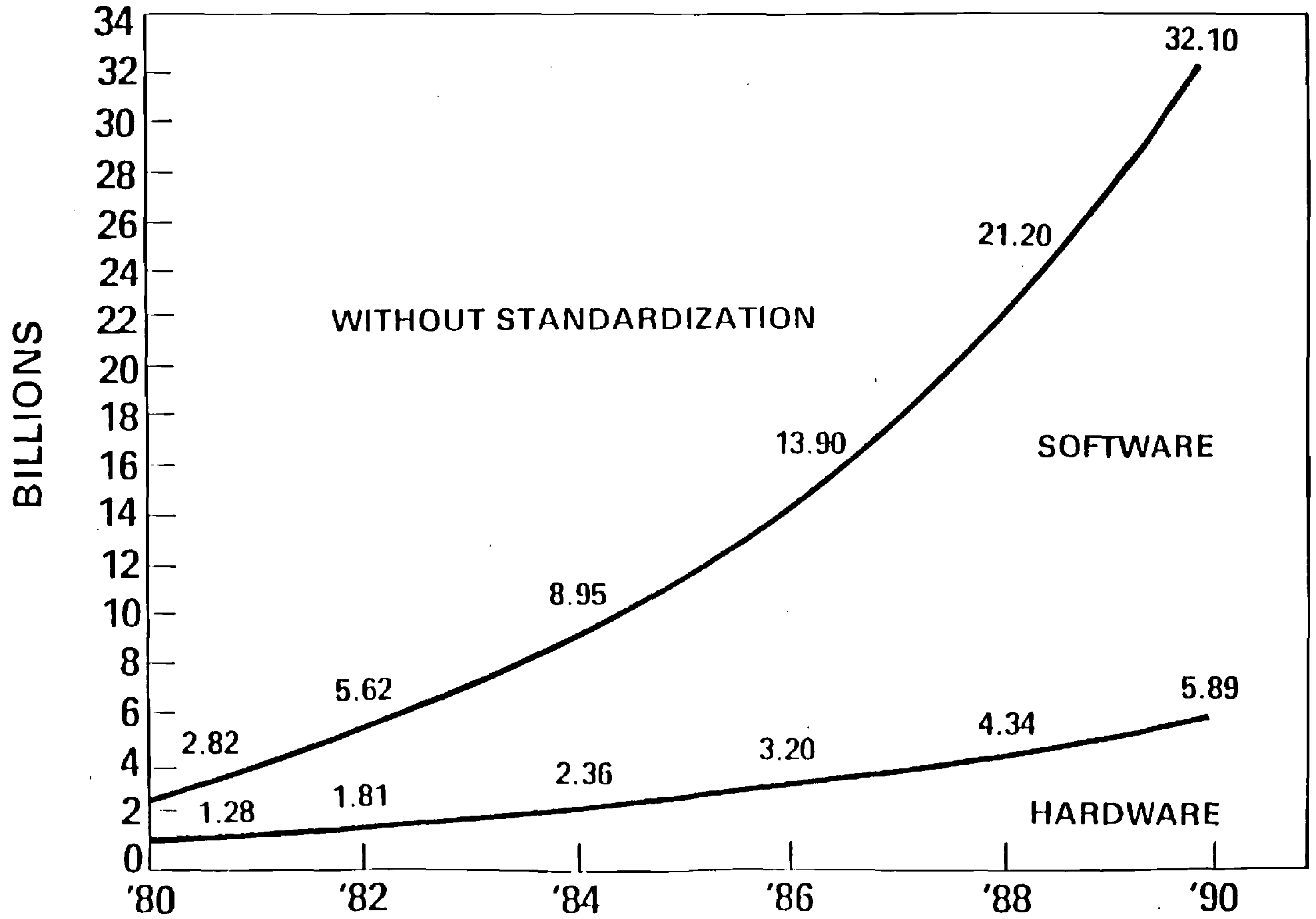
□ CONTINUOUS CHANGE

ESTIMATED SOFTWARE COSTS IN THE DOD
$4.5 BILLION/YEAR



2127 2

DOD EMBEDDED COMPUTER SOFTWARE/HARDWARE (ANNUAL COST — FY 1980 DOLLARS)



SOURCE: ELECTRONIC INDUSTRIES ASSOCIATION.

ADA EXPECTATIONS

- REDUCE COMPUTER SYSTEMS LIFE CYCLE COSTS
- ENCOURAGE INVESTMENT IN SOFTWARE SUPPORT TECHNOLOGY
- IMPROVE ADAPTABILITY OF SOFTWARE PERSONNEL
- ENCOURAGE DEVELOPMENT OF REUSABLE SOFTWARE
- DISCIPLINE SOFTWARE ENGINEERING

STEELMAN REQUIREMENTS

- STRONG TYPING – EXPLICIT DEFINITION AND ENFORCEMENT OF CHARACTERISTICS OF DATA ELEMENTS
- ENCAPSULATION – RESTRICTS VISIBILITY AND USE OF SELECTED VARIABLES; FACILITATES BOTH TOP DOWN DEVELOPMENT AND ACCUMULATION OF REUSABLE MODULES
- GENERIC FACILITY – PROVIDES EXTENSIBILITY TO THE PROGRAMMER WITHOUT EXTENDING THE LANGUAGE
- TASKING – STRUCTURED APPROACH TO CONCURRENT PROCESSING AND INTERPROCESS COMMUNICATION
- EXCEPTION HANDLING – FACILITY FOR DEALING WITH EXCEPTIONAL SITUATIONS WHICH OCCUR DURING PROGRAM EXECUTION
- INTERRUPT HANDLING – FACILITY FOR PROCESSING INTERRUPTS AND OTHER EXTERNAL STIMULAE
- NUMERIC PRECISION – MACHINE INDEPENDENT APPROACH TO INTEGERS, FIXED POINT AND FLOATING POINT
- MACHINE DEPENDENCIES – EXPLICIT DECLARATION AND ENCAPSULATION OF HARDWARE AND OPERATING SYSTEM DEPENDENCIES

APSEs

PURPOSE:

TO SUPPORT THE DEVELOPMENT AND MAINTENANCE OF EMBEDDED COMPUTER SOFTWARE THROUGHOUT ITS LIFE CYCLE

VIEW POINT:

- PROGRAMMING IN THE LARGE
- PEOPLE ARE MORE EXPENSIVE THAN MACHINES
- MACHINES ARE MORE RELIABLE THAN PEOPLE
- OPEN-ENDNESS IS A CRITICAL REQUIREMENT
- THERE ARE TOOL INDEPENDENT GENERIC CAPABILITIES REQUIRED IN ALL INTEGRATED ENVIRONMENTS
- DATA BASE AND KERNAL STANDARD CONVENTIONS ARE NECESSARY FOR TOOL COOPERATION, COMPOSITION

Mr. Greenlee: Thanks very much, Vance. In addition to serving as a site for our meeting, the DSMC has been involved substantively in our software T&E project all along. And this is attributable to Col. Ken Nidiffer. He has made all the arrangements to enable us to have this Workshop here. One of Ken's duties on campus is to direct the course that is presented in management of software acquisition. He's done this through many cycles and through his own thinking process, contacts with industry and the students, who I'd like to mention are not students in the sense of unwashed tenderfeet, but experienced people from industry and the Services who are perhaps, for example, going to undertake some software development assignments in the future, but come in here with their own sets of qualified opinions. Under Ken, the school has developed a perspective on software acquisition, and this relates to testing. So, Ken, I believe, is going to talk to us a little bit about the three aspects, the academic, government and industry, and how they relate to software acquisition and testing in particular. Col. Ken Nidiffer.

LT. COL. KEN NIDIFFER: DEFENSE SYSTEMS MANAGEMENT COLLEGE

Lt. Col. Nidiffer: I've been in the software acquisition business about 20 years. When I started in the early days of our military space efforts, testing was not like we view it today, because we really didn't have a test program. In fact, we were very fortunate to get enough software to load into our satellites such that we could make them work. Today, a well organized test program is a very large factor in what we try to develop. Unfortunately, there is a large amount of controversy on how you develop a well organized test program, and that's why we are here today. To start off a little bit differently, I am going to present my summary conclusions, and then work toward these summary statements.

The first conclusion is that software is an integral part of the acquisition process in terms of its lifecycle, and therefore, software testing must be also. You should consider software testing in the conceptual and demonstration/validation phases, as well as full scale engineering. The second conclusion is that the government will never back away now from its role of defining its requirements. I feel requirements definition is a very important part of our job, and I think this key part of the job will always be with us. The third conclusion which was mentioned by Dr. Martin and several people so far is the need for prototyping. For a long time, industry has prototyped software, but the government has never recognized it within their procurements. The government is beginning to recognize that prototyping is an important function that is being accomplished by industry to meet their needs. The last conclusion, which I think is the most important, is the need for architectures. I think we need an architecture for our systems in which we are evolving.

I will now present how I arrived at these conclusions via historical perspective. Credit is given to Dr. Winston W. Royce of Lockheed Space and Missile Systems Corporation who originally developed some of the concepts which will be presented next.

I first became involved with large scale embedded software systems at the Space and Missile Systems Organization in 1967. In those days, we gave the contractors our software requirements and prayed that they would come back with some code that we could load into the satellites. Although not obvious, there were a lot of advantages to what we did back then.

The first thing, it accented the most important points in terms of productivity as far as an engineer is concerned: analyzing and producing code. He was not tied down in writing documentation. The second thing that was important is that code was produced quickly, and we got a fast response to any change in requirements. We seemed to get more code per unit dollar back then, and that was important in terms of handling our early requirements. There were also a lot of disadvantages. As our systems grew more complicated, we found out that we had to somehow get a methodology by which we laid out our requirements so that we could get them developed. The second thing that happened to us is that we found out that we could not change our requirements very quickly. It took the contractors longer to develop the software because of the complex interfaces associated with the more complex requirements, and they needed more reaction time. In summation, the government and industry needed a communication medium. We borrowed a concept from the hardware people, which was called Baseline Management, a concept of management that had developed out of the Air Force 375 series regulations.

As shown on this chart, we place on the front end of analysis a requirements phase. For example, we started out and required a functional analysis to be accomplished. We required that the functional analysis establish tradeoff studies, and that these tradeoff studies be an iterative recursive process until high quality computer program development specifications (B5/Part I) were finalized, specifications that would define the functional requirements for what we were after. We then permitted the contractor to proceed into design. We kept the coding phase but added something which we are addressing today; we added a formalized testing methodology on the back end of the software acquisition cycle. There are a lot of advantages associated with this approach.

First of all, it was really geared to handling complex requirements. Thanks to Baseline Management, we began to be able to establish more control over the software acquisition process. We also established reviews, such as the systems requirements review, the systems design review, the preliminary design review, the critical design review, reviews that allowed us to review the progress of the contractor. We also formulated MIL-STD 1521, which set up the rules and guides for these reviews. We established some configuration management points. When everything was completed in terms of defining the functions for the software, we developed a functional baseline. Then, when we had allocated all our systems into what we called computer program configuration items, we established an allocated baseline. Finally, we had a product baseline which looked at the as-built product. Because each step fed on the preceding step, we had good communication between the systems engineering that was done by the contractor and our government people, who were reviewing the work. We had another luxury back then also. We had a lot of people within the government who understood systems engineering.

This chart shows some of the weaknesses of Baseline Management. The first thing is that coding and testing are conducted very late in the development cycle. In fact, in a lot of our regulations, a requirement exists that there will be no coding before CDR. Now, there are not too many contractors who ever obeyed this rule, especially if they had to build prototype concept formulation. Yet, the regulations forbid them. Baseline Management also delayed the test program because everything was sequential, so the test program was held at the end. A back end test program leads to other problems. A primary motivating force for shortening a software test program is shortage of program funds and schedule. The program manager is always faced with two critical constraints, dollars and schedule, which he never has enough of. As a result, he robs the back end of his program. By the nature of the beast, the test programs that we have for software are usually too short. Thus, the test program manager has lower confidence in the quality of the software because of the limited number of software paths that are validated. We also produced a lot of paper under Baseline Management. As a result, we often had a thousand line program which had four feet worth of documentation associated with it. Many of the contractors that are out there now know the amount of documentation that we require on software products. As a result of all this paper, we find out it's very easy to misassess progress. Industry is faced with yet another problem which impairs Baseline Management, and that problem is too few resources in terms of systems analysts to do the work and the need for a better systems engineering process based on an evolving technical base. To meet this need, universities and industry jointly came up with a new methodology called Top-Down Design. This methodology is centered on three basic notions. Most of us are software oriented in here, and I'm not going to go over these in detail, but I want to point out where the test fits in. In a hierarchy of functions, we are concerned about the programs that have to be tested in terms of being used for test; they need to be designed first because of a concept we will call test stubbing, which I'll describe in a minute. The second notion of Top-Down Development is that the designer should start to tackle the hardest problems first in somewhat of a logical manner.

The third notion is associated with when the requirement definition, design, code and test should occur. That is, when we identify the module, we can begin to do the requirements definition, the design, and the test early on. We don't have to put that testing until a point late in the acquisition cycle. As a result, through this process, we have allowed testing and design to occur throughout the cycle. Let me provide you with the following scenario. You're a young software engineer who just graduated and got your masters degree in computer science. You are interested in productive labor of designing and producing, not writing hundreds of pages of documentation that will probably never be read. What Top-Down Development allows is for industry to take people who are heavily motivated in computer science and make them productive!

The strengths and weaknesses are presented on this chart. First of all, it does allow industry to be productive with their people. Second, it allows for inherent prototyping, which I think is critical in our systems, especially from the systems viewpoint. Third, testing is accomplished throughout the lifecycle. Top-Down Development has some weaknesses. For example, it is very hard for people to conceive of a top-down design. It takes a very special person or group of people to be able to take a system that they have never seen before and structure a top-down design. A couple of other weaknesses is that if the designers are wrong in their guess, there can be an unravelling in the requirements chain. From the government viewpoint, it is very difficult to set progress review points because there are not logical points in the software development where all the software passes discrete milestone points at one time.

The idea of evolving architectures is not new. It started out, I think, at Space Division way back in 1972. What it essentially says is that the government will have control over the activities that lead to the documentation of its software requirements in the computer program development specification. After a satisfactory allocated baseline has been accomplished, the contractor is allowed to develop products in accordance with the evolving architecture.

I'm recommending here that we examine an evolving architecture concept with respect to DOD 5000.3. Architecures are currently being set up by the Services, both planned and unplanned. For example, the Department of the Army has recently released a Post-Deployment Software Support (PDSS) Concept Plan For Battlefield Automated Systems, and TRW has produced for Air Force Logistics Command a Long Range Plan for Embedded Computer Systems Support which take into account some of these evolving architectures for deployed systems. I think what we are going to see in our structure, especially as Ada comes along, is a point where we can get these incremental releases for our weapons systems. Whether those occur under the same contract or whether those releases are competitively bid in a fixed price structure, I don't know. But, I do see it evolving. Thank you very much.

Mr. Devlin: As a tester, how does one test an incremental release of weapon system software, especially when major change requires retesting to conform to directives for issuance approval for service use?

Lt. Col. Nidiffer: I think our thoughts on testing in terms of the development are going to have to change, and the reason for having to change is because I feel industry is going to stay with top-down structured programming based on the need for better productivity from its limited work force. Based on this management approach by industry, we have to look at how to get better visibility. I don't have all of the answers on how to achieve this visibility. I'm sort of hoping that this group will come up with some of those answers.

Mr. Devlin: I agree with you. You might consider bouncing this off the Navy. The Navy's been involved with incremental releases for some time. There's been a number of problems though, i.e., training of the operators in incremental releases, funding, documentation, user/operator manuals, in general, total configuration control and quality assurance.

Lt. Col. Nidiffer: I think that's super, and I share it with you. I guess what I'm saying is that from what I've seen, that's where we're headed. I don't know all the answers on how to get there. But, I have some ideas.

VIEWGRAPHS

USED BY

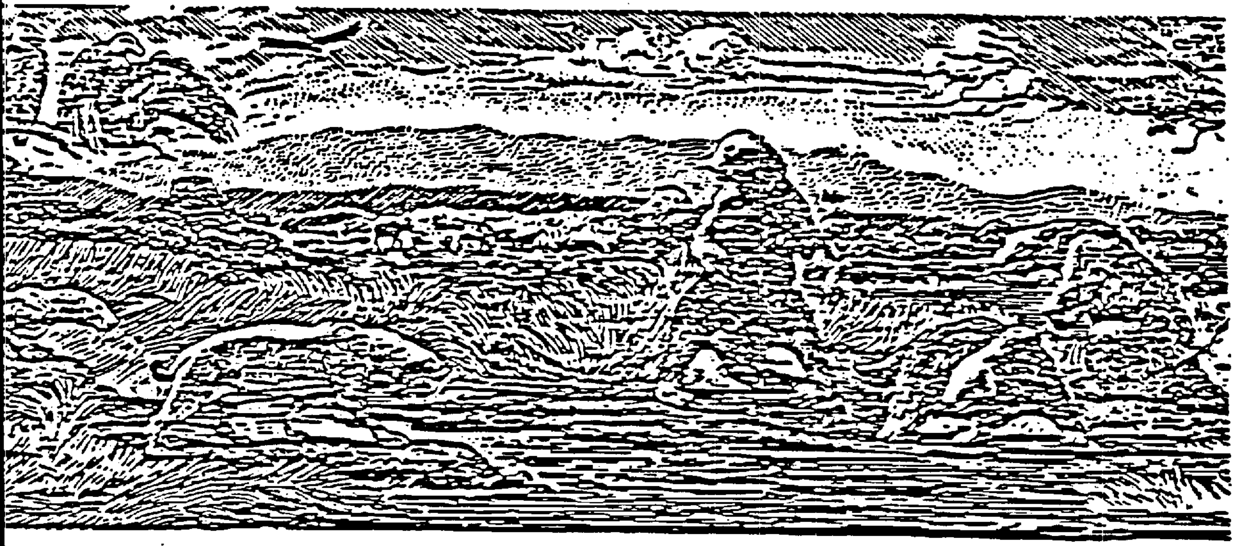
LT. COL. NIDIFFER

FOR THE

DSMC PERSPECTIVE

PRESENTATION

ACQUISITION MANAGEMENT OF COMPUTER SOFTWARE



Lt Col Kenneth E. Nidiffer
Defense Systems Management College
Building 207/Room 227
(703) 664-3477
Autovon 354-3477

SUMMARY

0 SOFTWARE TESTING IS AN INTEGRAL PART OF THE ACQUISITION PROCESS AND WILL NOT BE SEPARATED FROM IT

0 GOVERNMENT PROCUREMENTS WILL CONTINUE TO REQUIRE THAT REQUIREMENTS AND SPECIFICATIONS BE TRACEABLE

0 SOFTWARE PROTOTYPES WILL GAIN FAVOR

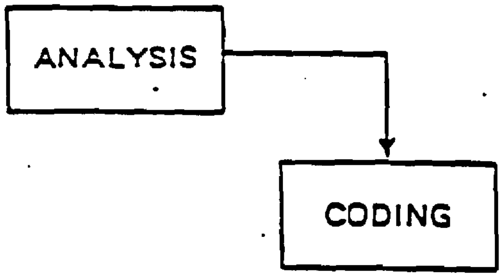
0 SYSTEM ARCHITECTURE STUDIES WITH TIME-PHASED GOALS WILL EVOLVE

ALTERNATIVE I:	ANALYZE AND CODE
ALTERNATIVE II:	BASELINE MANAGEMENT
ALTERNATIVE III:	TOP DØWN DEVELOPMENT
ALTERNATIVE IV:	INCREMENTAL RELEASE

COURTESY DR. WISTON W. ROYCE
LOCKHEED

ANALYZE AND CODE

(1959 - 1965)



ANALYSIS

CODING

120.28.2339

ANALYZE AND CODE

STRENGTHS

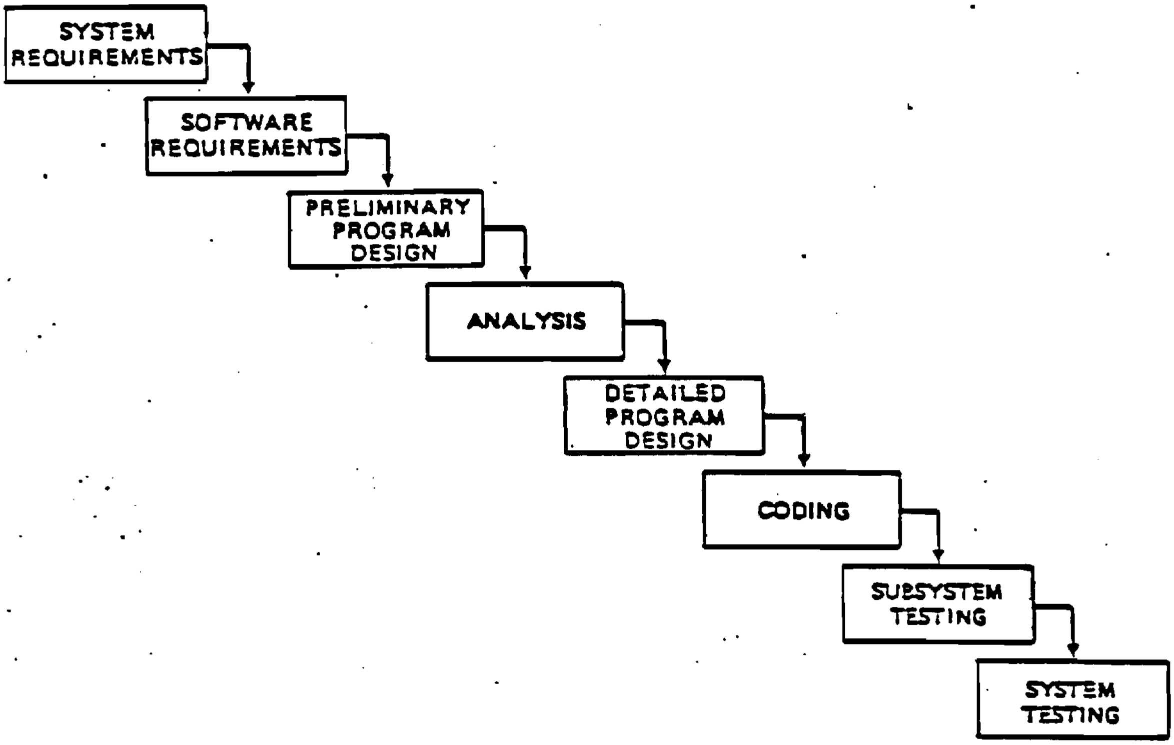
- EMPHASIS ON TWO MOST PRODUCTIVE STEPS
- TAILORED TO PERSONNEL PREFERENCES
- LOW COST, QUICK SCHEDULE RESPONSE IF PRODUCT IS ACCEPTABLE

WEAKNESSES

- NO REQUIREMENTS ANALYSIS
- NO BUYER-USER INVOLVEMENT
- NO PLANNING OR AUDIT
- HIGHLY DEPENDENT ON SKILL

BASELINE MANAGEMENT

CURRENT BEST PRACTICE (1965 - PRESENT)
AN EIGHT STEP PROCESS



SYSTEM REQUIREMENTS

SOFTWARE REQUIREMENTS

PRELIMINARY PROGRAM DESIGN

ANALYSIS

DETAILED PROGRAM DESIGN

CODING

SUBSYSTEM TESTING

SYSTEM TESTING

120.28.2339

BASELINE MANAGEMENT METHODOLOGY

STRENGTHS

- GEARED TO REQUIREMENTS ALLOCATION FOR LARGE HIGH TECHNOLOGY APPLICATIONS
- PERMITS BUYER/USER INVOLVEMENT
- PERMITS EARLIEST IMPOSITION OF CONFIGURATION CONTROLS
- PRECEDING STEP SERVES AS AN APPROVED, DOCUMENTED BASELINE FOR SUCCEEDING STEP

WEAKNESSES

- DELAYS DESIGN, CODING AND TEST
- TOO MUCH PRODUCTION OF LITTLE USED DOCUMENTATION
- TOO CONCENTRATED TEST PROGRAM
- FAIRLY EASY TO MISASSESS PROGRESS

TOP-DOWN DEVELOPMENT

(1972 - PRESENT)

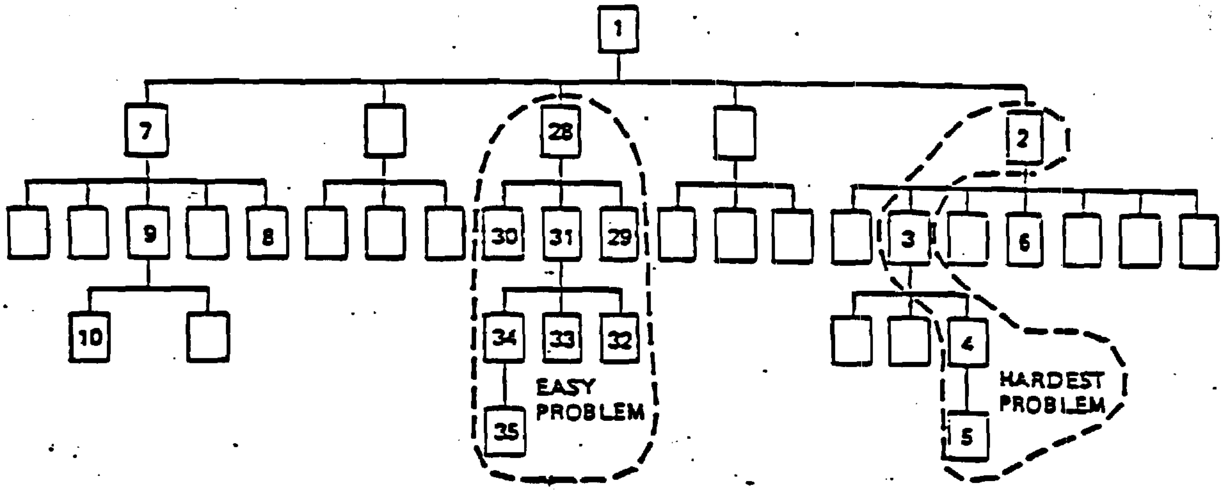
(1) ALIGN THE SOFTWARE FUNCTIONS INTO A HIERARCHY

- CONTROL ORIENTED: CALLING ROUTINES ARE HIGHER THAN CALLED ROUTINES
- DATA ORIENTED: ROUTINES THAT PRIMARILY SET DATA ARE HIGHER THAN ROUTINES THAT PRIMARILY USE DATA
- REQUIREMENTS ORIENTED: ROUTINES LEAST SENSITIVE TO REQUIREMENTS OR DESIGN CHOICES ARE BEST RANKED HIGHER THAN MORE SENSITIVE ROUTINES
- TEST ORIENTED: ROUTINES CRITICALLY NEEDED FOR TESTING OF OTHER ROUTINES ARE HIGHER RANKED THAN ROUTINES NOT NEEDED FOR TESTING

TOP DOWN DEVELOPMENT

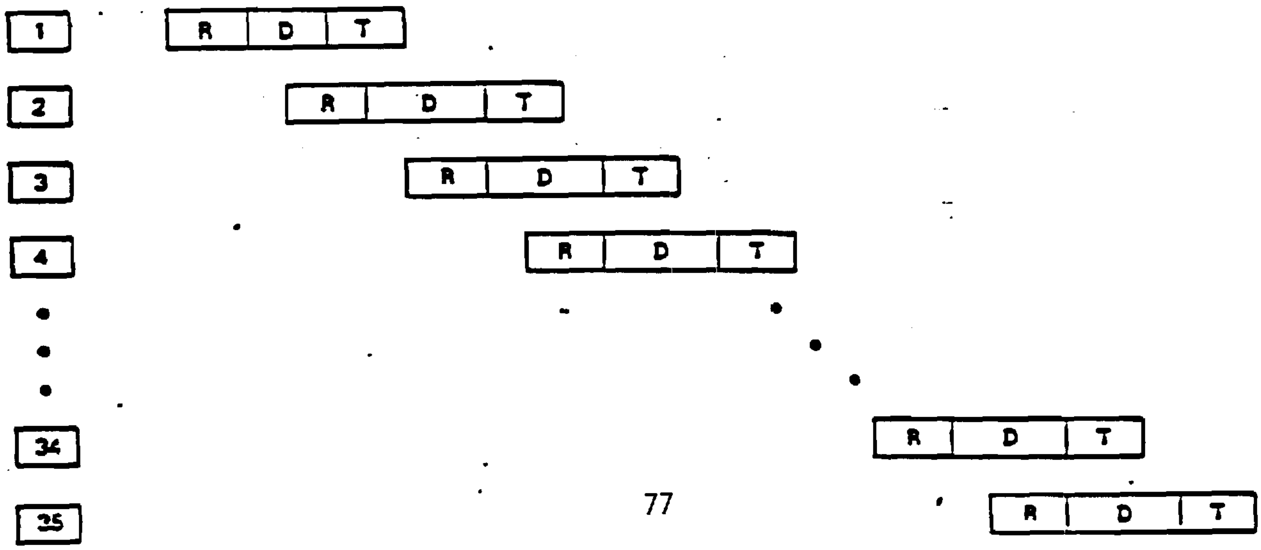
(2) IDENTIFY THE ORDER OF DEVELOPMENT SUCH THAT:

- WHEN DEVELOPING A ROUTINE ALL HIGHER ROUTINES ARE COMPLETED
- HARDER PROBLEMS ARE ATTACKED FIRST; EASIER PROBLEMS ARE DELAYED
- TEST STUBBING IS EASIEST



TOP DOWN DEVELOPMENT

(3) STARTING WITH THE TOP FUNCTION DO THE COMPLETE BASELINE MANAGEMENT LIFE CYCLE - REQUIREMENTS, DESIGN, TEST - BEFORE PROCEEDING TO LOWER LEVEL FUNCTIONS. CONTINUE UNTIL COMPLETE.



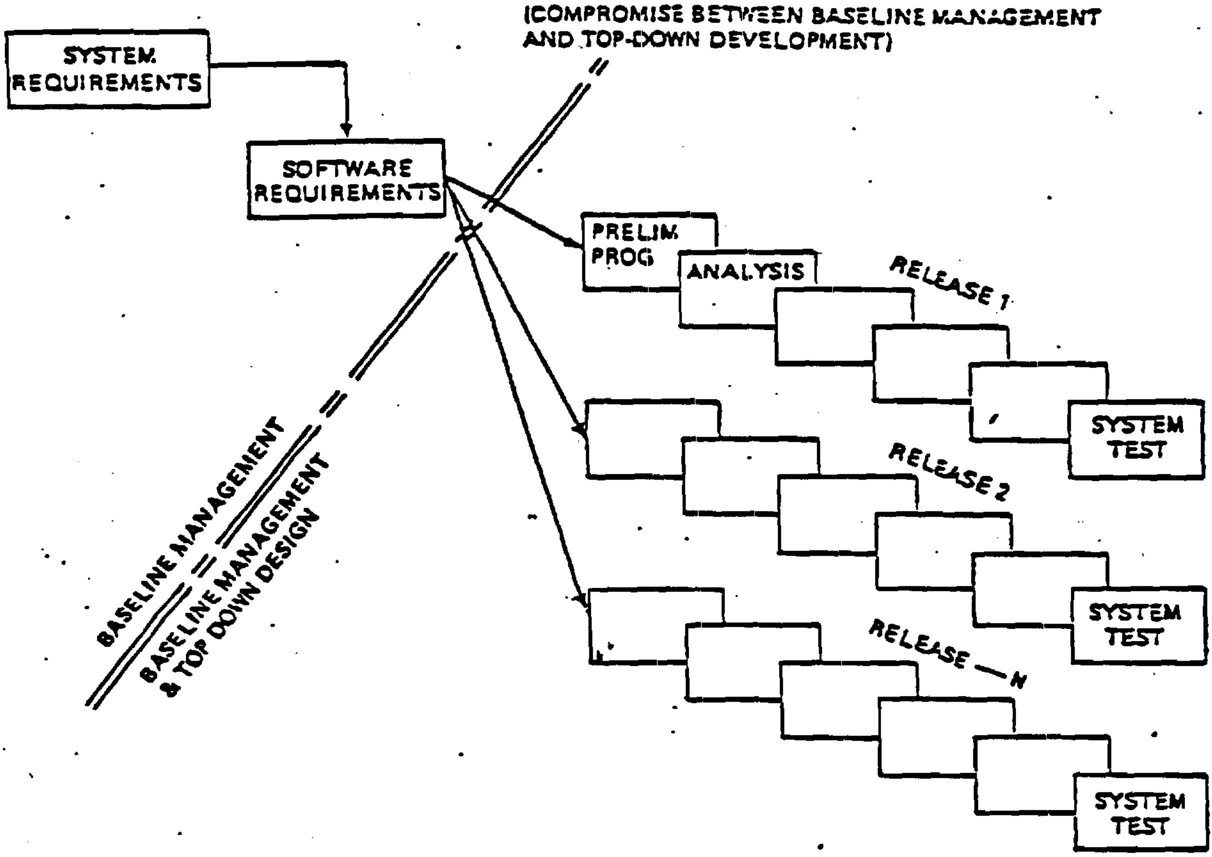
STRENGTHS

- EARLY DESIGN, CODING AND TEST
- CONCENTRATION ON CRITICAL DESIGN ELEMENTS
- INHERENT PROTOTYPING
- PROGRESSIVE DOWNWARD FREEZING OF TOP-LEVEL CONTROL, DATA, DESIGN AND TEST INTERFACES
- TESTING IS SPREAD THROUGHOUT THE LIFE CYCLE

WEAKNESSES

- FRAGMENTED REQUIREMENTS ANALYSIS
- HARD TO IDENTIFY TOP OF DESIGN AND MOST NATURAL ORDER OF PRODUCTION
- MOST PEOPLE THINK BOTTOM UP
- CONSTANT RISK OF UNRAVELING DUE TO UPWARD BREAKAGE
- TEST STUBBING IS HARD
- TESTING TO REQUIREMENTS IS HARD
- PUZZLING DESIGN REVIEWS BECAUSE THERE ARE NO NATURAL REVIEW POINTS
- COMPLEX CONFIGURATION MANAGEMENT

INCREMENTAL RELEASE



.28.2339

Mr. Greenlee: Thanks very much, Ken, for that presentation. That was excellent and very much to the point.

Our next speaker will present an industry perspective. As we've commented before, the vital participation of industry in this project is welcomed. The meeting today is not filled with representatives from industry because it was not intended to be that way. The National Security Industrial Association has kindly offered to serve as a conduit for industry opinions, perspectives, biases, complaints, and whatever, and will play a greater role as the project continues. Today, the NSIA and industry are represented by Dr. Kurt Fischer from CSC and Mr. Jack Devlin from Vitro. I would like to introduce Dr. Fischer, who will talk a little bit about the industry point of view on software testing.

DR. KURT FISCHER: NATIONAL SECURITY INDUSTRIAL ASSOCIATION

Dr. Fischer: A couple of introductory remarks with regard to who and what the National Security Industrial Association is. First of all, we are glad to be invited and to participate in your effort, Don. We were founded by the first Secretary of Defense, Secretary Forrestal, back in the very early 1950's, although I can't remember the exact year. Our objective is not to be a lobbyist group, but to work together with the agencies of the US Government that do business in the area of national security to help solve common problems. We are made up of about 300 companies, and we work with, as I said, the various agencies of the government, not just DoD, though DoD is the largest agency of the government in terms of budget and number of people that solve our national security problems, but we also are active with the Department of Energy and NASA. So, with that, let me also say that it is very difficult to present a total industry perspective in 20-25 minutes, though we did canvass about a dozen different companies. I would like to say that this is a really unofficial industry canvass, I'm very happy to be here, and the views here are not parochial to Kurt Fischer, but if we were to do a full job, we would probably require some type of workshop with 50 or 100 people.

I'm going to talk about 8 or 9 problems in the area of T&E that we in industry have found. And, I'm going to recommend some solution approaches to each of these areas.

First of all is the area of planning. We in industry see a lack of early planning in the area of T&E, specifically with regard to scope. We see a lack of planning T&E resources, frequently. We see a lack of testable requirements. I've seen countless numbers of specifications at the B5 level, where the requirements are not testable. For example, they talk in terms of adequate response time. However, the testable requirements that I find most insufficient are those specifically dealing with performance requirements. Functional requirements we seem to have more of a handle on, but performance requirements seem to allude us. We also find inadequacy in the areas of simulation/stimulation tools. However, we have some recommended solutions. First of all, we need earlier and more thorough test planning and analysis. One method to do this could be through modification of DoDD 5000.3. Another recommended solution is to make our test organizations independent from our project organizations. While we do that at the operational test level, we don't do a very good job of that at the development test level nor from the contractor point of view, we don't always do that in our own test and evaluation.

Another test problem area that we see is something that we will call "tail end Charlie." That is, there is a tendency to shortcut testing due to the budget and the schedule pressures. This forces incomplete testing, it forces testing to obsolete requirements, due to late changes to those requirements. Frequently, we have compromised IV&V efforts. Sometimes, we have inadequate handling of our software problem reports. So, how can we solve this problem? One problem addressed by the speaker previous to me was to develop a "build a little, test a little" environment. We should issue our software or systems in terms of releases or, what we at my company call them, builds.

Another solution could be to assign the IV&V responsibility, its funds and its schedules to the operational testing and evaluation agencies. OT seems to get involved too late. They know they do, the little influence they do have on the frontend is a little influence. If the IV&V power were embodied to them, they could have information continually throughout the development phase.

Mr. Watt: Some of the mechanizations particularly relative to the operational test agencies having more responsibility is one which we have thought about considerably and discussed rather extensively with the Services. Invariably though, we get into resource constraint problems. Operational test communities only have a certain amount of resources that they can apply, and they are very hesitant and reluctant to get back into the earlier phase of the acquisition cycle, i.e., during the development part of the project. So that's the dilemma we find ourselves somewhat in with them taking on more of the IV&V kind of responsibilities. However, I feel that somehow, we're forcing that issue more and more as each weapon system comes downstream and comes on line. You'll find that more and more an issue that we have to grapple with, and consequently, invariably we're driving the operational test community back further and further into the cycle. I see some of the guys back there shaking their heads, and I had a number of comments along that line. In the space business, particularly, one way that has been accomplished is actually by going out and contracting for independent tests, which says I buy resources to do that job. General Henry's shop did some of that with the shuttle, which was not so unusual although that was a rather significant effort for which he contracted directly the IV&V. So, I throw that out as a fundamental that is real world to us. We've been grappling with how to do it, how to get the operational testers more involved. I could go on with some other examples ... right now with some of the V&V work that we are doing, we are now looking at how do we perhaps contract out that independent test function, particularly during the early phase. There are disadvantages; there are advantages; we can solve the resource problem by building others. You may want to hold that as the kind of thing that is not directly related to this discussion of testing, it is related to the resource limitations and how to get the job done.

Dr. Fischer: Is your issue that the OT agencies do not want to get involved or that they don't have the funds to get involved?

Mr. Watt: It's not the funding resources, it's a talent - resource people problem. Although everything relates to funds, but that's not our primary concern. Our primary concern with these operational test activities, and we've had some long discussions along these lines, is that I see it moving more and more by necessity that these guys are having to get closer and closer up front. The test agencies have got to get involved up front. The pressures are there, but that's where the problems are - people.

Dr. Fischer: It seems to many of us that, from a contractor's point of view, if the OT agencies don't follow through what's going on during the development phase, as they get pushed further and further back, their impact on the program is going to be more and more diminished. I would appreciate hearing from the OT agencies this afternoon or even now.

Lt. Col. Blackledge: I'll mention that in the Air Force we do try to get on board early. The only influence that we have is that we do things like review the request for proposals and statements of work, and we try to make sure that the software concerns are addressed early on. If there is going to be an IV&V contractor, we try to put some hooks in the contract to allow that IV&V contractor to work with us. The AFTEC position is that we do not want to get into the monitoring of the IV&V. The reason is that we simply do not have the resources to do that. What we want to do is encourage a hybrid team of IV&V contractors working with, in our case, Air Force personnel - the user type people - the people who are going to be maintaining the software, get them as part of that IV&V team, use those kind of resources because they are the people that are going to use that system.

Cdr. Anderson: I have a couple of comments. You mentioned early involvement. We at OPTEVFOR all try to get involved early. I don't know if the rest of you have this problem or not. The Program Office attitude is "get out of my hair, OPTEVFOR, I'll see you later." We fight that syndrome all of the time. "I'm building the system, and you're going to test it, and I don't want to see you until it is time for OPEVAL." So, in the Navy side of the house, we are fighting the Program Manager, too. As far as early involvement, we'll be there, and we'll be attempting to influence the program development. However, OPTEVFOR is not chartered with that kind of requirement. We are there to see how it is being built so that when it comes time to test it, we can test it adequately.

Dr. Fischer: Can you set test requirements?

Cdr. Anderson: I interpret the operational requirements document to establish the criteria that must be met and the questions that should be answered by operational tests. As you pointed out, when you have an operational requirement that a system X works better than a system Y, what kind of a test requirement is that? You have got to have a good requirement. Its got to work three times faster than Y ... Ten times better than Y. You have got to have some kind of number in there.

Dr. Fischer: That's right, it has got to be quantitative.

Cdr. Anderson: As far as OPTEVFOR involvement in the IV&V early, we run across the same problem as AFTEC, we don't have the people to do that. We don't really have the people to do the job we are doing today.

Mr. French: As far as OTEA is concerned, in my briefing to be given later this afternoon, I don't know how many times I used the word "early", but it has got to be 25 or 30 times. I've got a contradiction on the other side ... the people who are going to be doing this early involvement at OTEA. Right now, there are 3, and we expect to expand up to 5 or 6. There is no way we can manage to do all of the early involvement that we all recognize as important with 5 or 6 people. So where the manpower is going to come from and how to start off small like we are doing and justify our existence to the point where we can grow to a useful size is the question. Right now, it is trial and error. We have to justify the early involvement and identify its value so that we can put the people and resources there, and then maybe we can expand. But, right now, we just can't justify it.

Dr. Fischer: I guess that is the problem. It is the resources.

Mr. Watt: Maybe we can go into that more this afternoon. The TEMP that you mentioned there is the proper vehicle, the Test and Evaluation Master Plan. We see it being utilized more. It is not nearly to the point where it should be, but it is an effective tool.

Mr. Devlin: As a straight man for industry, I would like to ask you the question ... Where would you like to see operational test organizations involvement? At what period would industry like to see that?

Dr. Fischer: Industry usually doesn't get involved in the early concept formulation phase. We don't get involved until a year or two has gone on within the Departments. So, I don't know that our position is near as critical as the position of the customer, because we are kind of the middle guy, it's the customer that defines the system and the customer really has to end up testing it and accepting it and using it. I see fewer problems the earlier the testing agency gets involved. I see fewer conflicts, both technical and political/managerial, the earlier that the testing agencies get involved. For another industry perspective, can you answer your own question?

Mr. Devlin: I'm caught in the middle. I'm working for an OT&E agency and industry at the same time. I could philosophize for days.

Mr. Watt: Let me make one statement before we go on. The new policy that requires that we make a commitment for production at a Milestone II which says we are now coming in to a point where normally we would say, "OK, we are ready to go to the next phase ... we haven't finished our review, but at least we have made enough of a test, enough of a hard look at the system that we can go on." Now when we go into a Milestone II review, we must say, "yes, we are committed." This says that we must get earlier involvement of the testing community, we have got to be involved right up front.

Dr. Fischer: Another problem area is that extreme testing requirements are levied without regard to the impact of software failures. The testing requirements and cost for noncritical software frequently is the same as that for highly critical software, and it doesn't necessarily have to be that way. Someone mentioned earlier today that we ought to do an analysis of the cost of testing versus the cost of failure. The people in industry who we surveyed wholeheartedly agree with that. The testing cost should be a function of the cost of the particular software failure. Some candidate recommendations are that we ought to establish some software criticality factors with graduated test requirements. For example, at the highest level you might have life critical software or life critical subsystems. The next level might be subsystems or software that has implications, strong implications in the area of national security. Lower than that, you may have ground support software, post processing software, test support software and other types of support software that may not require years of test cost.

Another problem area is in the area of test organizations. Software developers frequently have too much involvement in testing. Frequently, our software designers move into testing as the design phases down and the testing phases up. This allows them to, so to speak, grade their own papers, which does not meet our independence objectives. What happens is the designers tend to test the design rather than test the requirements. So, some recommended solutions may be to have testing in a separate organization from your development organization. Sometimes we do that. But, then again, sometimes, we don't. Also, we ought to discourage transferring the designing and programming personnel from the development shops into the test organization within a given project. This is not to say that designers should never be testers and vice versa, but they should not be testers on the projects that they designed.

In the area of test management, we see an undisciplined application of the software development process and especially configuration management. We find that our software baseline is not adequately fixed or defined. Many of our designers have taken courses in fuzzy set theory, I'm afraid, and frequently, there is a premature declaration of ready-to-test software, i.e., testing is initiated by schedule, not by whether or not the software is fully developed. Frequently also, we see an incorrect specification of software interfaces and the lack of quality software documentation. So, what can we do about that? Well, we can increase our commitment to formality with more configuration management, more test control, better, not necessarily more, but better documentation, and a stronger understanding of organizations. We should establish some useability and tracking mechanisms for baselining our status, for understanding and tracking our design steps, for identification and definition of interfaces, and for tracking our software problem reports.

We also see problems in the area of requirements definition. The software community really has not yet learned how to write good requirements specifications. They are still incomplete, inconsistent, unclear, and quite untestable. What can we do about that? Well, I think we need a more structured approach to requirements definition and analysis that focuses requirements information to the designers. In our R&D communities, we have developed several techniques. We have a technique called problem statement language/problem statement analyzers, PSL/PSA, that we have used in a limited environment. There is a methodology developed by the Army called the Software Requirements Engineering Methodology, SREM, that we have used in various systems. We ought to take this knowledge that we've used in the R&D lab, and bring it into the production environment. There is a wide gap between what the R&D'ers know and what the software production people in the trenches know. Somehow, we have to do a better job of taking the technology and transferring it from the lab into the production trenches.

There is one inherent characteristic of the beast, and that is that many of our systems are extremely complex. From a testing point of view, it's virtually impossible to test all system level combinations. Moreover, we have very few tools to help us do that. When we try to test all the paths through the code, we usually do that at the module level. Unfortunately, however, even when we integrate thoroughly tested modules, you still have not tested all the system level paths. So, we're still finding that we need better techniques to help us test systems. How can we do that? Well, one method is to use more simulation techniques. We ought to go through some type of sampling process, where we sample our software testing to verify the range of input (the stimulation) of our particular software system. We could develop automatic test generators. Again, in the labs and in our R&D centers, we have developed some very useful automatic test generators, but we in the software engineering community and management community have done a very poor job of taking that technology from our R&D centers and really implementing it on our production programs.

Another problem area is regression testing. The problem here is that minor software changes can require extensive retesting. I was at NORAD about a year ago and spoke with Col. Phil Deering, who is the director of the Space Computation Center, and he told me that his number one problem was trying to select test cases to re-run after they had made changes to their software modules. It's a real problem in the operations and maintenance organizations. How do you retest your software? It's very difficult to determine the logic and data dependencies that flow or ripple through after you have made a change to a piece of software. We have no tools or very few tools available to help us do that. One problem in the area of acquisition management is what few tools we have during the development are rarely transported to or given to the maintenance environment to help those people to maintain the system. So, the problem is two-fold. Number one, it is a difficult problem, and we have few tools to help us solve it. Number two, what few tools we have in the development shop are rarely or never transported over to the maintenance shop, and they're the people who need those tools to help them do their regression testing.

So, what can we do about that? First of all, we should pay closer attention to test requirements during design. We should enforce a structured design, partitioning of functions to isolate the impact of a particular change, and we should develop better software tools to assist designers and testers in cross referencing. Some of our projects have good traceability matrices. Other projects have no idea what the word traceability means. We ought to do a better job of enforcing the requirement of having traceability. That would help a great deal when it comes time to change your requirements.

I have addressed the issue of software tools in many of the previous viewgraphs. Our software toolbox, industry's software toolbox, has suffered due to a lack of investment. Partially, that's industry's fault for not coughing up the necessary IR&D funds to build such a toolbox. But, I've also seen statements of work coming out of the Department of Defense that say, no tools shall be developed under this contract. Another problem is that some tools that we in industry wave in front of our customers do not always perform to the level of adequacy or accuracy to which the salesman has portrayed them. I've seen many cases in industry where people have said they've got this tool that does that, or that tool that does this, great tools. But when you really pin them down, "where have you used this?", "let's see your documentation", "can I use it on this project?", well the industry representative suddenly becomes more fuzzy. Another problem is that many of our embedded computer systems are still coded in assembly language, yet most of our tools are geared toward the higher order languages. So, in some of our systems, we have a problem.

So what can we do? Well, I'm a firm believer in the programming environments. I believe, and so do the people we talked to when we surveyed the members of NSIA, that we need to structure an integrated software tool development environment. We have made much progress in this area, e.g., the Ada Integrated Environment that the Air Force has and the Ada Language System that the Army has. We will need to put more tools in the integrated environment, and I hope that those tools will be centrally available from some source so that when I go on a project and I know I need tools, I know I can write to somebody and get a set of tools to put in my Ada Integrated Environment. Secondly, we need to use, we are beginning to use, more higher order languages for embedded software. We need to standardize on that yet, and we are doing that through Ada.

Maj. Hammond: I noticed that you and several other previous speakers were talking about designing for testability, which is a concept that we fully support also. It seems to us that synchronous systems that operate on a fixed clock cycle are much more testable than asynchronous systems. So much so, that we are considering banning the use of asynchronous systems for things that have extreme reliability requirements, like nuclear criteria, safety of flight, etc. The only problem we see with that is that it appears to be counter to the current thrust in computer science and also would throw away much of the advantage that Ada gives with its built-in tasking. That makes us a little reluctant to put that ban on. Do you have any comment on that from the industry viewpoint?

Dr. Fischer: I have a boss that did his dissertation in the control of asynchronous systems back in the very early 70's, and he gets so angry when he sees systems that are forced either to go into synchronous designs or that are misdesigned in an asynchronous mode. It's a complex issue, and it's one that we haven't solved though he and others believe it is solvable if we could just get to it. And, if we could just get somebody to fund this work and get it out in the open literature, if you will. RADC has several efforts going in the area of distributed control. Perhaps, out of their research, an answer will come. Certainly, I was not aware of your initiative, if you will, to ban asynchronous systems.

Maj. Hammond: I would not dignify it with the title "initiative." Right now it's just exploratory thinking on the part of me and my boss.

Dr. Fischer: That's certainly one alternative. But, is that the right one? I think we ought to solve the problem. Now, to really answer your question, I can't, I don't know how, I don't have any information, I don't have any industry-wide perspective to solve the problem. It's a technical problem, and I think we probably will solve that within the next half a dozen years. Now, from a T&E point of view, I do know that it is a very difficult problem to test. On one major weapon system program, they just increased the flight test period from something like 12 months to 27 months, because of that very issue. All I can say now is that I acknowledge that it is a difficult problem, but I know there are some areas in which they are trying to solve it. Anything else?

VIEWGRAPHS

USED BY

DR. FISCHER

FOR THE

INDUSTRY PERSPECTIVE

PRESENTATION

SOFTWARE AD HOC GROUP

of the NATIONAL SECURITY INDUSTRIAL ASSOCIATION

ISSUES IN

TEST AND EVALUATION

TEST PLANNING

- PROBLEM: LACK OF EARLY EMPHASIS ON T&E RESULTS IN BELATED RECOGNITION OF SCOPE
- RECOMMENDED SOLUTIONS
  - -EARLY AND THOROUGH TEST PLANNING AND ANALYSIS
  - -MAKE TEST ORGANIZATIONS SEPARATE FROM PROJECT OFFICES

TAIL END CHARLIE

- PROBLEM: TENDENCY TO SHORT CUT TESTING DUE TO BUDGET AND SCHEDULE PRESSURES

- RECOMMENDED SOLUTIONS

  -BUILD-A-LITTLE/TEST-A-LITTLE

  -ASSIGN IV&V RESPONSIBILITY, FUNDS, AND SCHEDULES TO OT AGENCIES

COST EFFECTIVE TESTING

- PROBLEM: EXTREME TEST REQUIREMENTS LEVIED WITHOUT REGARD FOR IMPACT OF SOFTWARE FAILURES

- RECOMMENDED SOLUTIONS

  -ESTABLISH SOFTWARE CRITICALITY CATEGORIES WITH GRADUATED TEST REQUIREMENTS

  -REQUIRE ECONOMIC ANALYSIS - COST OF TESTING VS. COST OF FAILURES

TEST ORGANIZATION

- PROBLEM: SOFTWARE DEVELOPERS HAVE TOO MUCH INVOLVEMENT IN TESTING

- RECOMMENDED SOLUTIONS

  -TESTING MUST BE DIFFERENT ORGANIZATION

  -DON'T TRANSFER DESIGNERS/DEVELOPERS TO TEST ORGANIZATION

TEST MANAGEMENT

- PROBLEM: UNDICIPLINED APPLICATION OF CM AND S/W DEVELOPMENT PROCESS

- RECOMMENDED SOLUTIONS

  -COMMITMENT TO FORMALITY

  -ESTABLISH USABILITY AND TRACKING MECHANISMS

REQUIREMENTS DEFINITION

- PROBLEM: THE COMMUNITY HASN'T LEARNED HOW TO WRITE GOOD REQUIREMENTS
- RECOMMENDED SOLUTION
  - -NEED REQUIREMENTS ANALYSIS TOOLS

COMPLEX LOGIC

- PROBLEM: IMPOSSIBLE TO TEST ALL SYSTEM LEVEL COMBINATIONS
- RECOMMENDED SOLUTIONS
  - -USE OF SIMULATION TECHNIQUES
  - -SAMPLE TESTING OF S/W TO VERIFY STIMULATION
  - -DEVELOP AUTOMATIC TEST GENERATORS

REGRESSION TESTING

- PROBLEM: MINOR S/W CHANGES CAN REQUIRE EXTENSIVE RETESTING
- RECOMMENDED SOLUTIONS
  - -PAY ATTENTION TO TEST REQUIREMENTS DURING DESIGN
  - -ENFORCE STRUCTURED DESIGN, PARTITIONING OF FUNCTIONS TO ISOLATE CHANGE IMPACT
  - -DEVELOP S/W TOOLS TO ASSIST DESIGNERS IN CROSS REFERENCING

SOFTWARE TOOLS

- PROBLEM: TOOL BOX HAS SUFFERED DUE TO LACK OF INVESTMENT

- RECOMMENDED SOLUTIONS

  -STRUCTURE AN INTEGRATED S/W TOOL DEVELOPMENT ENVIRONMENT

  -USE STANDARD HOLS FOR EMBEDDED S/W (ADA)

EDUCATION

- PROBLEM: EE GRADUATES CAN'T COMPUTE -
  CS GRADUATES CAN'T ENGINEER
- RECOMMENDED SOLUTIONS
  - -CAREER COUNSELING
  - -DEVELOP MORE S/W ENGINEERING CURRICULA

WE HAVE THE INCENTIVE TO IMPROVE

- TEST COST NEARING DEVELOPMENT COST
- FIELDED SYSTEMS HAVE KNOWN ERRORS
- S/W IMPROVEMENTS NOT MADE DUE TO HIGH REGRESSION TEST COST

Mr. Greenlee: Thanks very much, Kurt, for your eloquent and dramatic exposition of some of the problems.

As I discussed with most of you, our goal with the Services and DCA presentations is not to hear a formal explication of a Service posture or point of view or policy, but rather a discussion of the state of the art of common practice as it exists within that Service and in its relationships to industry. Successes, failures, lessons learned, problems, etc., the motivation being, of course, to provide us with some kernels or seeds of thought to use in eventually developing guidelines for improving test and evaluation. To focus the presentation, I've asked the representatives of the Services' independent test agencies to lead the presentations, but obviously, as before, we will welcome full interaction and participation by everyone present. To lead off, our first speaker will be Mr. Steve French, from the Army's Operational Test and Evaluation Agency. Steve.

MR. STEPHEN FRENCH: OPERATIONAL TEST & EVALUATION AGENCY (ARMY)

Mr. French: Good afternoon. Don, first I would like to thank you for giving me the opportunity to be a part of this. Hopefully, we can all gain insight and information from each other. I am the Chief of the Methodology and Software Testing Section at OTEA. The subject of my presentation this afternoon is the Army perspectives on software testing.

And, just kind of as an aside before I start, I kind of got a chuckle out of Dr. Fischer's presentation this morning. He made a comment about operational testers getting involved early. When I gave my briefing here to a couple of my coworkers yesterday afternoon to read through and kind of criticize, the only thing they said about it was that I used the word "early" too much. I had to go back through it and try to cross out some of the "earlys" so that it would read a little bit more intelligently. But, that's really the main thrust of what we're doing, we're trying to get more involved in what's going on.

Before I start, really it ought to be recognized that as the Army's operational tester I do not represent the contractor, an Army Project Manager or the Army's Development Test Community. I'm really only representing OTEA. I will be making comments about what project managers do, I will be making comments about what developmental testing organizations do. But, my positions aren't official. They are more to generate discussion and information. The topics of my briefing are shown on the next slide here.

I think the majority of my briefing covers the operational testing perspective and some of the Army Science Board findings. But, I'm going to give a brief overview of some of the other things to put everything in the right perspective. This presentation is perhaps a little premature in that the Army is still in the process of reacting to last years Army Science Board study on the testing of electronics and software intensive systems. There is significant potential for the Army's perspectives to change in the near term as a result of the study findings. At the close of my briefing, I will highlight some of these Science Board findings as an additional basis for discussion.

The testing of embedded software starts with the development contractor. Ideally, this contractor will organize his software development staff independently of his quality assurance staff. This independence of the development and quality assurance is, we think, kind of important to getting the quality out of the product. Both functions are vital to the successful production of quality software.

The contractor's software development personnel are responsible for testing their own program modules. This testing should be extensive and closely tied to the software design specifications.

The contractor's quality assurance team is responsible for independently testing to find errors, problems and deficiencies. This is done by forcing adherence to good programming standards, by holding specification reviews, by introducing error-avoidance or error discovery techniques, by requiring satisfactory documentation, by utilizing software test and diagnostic tools, and by holding peer walk-throughs, preliminary and critical design reviews and all of the standard development error avoidance techniques. This should not replace module and integration testing, which should be done by the individual programmer, but should be done over and above his efforts.

Development programmer testing and quality assurance testing generally address the software design specification which is sometimes referred to as the C-5 specification. MIL-STD 52779A addresses the software QA functions and adherence to these principles is fundamental to the development of quality software. It should be a requirement of a contract that the developer exercise proper QA management techniques.

A second type of testing which is critical to the success of the development of embedded software is the utilization of an Independent Verification and Validation Contractor. The V&V Contractor works directly for the Project Manager and is completely independent of the software developing organization. Verification is the process of checking one software product against the previous product. Validation is that testing done to assure that a product satisfies a requirement.

The tools available to the V&V tester are almost limitless. These include, but are in no way limited to, utilizing automated code analyzers, problem statement languages and analyzers, general-purpose simulations, input/output mapping, error insertion techniques, documentation reviews, walkthroughs, etc.

V&V testing generally tests through the level of the software performance specification or the B-5 spec. The Air Force has published an excellent guide called the management guide for Independent Verification and Validation testing, which I find quite helpful in learning about the kinds of things that ought to be going on in a V&V organization.

Upon delivery of a software product to the Project Manager, government testing begins. DT&E is that test and evaluation conducted to assist the engineering design and development process and verify the attainment of technical performance specifications and objectives. This is generally system level testing and addresses the system specifications or the A-level specifications. These specifications define how the system will functionally perform its mission.

Development testing makes significant utilization of single and multiple thread testing. Because it is unrealistic to expect every logic path to be exercised, developmental testers pay particular attention to the exercise of logic which has a significant impact on system success and which has a high risk of causing difficulty.

To aid in the single and multiple thread testing, TECOM, the Army's development tester, is developing the Modular, Automated, Integrated, System Interoperability Test and Evaluation System called MAINSITE. MAINSITE's emphasis is on the application of automation and simulation technology to assure complete and adequate systems performance and interoperability testing of automated communications, command, control, and intelligence (C^3I) systems. Additional emphasis is on the development of systems performance analysis, test control, data management and specification measurement capabilities. The reason I mention MAINSITE is it is an example of the commitment the Army is making to the development of sophisticated system test and diagnostic equipment and system loading devices to better represent a realistic environment for software testing. The MAINSITE system is going to be a pretty expensive and hopefully, a very worthwhile device for both developmental and operational testing.

Emphasis is also given in DT testing to repeatability and use of diagnostics. This gives the DT tester the capability to probe more fully into the software logic and function by setting up and repeating test events, thus assuring a thorough understanding of what happened and why.

Development test personnel rely heavily on the Computer Resource Working Group. The CRWG is to assist the materiel developer in initiating early tasks and activities that are prerequisite to effective system development and adequate testing. The CRWG includes personnel representing the combat developer, the materiel developer, and the development and operational testers. It provides a valuable forum for information exchange and represents the vehicle for early understanding of the system and its function, the early definition of test and data collection requirements, and the early definition of the scope of contractor quality assurance and V&V testing.

OT&E is that test and evaluation conducted to estimate a system's operational effectiveness and operational suitability. The primary emphasis of operational testing will be on the user requirements, not the system specifications. Operational testing will help identify those errors resulting from translating user requirements into specifications.

The Operational Test and Evaluation Agency (OTEA) is committed to the effective implementation of the following quote, taken from DOD Directive 5000.3. This is a new commitment in that I really don't think that OTEA has in the past done as good a job as we hope to do in the future in the testing of software intensive systems. We're putting a concerted effort to improve in that area. To do this, OTEA has instituted a six person section with the directive to develop and implement an effective operational test methodology for software intensive systems.

As a first step towards effective software test and evaluation, OTEA recognizes the need to become involved early. One of the biggest benefits to be gained is a thorough familiarity with the system, its requirements, its operation, and with the interests, actions, and responsibilities of other development and test activities. Early involvement will lead to early identification of OT data requirements. This should help not only in the test planning but in the elimination of surprises to both the Project Manager and the development contractor. Additionally, knowledge of the available built-in test probes and monitors will help the tester determine useful, appropriate and desirable data. Early involvement will lead to early identification of test instrumentation, simulation and stimulation requirements, hopefully while there is still time for their development and use.

Knowledge of system capabilities, software management methodologies, decision functions, logic, control interfaces, input/output functions, all will allow the operational tester to develop more effective test scenarios.

OTEA executes test scenarios under operationally realistic conditions, and attempts to subject the equipment under test to a broad spectrum of stimuli. It is important for OTEA to recognize the system capabilities just mentioned, as well as their limitations, in order that we will then be able to design scenarios which exercise the system at or near its built-in limitations. If done effectively, this should increase the probability of discovering unknown errors, if they exist. This does not mean that OTEA will be testing only to extremes of performance and environment, but it does mean that when an extreme is appropriate, it will be tested.

The kind of thing that I'm talking about here is one that we came across not too long ago in the file management system in an air defense weapon. The system was built so that it could handle a finite number of aircraft - it could manage them. It was important that we test not only within the band of what it could manage, 5 or whatever it was, aircraft at one time, but that we subject the system to a 6th or 7th aircraft to see what it would do with the additional stimuli. It is very tactically realistic to have 7 aircraft in the sky at a time. The built-in software could only do a specific thing with 6 or whatever the right number was. It was important for us to know that so that we could design the scenarios to look at those particular management-type capabilities within the system.

A major component of the software testing and evaluation effort at OTEA will be the examination of the embedded tactics and doctrine assumed or employed by the software.

Valid questions might be:

- Are the embedded tactics and doctrine compatible with the user requirements?

- Do the embedded tactics and doctrine support the requirements of field use?

- Are algorithms such as development of target priorities or development of IFF rules compatible with the user needs?

- Has the contractor inadvertently been developing his own doctrine, and if he has, is it compatible with the user needs?

OTEA will be making a concerted effort to identify and assess these types of embedded tactics and doctrine.

Due to the nature of operational testing, results will not always be repeatable. If the test events are well documented by test monitors or imbedded probes, problem areas can sometimes be identified. Early knowledge of planned scenarios may lead to more effective test probes; hopefully by leading to more consistent explanation of the test results.

Early involvement will also minimize duplicate testing. This will expand the utility of the all too scarce operational testing. It serves no purpose for OT to run out and do the same thing that the development testers have done a month earlier.

The Army RAM community has recently published its baseline failure definition and scoring criteria. This document, by recognizing the existence of software problems in a forum which requires a management response, puts a significant increase in the emphasis on software. The software section at OTEA will provide software expertise at the scoring conferences. Identification of a test incident as software related serves no purpose unless a concerted effort is made to identify exactly what flaw there is in the software. This is primarily a Project Manager's responsibility, but the tester's participation and understanding is essential.

OTEA will be actively participating in the Computer Resource Working Group (CRWG). The CRWG will be a prime vehicle for the exchange of information and the definition of resource requirements. In the past, we generally haven't been going because we didn't have the manpower, but hopefully, my group will be able to start filling in that gap.

Finally, the Software Section at OTEA will provide the analysis of those system performance parameters affected by software in order to more fully explore and define the degree to which the software supports the intended system function.

The following chart shows some of the key software concerns of the operational tester. By putting them there, I don't mean to imply that those are all inclusive, that one is more important or less important than another; it's just that those are really kind of the key areas we are interested in.

Loading is that testing done at or above some extreme of software logic or computer management capability, the kind of thing I talked about a few minutes ago. It will be tactically realistic, but will hopefully be at some performance or management extreme.

Performance is the measurement of those system parameters which are indicative of adequate software function.

The remaining concerns are fairly self-explanatory, but I will answer any questions if there are any.

Earlier, reference was made to the findings of the Army Science Board. These findings and recommendations are currently a significant impetus to work towards improvement in the testing of software systems. Some of the recommendations of the Science Board are already being utilized. In any case, the Army is currently preparing a response to the Science Board findings. As such, it is premature for me to predict any reaction to the recommendations. But, it may be of value to present some of the recommendations for discussion purposes.

At the early concept stage of development, the following recommendations have been made. Early system simulations are valuable tools which aid in the definition of system requirements and the translation of those requirements into specifications. Many of the problems experienced in software testing are directly relatable to problems in the definition of system requirements and the translation of those requirements into specifications. Additionally, these simulations are invaluable to the developer in conducting trade-off analyses on system design alternatives. These simulations will also aid in the establishing of quantitative and testable system requirements.

The testers' early involvement in the specification and requirement writing process will also facilitate the identification of the computer-based test tools, software monitors and system drivers necessary for adequate testing. Additional funding in early program stages is necessary to accomplish these recommendations.

The next chart shows some of the recommendations in the system development area. The first three recommendations recognize that the cost of software development is much higher than the corresponding hardware costs, and that maximizing the testability and traceability of the software development is more beneficial in the long run than the efficiency to be gained by artful software programming. Sacrificing efficiency for testability and traceability is frequently a good practice for software development.

In the area of testing, on the next slide, the following recommendations were made. The second recommendation is key from OTEA's point of view. Thorough testing and appropriate fix and retest during DT are crucial for getting the maximum benefit out of an OT. It's not an easy recommendation to carry out as it is directly in conflict with current philosophies to shorten the testing cycle by combining DT and OT and to define program schedules that are so inflexible that slips in the start of OT can't be tolerated. But basically, we think that that's a good philosophy to follow so we can get the maximum results out of an OT.

It is only by involvement in the system development that a test evaluator, whether DT or OT, can truly define and assess the relationships between pre-production and software post-production designs. We want to put more effort in that area.

The recommendation to give more emphasis to interoperability testing is the one that OTEA supports highly. Implementation of the Automated Tactical System Test Bed, the ATSTB, concept should help in this area.

The Science Board recognizes the need for an established post-development software growth plan to include testing and fix cycles. By that, they are really recognizing the fact that just because something passes an OT-II or gets into the field doesn't mean that there is not going to have to be a lot more work done on it to make it a first class system.

In closing, this presentation was intended to provide a brief overview of the Army software testing effort and to explore various perspectives toward software testing. In keeping with Don's goal, any questions or comments will be appreciated.

VIEWGRAPHS

USED BY

MR. FRENCH

FOR THE

ARMY

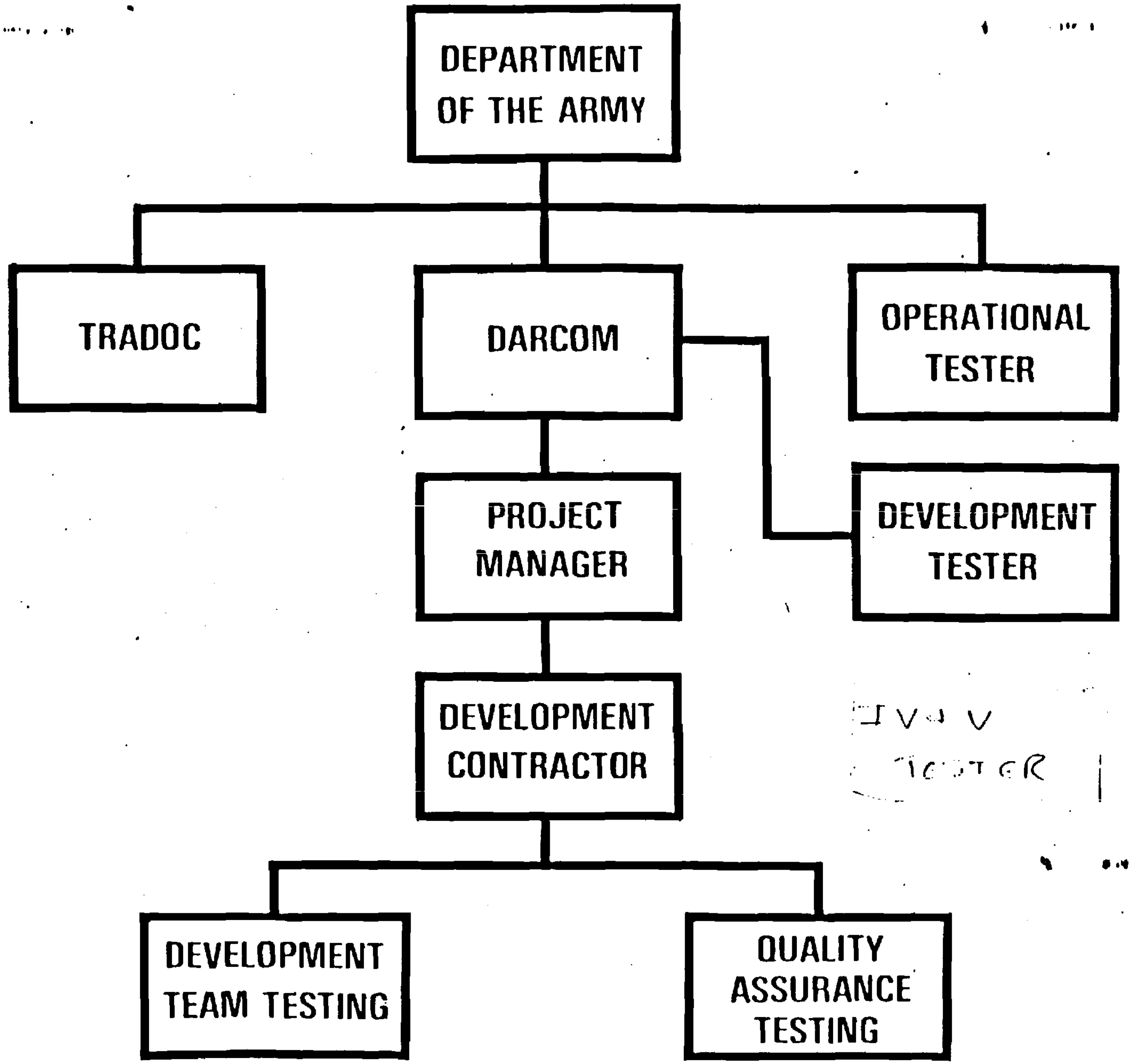
PRESENTATION

OPERATIONAL TEST AND EVALUATION AGENCY

BRIEFING SEQUENCE

- CONTRACTOR QUALITY ASSURANCE AND TESTING
- INDEPENDENT V & V TESTING
- DEVELOPMENTAL TESTING
- OPERATIONAL TESTING
- ARMY SCIENCE BOARD RECOMMENDATIONS

ARMY SOFTWARE TESTING



DEPARTMENT OF THE ARMY

TRADOC

DARCOM

OPERATIONAL TESTER

PROJECT MANAGER

DEVELOPMENT TESTER

DEVELOPMENT CONTRACTOR

DEVELOPMENT TEAM TESTING

QUALITY ASSURANCE TESTING

IV & V TESTER

DOD DIRECTIVE 5000.3

BEFORE RELEASE FOR OPERATIONAL USE, SOFTWARE DEVELOPED FOR EITHER NEW OR EXISTING SYSTEMS SHALL UNDERGO SUFFICIENT OPERATIONAL TESTING AS PART OF THE TOTAL SYSTEM TO PROVIDE A VALID ESTIMATE OF SYSTEM EFFECTIVENESS AND SUITABILITY IN THE OPERATIONAL ENVIRONMENT

OT SOFTWARE INTERESTS

- LOADING
- PERFORMANCE
- EMBEDDED TACTICS AND DOCTRINE
- INSTALLABILITY
- RECOVERY
- DEGRADED OPERATION
- MISSION RELIABILITY
- DOCUMENTATION
- HUMAN FACTORS
- INTEROPERABILITY
- BUILT-IN TEST EQUIPMENT

CONCEPT STAGE RECOMMENDATIONS

- CONSIDERATION SHOULD BE GIVEN TO EARLY SYSTEM SIMULATION
- FUNDING IN EARLY PROGRAM STAGES CAN BE EXPECTED TO LEAD TO REDUCED LIFE - CYCLE COSTS AND SHORTENED TIME SCALES
- EMPHASIS SHOULD BE GIVEN TO ESTABLISHMENT AND DOCUMENTATION OF QUANTITATIVE "TESTABLE" SYSTEM REQUIREMENTS
- COMPUTER BASED TEST TOOLS SHOULD BE DEVELOPED INDEPENDENT OF THE SYSTEM CONTRACTOR TO DRIVE ENGINEERING AND PRODUCTION MODELS OF SOFTWARE SYSTEMS

DEVELOPMENT RECOMMENDATIONS

- SOFTWARE DESIGNS SHOULD BE DIRECTLY RELATABLE TO SYSTEM REQUIREMENTS AND TESTABLE AT MODULE AND SUBSYSTEM LEVELS
- AUDIT TRAILS SHOULD BE PROVIDED THROUGHOUT TESTING
- FORMAL REVIEWS OF BOTH HARDWARE AND SOFTWARE DESIGN STATUS THROUGHOUT ENGINEERING DEVELOPMENT
- A COMMON LIBRARY OF SOFTWARE V & V TOOLS SHOULD BE DEVELOPED AND USED ARMY-WIDE
- ESTABLISHMENT OF RELATIONSHIPS BETWEEN SOFTWARE EMPLOYED IN DT-II / OT-II AND THE ULTIMATE PRODUCTION DESIGNS

TESTING RECOMMENDATIONS

- EARLY PARTICIPATION IN THE DEVELOPMENT CYCLE
- DT BEFORE OT
- FOE'S SHOULD BE PLANNED AS A REQUIREMENT TO ASSURE ADEQUATE FUNDING
- NEED TO STRENGTHEN THE EXTENT AND FIDELITY OF INTEROPERABILITY TESTING
- PROGRAM CHECK-POINTS AND PHASED DEMONSTRATIONS SHOULD BE SCHEDULED AFTER OT-II FOR BOTH HARDWARE AND SOFTWARE IMPROVEMENTS

Mr. Greenlee: We will continue with the Navy presentation, which will be led by Commander Mike Anderson from the Navy's Operational Test and Evaluation Force at Norfolk.

CDR. MIKE ANDERSON: OPERATIONAL TEST & EVALUATION FORCE (NAVY)

Cdr. Anderson: My presentation will be, I guess you can categorize it as a view from the trenches, so to speak. There was some talk this morning about some of the high level decisions and issues we need to address. I find it a little bit difficult to relate to it as I am down there in the trenches shoveling the proverbial cow manure out of the way so we can get something done. So, I'll try to present some of the problems as I see it from my level down there, way down low and, hopefully, that will relate to what we are trying to do here.

We talked about early involvement, I guess we've been talking about that all day. You look on the slide there and see the OPTEVFOR column with a bunch of X's in it. I'm trying to indicate that somewhere in the late 60's, early 70's we were doing operational evaluation, and that was all we were doing. By operational evaluation, you'd show up one day, pick up the system, go out and test it. Since that time, direction has changed and, as you can see by the addition of the X's, that we have got ourselves involved in the system acquisition process right from operational needs/advance system concept design. We're in the acquisition process as early as possible. By the way, feel free to interrupt me any time you want to and ask a question. I don't have a formal type presentation prepared.

This early involvement, as I indicated earlier today, is not as easy as it appears to say it because early involvement means that I've got to get in the program manager's hair quite a bit of the time. In other words, I'm going to look over his shoulder to find out how he's building the system and why he is building it that way, so when I show up to test it, I'll know how it's put it together. In some cases, I can take my recent operational experience. By the way, the criteria for being assigned to OPTEVFOR is that you have to have recent operational experience, so when you go talk to somebody, a contractor or somebody in Washington about what the fleet is doing today, you know what the fleet is doing today. You've just been there, so we have some credibility. So, I can go interact with the contractor and with the program manager with some degree of credibility as I just came from there and I know what it's like.

Traditionally, OPTEVFOR has done a great job, what we consider a great job, in this area of testing the requirements of the hardware that goes together to make a system. In other words, a gun system, be it a 6 inch gun or an 8 inch gun or whatever, we can go out and shoot that gun and do a great job of seeing how accurate it is. As we develop the software intensive systems, we realize now that we have to look at the software, at the same time we've got to look at some operators, some human factors interfaces of the system. We've really got about three areas we need to look at. We need to expand ourselves out here to look at the software and the operators interface with the system. So, we need to watch that, no less than, strictly no less than hardware.

Mr. Devlin: How do you do that? For example, do you use a similar approach to IV&V?

Cdr. Anderson: No, we don't use an IV&V effort, when I say be aware of that requirement translation in the software, I mean you are attending the design reviews, reviewing some of the top level documents that are being used by the contractor to develop a system so you know how the system is being built. Not necessarily to get in there and tell them how to code it or how to draw a flowchart, but to know what is being translated from the written requirement into the written specification. And, if there is a translation problem, try to help it out and correct it.

I went into problems here. I teach the test director course down at Norfolk. I spend about an hour and a half, I guess, talking about software testing. A problem I run into with the typical fleet person who has just come of the fleet, that is worried about making the launch tomorrow in his aircraft or dropping a bomb on target, is that now I'm telling him he's got to look at software, and he says, "where is it?" Most of the time, we have to tell him that it's right here in a pile about this high, and you've got to look at all that paper. Well, you've got to limit what you're looking at to the top level documents. Otherwise, the operational test director, there is no hope of him being able to make any impact at all. He is just swamped, he's got too much paper there.

Mr. Greenlee asked me to have some examples of some early involvement. The most recent one that my shop was concerned with is the system called JTIDS. This system is an advanced command and control system and data link/anti-jam link terminal that will provide secure communications of data and voice and has an AJ margin and is advanced state of the art. It's presently just got through the DSARC II decision point. I think it's come all the way through. OPTEVFOR has been involved in this particular development, which is kind of unusual for OPTEVFOR; we don't usually do any. We hardly ever do OT-I's, but OPTEVFOR did an OT-I on JTIDS. I feel we provided the program some really good direction, because when we went out to the landbased test site at San Diego, and got together with the engineers out there who were building this system, we provided them scenarios and we said, this is what we are going to do with your system. Most of them said, "gee, I never thought about it that way". "I didn't realize that you were going to try to do something with it like that." "Now, I understand why you wanted this switch to do that." So, we feel we may have made a very good contribution to the program by getting involved early at OT-I in this case. Additionally, this is the first DSARC that I've been through, so I got a chance to see how it worked, and I was somewhat impressed with how much they wanted to know about what OPTEVFOR thought about the system. If we had not have done OT-I, we couldn't have thought about the system very much, except by looking at a pile of papers. So, we did two weeks of actual testing out on the West Coast with JTIDS.

I've said some good things about JTIDS, now let me tell you some bad things. This is our interface with the developer, who is the program manager in this case. About six months ago, eight months ago, we were talking in a meeting, and he said something to the effect that I don't care what the system does, I'm building a terminal. And the red flag went up for OPTEVFOR because, when we test something, we're testing a system. In other words, if you have to bring a terminal, an antenna, a new computer on board a ship to make this system function, by God, when we test it, we're going to test the whole thing. We're not just going to test the terminal. We have to test the whole thing. So, we went through several months of discussion with the program manager and finally, he became convinced that maybe he rightly should be concerned with the whole system and not just the terminal, and build a whole system and let us test the whole system. So, early involvement got us somewhere there.

Another example of an ASW system that I've been working on for about a year now, this is a bad example, this particular system has had a history of very poor reliability. OPTEVFOR has been testing it for more years than I care to think about, and every report I read about this system talks about the hardware improvement program going on, we're going to fix this system. But, you look in the next paragraph, and for instance, the last operational evaluation he had, he had one hardware failure and 9 software failures. Well, where is he putting his money? He's putting his money in building better hardware. Now, I don't know what that has to do with how good the software works, but he seems to think it is connected. The system is still not blessed by OPTEVFOR as far as passing an operational evaluation.

I've been talking about system testing, and I will give you some of our philosophy at OPTEVFOR on how we test systems. Scenario driven testing of complete systems, repeat, complete systems, meaning the hardware and the software. We don't test hardware and say that it has 500 hr. MTBF and you ran this software for 25 hours like the MIL-STD says, and it's good. You put them both together, and what you get is a 25 hour system, not what CNO said he wanted. He said he wanted a 500 hour system.

Dr. Fischer: Do you wait until the final version of software?

Cdr. Anderson: No. If that version is going to be used by the fleet, my operational forces will test it before it goes out there.

Dr. Fischer: But, you wait until you get one that is intended for use, however, rather than a development build or something like that.

Cdr. Anderson: Yes. We want to test one that's going to be in the fleet. Now, occasionally, we'll get one that we will get halfway through a test and they will say, "oh, by the way, this is not your final version". And, we'll say, "what are we testing it for then?" We need to test the final version, the one that is going to be used by the fleet. If it's going to be like NTDS model 4.1, 4.2, 4.3, we will try to test or at least look at each one of those versions. NTDS is the Naval Tactical Data System; it is a large command and control system on board most of our ships at sea today. They released the programs in versions model 3, model 4 and model 5 coming up, etc. As we talked about earlier this morning, there is a series of releases over a period of time. New bells and whistles get added, new program versions get put out to the fleet. OPTEVFOR should look at each one of those. In fact, we are required to by a DoD Instruction.

Mr. McOmber: One of the problems in using this for an example, and the reason OPTEVFOR can't get involved, is because all of those systems have been built, modified, and enhanced not with R&D dollars. By charter, you can't do this.

Cdr. Anderson: But, by OPNAV Instruction, I'm required to look at each, if it's a significant modification to an existing program, we're required to do it. NTDS is a bad example because OPTEVFOR has never evaluated NTDS. It's been in existence since the late 60's, and we have not looked at it.

Mr. McOmber: I think we need to do away with the restriction that if it is not done with R&D dollars, you can't look at it.

Cdr. Anderson: Yes, I agree.

Mr. Devlin: The color of monies has no impact on the test or not to test approach. Significant hardware/software modification, however, does. Navy-wise, CNO (program sponsor) and the Systems Command, i.e., NAVSEA, NAVAIR, etc. (developer) make the decision to refer the system to IOT&E or in-house T&E. However, at times, the R&D (098) within CNO's organization makes the decision for all based on the level of effort, including money and impact on operational readiness.

Mr. French: In the Army, the key decision point that the operational testers really get into is OT-II. I don't think I'm overstating it if I say we don't seem to have the ability of getting production software at an OT-II decision point. What is different about the Navy that allows you to wait for your testing until you get a software version which is production software? Are you talking about a later level testing, maybe OT-III?

Cdr. Anderson: We do OT-II testing primarily.

Mr. French: How do you get production software to test for an OT-II when the Army can't?

Cdr. Anderson: I don't know. Jack, do you know?

Mr. Devlin: A lot of decisions are made early in the program as to what, when and how, at least in documentation. The Test and Evaluation Master Plan is your contract between the developers and the operational testers. If you call for production software or at least a prime example of production software, you'll be headed in the right direction. First thing, the software is not going to be error free. You are going to turn up bugs, but if you can get a good base, you can get the system marriage, the hardware and the software. Maybe, we have a little bit more latitude or Navy exercises more control over it's DA's in just what is required to be tested during OT-II well ahead of the start of OT-II testing.

Mr. French: How do you define your requirements for this baseline software? I'm not sure I understand what you mean by that.

Mr. Devlin: Baseline software is what you have when you have a strict configuration control program. Baseline software only changes when there is reason for change, i.e. numerous PTR/ECP trouble reports that have been tested and found problem-free. This allows for re-compilation of the baseline software.

Mr. French: Where the Army went wrong with Patriot, we got to OT-II, and we had software that was so incredibly terrible. Was it in the way that we wrote our TEMP, was it in our contract?

Mr. Devlin: How about early involvement?

Dr. Fischer: But, it boils down to what you just said, the TEMP, I think that OT agencies do have an input. They have that opportunity, everyone has that opportunity. There is a wide range of people and organizations that have an input into Test and Evaluation Master Plans. That's your shot. That is your early involvement.

Mr. McOmber: Does the Army have someone at any level to certify the system ready for OPEVAL, for example, a TECHEVAL?

Cdr. Anderson: That's right. Prior to the Navy taking the system to operational evaluation, the developer has to complete what we call a TECHEVAL, as part of that TECHEVAL, he's required by the Military Standard 1679, etc. to complete a certain amount of endurance runs on his software, to stress it so many hours, to run it so many hours, and he has to either up front tell the CNO that he has not completed that or that he has successfully completed it and sign a paper to that effect. Now, I don't see in the Navy a problem with getting too immature software for OPEVAL. I don't see that problem. I'm not saying that we don't have bad software systems out there, it's not because they are not mature, just a terrible job of development.

Cdr. Anderson: I don't know. Jack, do you know?

Mr. Devlin: A lot of decisions are made early in the program as to what, when and how, at least in documentation. The Test and Evaluation Master Plan is your contract between the developers and the operational testers. If you call for production software or at least a prime example of production software, you'll be headed in the right direction. First thing, the software is not going to be error free. You are going to turn up bugs, but if you can get a good base, you can get the system marriage, the hardware and the software. Maybe, we have a little bit more latitude or Navy exercises more control over it's DA's in just what is required to be tested during OT-II well ahead of the start of OT-II testing.

Mr. French: How do you define your requirements for this baseline software? I'm not sure I understand what you mean by that.

Mr. Devlin: Baseline software is what you have when you have a strict configuration control program. Baseline software only changes when there is reason for change, i.e. numerous PTR/ECP trouble reports that have been tested and found problem-free. This allows for re-compilation of the baseline software.

Mr. French: Where the Army went wrong with Patriot, we got to OT-II, and we had software that was so incredibly terrible. Was it in the way that we wrote our TEMP, was it in our contract?

Mr. Devlin: How about early involvement?

Dr. Fischer: But, it boils down to what you just said, the TEMP, I think that OT agencies do have an input. They have that opportunity, everyone has that opportunity. There is a wide range of people and organizations that have an input into Test and Evaluation Master Plans. That's your shot. That is your early involvement.

Mr. McOmber: Does the Army have someone at any level to certify the system ready for OPEVAL, for example, a TECHEVAL?

Cdr. Anderson: That's right. Prior to the Navy taking the system to operational evaluation, the developer has to complete what we call TECHEVAL, as part of that TECHEVAL, he's required by the Militar Standard 1679, etc. to complete a certain amount of endurance runs on hi software, to stress it so many hours, to run it so many hours, and he h to either up front tell the CNO that he has not completed that or that h has successfully completed it and sign a paper to that effect. Now, don't see in the Navy a problem with getting too immature software fo OPEVAL. I don't see that problem. I'm not saying that we don't have ba software systems out there, it's not because they are not mature, just terrible job of development.

Cdr. Anderson: Pressing on. Report effectiveness and suitability of a total system. It seems like a simple statement, but in actuality it is not in practice, because of a tendency to allocate software to a separate box and say, we'll let it run 25 hours, and we'll make the hardware work 500 hours. OPTEVFOR will look at the total system, including hardware and software, as a system in itself, and will attempt to identify the software errors, but only to tell the developer that he has a problem in this module or that module. The software errors are strictly for correction only. The overall system effectiveness and suitability is used for the approval for service use determination. ASU is the Navy paper that says deploy it. Buy it, deploy it. In the fuzzy category area, effectiveness and suitability sometimes get a little mushy in software. Effectiveness is how well it does its job, and suitability is the reliability, maintainability, durability, all the "ilities. " Effectiveness, if I've got a software problem that say, I've got a function that I have to do and it takes 10 steps, and if I programmed it correctly, the program worked correctly, I could do it in one step. The fact that it takes me 10 steps to do it might take me 20 seconds instead of 10, but I can still do it, therefore, I'm working around the problem and it is not as effective as it could be if it were properly programmed, but it works. So, I can work around it. Suitability, reliability of the system, what I'm saying then is, hey, every 10 hours the thing faults and doesn't run anymore. I've got to reload it. I've got a critical or major failure. That's your suitability.

In suitability testing for reliability, which is the meantime between failures, we'll calculate it the same way for software as we do for hardware if we've got all the major failures corrected. In other words, it runs. I don't take it out on my ship, turn it on, it runs for 10 hours and quits, and it's going to take a programmer to fix. I can't do that because I can't fly a programmer to the Indian Ocean to get it fixed. Some minor failures have been corrected or worked around, and the system is large. If that is true, I can then take the number of failures divided by the number of hours and I get an MTBF for software. Only used for the DA to fix his problem and you give him a report card. I don't use that to say the system is good or bad.

Maintainability is slightly different. If I have a failure, how long does it take me to fix it? For software, it may be a matter of seconds. I might have an automatic reload or bootstrap in this program and it will take 5 seconds and it's back on line again. So, how long will it take me to fix it? Five seconds, or two weeks to get a programmer out there to fix the problem? Big question, right? There's no easy answer. We take the easy way out at OPTEVFOR and say that it takes 5 seconds because I'm up and running again in 5 seconds and I can shoot a missile or direct an aircraft or drop a bomb after the reload. I may expect that in another hour or so it's going to go down again, and it'll reload itself, but it is still working. I haven't terminated the program.

Mr. French: What if the process that caused you to go down is a function of whatever exercise it takes. Just because you come back up in 5 seconds, if you go down again because you have the same aircraft flying over you, you've got the same cause of failure.

Cdr. Anderson: Did that prevent me from doing a critical function of this system? If it did, then I've got a major failure. I'm probably going to have to say "the exercise is over, let's go back home to get it fixed." I'll do it again. I'd go out and take it back out and retest it.

Maj. Hammond: Repeatability may be the word. If you continue to repeat, let's say, the same divide by 0 function, you've got a major problem. However, if it is just one loose 1 or 0 running around there that every once in a while decides to raise its ugly head and crash the system, you've got a different problem. The next thing to ask is, when you reload it, what did you lose? Now, you are in the effectiveness area. You then consider the worst case. Well, I lost everything. Now, you have to move it up from the maintainability area to the effectiveness area. It is not very effective when you're reloading.

Cdr. Anderson: Notice also that the restore time must include databases and files. If I've got a command and control system, I'm tracking 300 aircraft, and I dump my program and the automatic reload takes over, my program's back up in 5 seconds. What about those 300 tracks, those 300 aircraft out there? I've got to wait until they're back in there before I start/stop that clock. Other words, I'm not effective until those databases and files are reloaded, reinitialized.

Availability, which is operational availability, which we define in OPTEVFOR as up-time over up-time plus down-time, and will give you, normally we look for .9 or .95 availability. In the case of software alone, a properly designed system with a very small maintainability will give you availability of almost 1, because your downtime is very small, seconds. As compared to the total runtime, which is maybe days or weeks. So, in the case of software, availability doesn't mean much to us really.

The remaining suitability tests, such as interoperability, a whole bunch of "ilities", are basically the same for software as hardware. I do have a couple of additions down here. Human factors is becoming an increasingly big problem, particularly in the area of diagnostics. The Navy is presently buying or getting ready to buy a new mainframe embedded computer and a couple of smaller computers. The Navy computer standards in the late 80's, and their maintenance concept is they are going to give somebody a couple of weeks training, put him out on the ship, and he's going to be able to maintain them. The only way that will work is with good diagnostics that will point that limited ability technician down to the card, in this case, they need to replace. So, diagnostics is becoming an increasingly big problem in the Navy. Our answer to the problem with not having enough qualified technicians out there is you better build one into the system, so we don't have to have many really qualified technicians that can change a chip. Maybe, all they wanted to do is change a box or change cards.

Along that line, my last slide here is that I've taken liberties with this famous software iceberg and turned it upside down. The operational tester is not only concerned with this down here, but he's looking at everything. So, when the system goes into operational test, you better have all those wickets in line, because we're probably going to look at them all. And, we should look at them all. We may not have the time, we may not have the people, but we're going to look at them.

Any questions?

Maj. Hammond: Do you really believe that the concept of MTBF has any validity? Software doesn't wear out; it goes down because it gets an input it can't handle.

Cdr. Anderson: Well, I agree with you, and I'm taking a simplistic look at it from the operational testers viewpoint. Academically, I'm sure it is not proper to do it that way. But, for our purposes, the operational test purposes, I don't see anything wrong with it. All the operational tester has to do is tell the captain of that ship, "hey, it's going to run". "It looks good to us, it's going to run." Or, "you are going to be able to fix it." And, I don't think we need to go beyond that as far as operational testing goes.

Maj. Hammond: It just seems like an awfully artificial concept. I can give you 100% reliable software as long as you allow me to restrict the input.

Cdr. Anderson: It won't be reliable if you require the input to be restricted such that it won't do its mission. Then it's not a reliable system, because it's going to go down.

Maj. Hammond: That's my whole point. The reliability of the system depends totally on the input and not on how long it's going to run.

Cdr. Anderson: That's right. The scenario developed to test that system will ensure that all possible inputs or all expected inputs are there and that unexpected ones don't cause the software to abort.

Maj. Hammond: I would be willing to bet that if you went back and looked at it, you were exercising a negligible percentage of the possible inputs.

Cdr. Anderson: No doubt about it. No doubt about that at all.

Mr. French: I think, at least from the Army standpoint, it becomes a valuable tool not in that the answer is .8 or .9 or whatever it is, but that when you start talking about it in reliability terms at a scoring conference or some other process like that, the Project Manager is forced to respond. That then becomes a valuable tool.

Cdr. Anderson: That was at the top of the line, that we are going to look at the whole system and the discussion that went under RM&A for software is strictly for the developer to know how well he did and where he has to work to fix it.

Maj. Hammond: I think in that case it would make more sense to say it works in this area, and it doesn't work in this area. It works if I try it with this, and it fails if I try it with this.

Cdr. Anderson: You can say that, but 90% of the developers are going to want you to put a number on it. Give me a grade, I need A+ or B or C-. They want you to quantify it.

Maj. Hammond: However, I would object if someone gave me a grade like that.

Mr. McOmber: I have two stories to tell about mean time between failure problems. One, a few months ago, Admiral Lewis called down to our office and sent a few figures he had copied out of a TEMP for one of the Navy systems. In that TEMP, it had mean time between failure for software thresholds and goals. He asked, "what does that mean? Are these good figures? Could they ever be reached?" I called the system engineer and asked him, "where did you get those mean time between failure rates?" He said, "look out the window and just grab and pull them down." "That's where we got them." My question to you and the guys at OPTEVFOR is, is that what they should do?

Cdr. Anderson: They should not.

Mr. McOmber: I agree that maybe you guys have a use for them as a lever going back to the guy that's developing the software, but the goal is unrealistic, absolutely meaningless.

Cdr. Anderson: The only number that should be in the TEMP is the system criteria. That's it.

Mr. McOmber: Another example. I know a guy that helped develop, worked for IBM, one of the large airline reservation systems, and they concluded that it would take over 100 years to test the system out in all of its variations. So, they decided to stop at a certain point. They stopped at a certain point where they had developed a mean time between failure that had some meaning for that system. They know that every 8 hours the system is going to go down with a soft error. It will be software related, but it won't lose data, it will come back up in a few seconds, and they don't even try to fix it. They're going to live with it. They expand that to a little bit harder error. Once a day, it is going to crash and take 10 minutes to fix. They still don't figure it is going to be useful to them to figure out what the problem is. They don't care. They can live with it, and from that, now they have concluded they have a mean time between failure. And, that may be appropriate because they are never going to go and try to find the error.

Cdr. Anderson: The key is that you said that they could live with it. Now, if the captain of the ship can live with a system that goes down every 8 hours and is back up in 5 seconds, no problem. If that system goes down every 8 hours and stays down for an hour, they aren't going to live with that because what if he gets attacked by a bad guy, and he wants to go push the button, there is no way to do it. "Sorry, Captain, it's down, be up in an hour." No.

VIEWGRAPHS

USED BY

CDR. ANDERSON

FOR THE

NAVY

PRESENTATION

TEST PROCEDURES

1. SCENARIO DRIVEN TESTING OF COMPLETE SYSTEMS IS STILL THE KEY.

- REPORT EFFECTIVENESS AND SUITABILITY OF THE TOTAL SYSTEM
- IDENTIFY S/W ERRORS FOR DA CORRECTION
- OVERALL SYSTEM EFFECTIVENESS AND SUITABILITY USED FOR ASU DETERMINATION

2. FAILURE CATEGORIES

- EFFECTIVENESS - ARE "WORK AROUNDS" AVAILABLE?
- SUITABILITY - TOTAL SYSTEM FAILURE OR COMPLETION OF PRIMARY MISSION IS PREVENTED

3. SUITABILITY TESTS

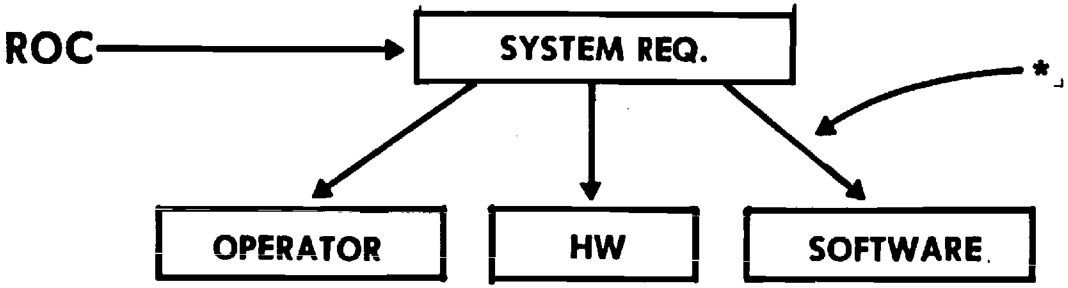
- **RELIABILITY (TEST S-1) CALCULATE THE SAME AS HW IF:**
  - A. MAJOR FAILURES HAVE BEEN CORRECTED
  - B. MINOR FAILURES HAVE BEEN CORRECTED OR WORKED AROUND
  - C. THE SYSTEM IS LARGE

- **MAINTAINABILITY (TEST S-2)**
  - A. DIFFERENT FROM HW
  - B. RESTORE TIME MUST INCLUDE DATA BASES/FILES
  - C. MAY BE A JUDGEMENT CALL

- **AVAILABILITY (TEST S-3)**
  - A. VERY CLOSE TO 1.

- **REMAINING QUALITATIVE SUITABILITY TESTS**
  - A. THE SAME AS HW
  - B. HUMAN FACTORS

COMMITMENT TO EARLY INVOLVEMENT



* WATCH THIS LINE OF REQUIREMENTS TRANSLATION

NO LESS THAN THE OTHERS

	DEVELOPING CONTRACTOR	DEVELOPING AGENCY	SUPPORT AGENCY	OPTEVFOR	FLEET	PARTICIPANTS / SW PROGRAM PHASES
MS 0				X	●	R&D OPERATIONAL NEEDS, ADVANCED SYSTEM CONCEPT
		●		X		ADV. DEVELOPMENT PROPOSAL/SYSTEM REQUIREMENTS
MS I OT-II REQUIREMENTS		●		X ✓		SOFTWARE PERFORMANCE DESIGN REQUIREMENTS
MS II	●					PRELIMINARY DESIGN
SW OT-III DESIGN		●		X		PRELIMINARY DESIGN REVIEW
	●					DETAIL DESIGN
MODULE OT-III IMPLEMENTATION		●	●	X		FINAL DESIGN REVIEW
	●					CODING AND TESTING
SW OT-III INTEGRATION & TEST	●		●	X		SOFTWARE TEST AND INTEGRATION
		●				SOFTWARE ACCEPTANCE
			●			INSTALLATION
OT-III OPEVAL				●		OPERATIONAL EVALUATION (SYSTEM)
MS III					●	OPERATIONAL USE
OT-IV/V			●			MAINTENANCE

Mr. Greenlee: Very good. Thank you, Mike. You are a good lightning rod. Our next presentation from the Air Force side is by Lt. Col. Mike Blackledge from AFTEC (Air Force Test and Evaluation Center) at Kirtland AFB. Mike?

LT. COL. MIKE BLACKLEDGE: AIR FORCE TEST & EVALUATION CENTER

Lt. Col. Blackledge: Thank you, Don. I'm Mike Blackledge from the Air Force Test & Evaluation Center at Kirtland. We know in the Air Force that the rated personnel are out there depending on us to evaluate their software. What I want to do today is not go over very much background because I think you are all sufficiently motivated that software is a problem. But, I will tell you what we're doing at AFTEC, how we're organized to attack that problem, what we're doing about it now, and what we plan to do about it in the future. I just have a couple of background slides to remind you that the basic problem that people have been stating is that most software errors occur early on in the requirements and design time frame, but they are not found until you get into the testing time frame, and that is usually much later on in the program, when the costs are significantly higher.

I promised Don I wouldn't show any regulations, but I put one in here to remind you also that we're kind of unique in that we have our own special paragraph in DoDD 5000.3, Test and Evaluation. One paragraph just for software. I'm still fighting that problem even at AFTEC. People say, "how can you have a special group to evaluate software?" "We don't have a special group to do engines." I guess the point is that they don't have as much trouble with engines. Perhaps someday, we won't have a special group for software. That's really what we are all pushing for. Perhaps after there are computers in every home, and there are 1,400 standard test tools on the shelf to pull off whatever you want. Then maybe we won't have to break software out separately.

Here is a stylized look at the way we're set up at AFTEC to handle software. Essentially, for every major Air Force system, there is a test manager assigned. One individual dedicated to that one project. He doesn't write the test plan, he doesn't do the test design or test plan all by himself, he draws on other expertise in other areas within AFTEC. He gets a resource manager to help him set up what he's going to need to run the test, he get's an Ops Analyst to help him define how the operational effectiveness part of the test should go. He gets a logistician type analyst to help him define how the suitability part of the test should go, and he gets somebody from our shop, a software specialist, to help him define what kind of software subobjectives he should have within that test.

Mr. Devlin: What kind of folks do you have in the software evaluation area?

Lt. Col. Blackledge: That fits in on the next slide.

Mr. Devlin: Before you leave this slide, let me ask one question. How many programs does that one test manager manage?

Lt. Col. Blackledge: The one test manager manages one. But, the software people such as we have are divided into different functional specialties. For example, one individual might handle three or four different C^3 systems. We have one super guy who handles the air launch cruise missile, the ground launch cruise missile and the medium range air-to-surface missile. So, one software guy lends his expertise to several test planning groups within the headquarters. The test manager however is dedicated to that program.

We have right now about 16 people, and we are going to about 20-22 by the end of the fiscal year, as far as software specialists. These are two different kinds of people. You saw the slide that computer science people can't engineer and vice versa. Well, we fight that problem by taking half of each. Half of us are EE's, and half of us are computer science type people. That's just at the headquarters. Out in the field, we have a test team. AFTEC sets up a test team at Edwards AFB or wherever the particular system is going to be tested, and on that test team, if it's for a software intensive system, we also place one of these software specialists. We try to make this an AFTEC slot. That's not always been the case in the past, and that's kind of a new policy. The air launched cruise missile happened to be a Strategic Air Command individual, but we try now to make this an AFTEC slot. You'll see that very few positions on the test team are actually test and evaluation professional testers, if I can use that term. As you all know, there is no specialty for test and evaluation. These are people on whom we've painted the AFTEC badge and given them some training on what they should know about operational testing and evaluation. But the Deputy for Software Evaluation we try to make an AFTEC slot. The people working for him will come from the using command or come from the supporting command to help evaluate that software.

The next chart has an awful lot on it, but it covers pretty much what we do. Let me go over it just a little bit. You see the "early" on there again. Same type of thing. We try to get out there early, as I mentioned this morning, we review the request for proposal, statement of work and whatever comes across on the system in order that we can place the right kind of hooks in it. For example, we suggest that they use MIL-STD-1679 or suggest that they put in some hooks for independent verification and validation. We try to get those things in early. We also write up the software portions of the test plan. We first do a test approach to outline how we are going to do it. All those people that I indicated were on that test planning group from the headquarters, all get together and do their own specialties, write up their own objectives, write them all together into one overall test approach. Then, from that, the test plan is developed. We also attend preliminary and critical design reviews, we participate on the Computer Resource Working Group and the Test Planning Working Group. Once the test is underway, we (the AFTEC people or the test team people) try to observe the in-plant testing that goes on. We don't participate in that directly, but indirectly we get involved in it as much as we can. We do get directly involved, of course, in the onsite testing. We take the test data and evaluate it, do the analysis on it, write up the report, and the report is forwarded to the interested parties and directly to the Chief of Staff of Air Force. That is kind of an overview of what we do.

Mr. Watt: How much do you affect the actual test scenario?

Lt. Col. Blackledge: That's something we're trying to get into a little bit more. How much do we affect the actual test scenario. One of the things we are going to use is the independent verification and validation contractor. If he's got some ideas on whether there are critical modules in that particular software, what things might happen, we try to add some OT&E type tasks onto his IV&V contract to help us define how that test scenario should look.

Dr. Fischer: Are you saying you use OT&E money then to pay the IV&V contractors?

Lt. Col. Blackledge: This is something brand new for us that we are just trying. Right. Just doing this year. Do an add-on task.

Mr. Devlin: Who hired the IV&V contractor?

Lt. Col. Blackledge: The SPO, the Program Office has hired the IV&V contractor.

This slide I love because we've been working for 3-4 years on how to put what we do on one slide, and this is the closest we've gotten. Overall operational test and evaluation, as you've seen, is broken up into operational suitability and effectiveness. How does the software evaluation affect that? We support both of those areas. We do it by trying to answer these three questions: Does the software restrict or even degrade the system's performance? Does the software help the individual who is trying to run the system? And, how easy is it to change the software? I can rephrase those into maintainability and effectiveness and other things, but that kind of sums up our job quickly. Now, how do we do that? Well, in the operational effectiveness part of it, software performance as it is called here, we're really looking at a system level test. We run the system level test and use a "by-exception basis" as our standard approach. If something broke down, then take a look at that problem. Is it a software problem, is it a hardware problem, and go from there. The trouble is how do you know that you've exercised the right paths in the software. That's where I mentioned that we try to use that IV&V contractor to give us a little insight. He is supposed to be intimately familiar with that software; if there is indeed an IV&V contractor on board, he should be intimate with that software, he should be able to give us some insight into how to influence that test design.

Maj. Hammond: Do you find any problem with the PM paying the bill and the tester asking the questions of the IV&V guy?

Lt. Col. Blackledge: Not when we add it on as a task, we don't. That's what we are trying to do. You are talking about the color of money. That's one of the problems, with the color of the money. They try to add that task on with our money.

Unidentified person: You are going to attempt to add a task on to the Program Manager's expense action with the IV&V contractor ...

Lt. Col. Blackledge: Right. I'll say we haven't been successful yet with getting through the contractual binds, but we're still struggling with it.

Unidentified person: The money you are using at that point in time is probably RDT&E money which comes through the Program Office anyway. So, you are using the Program Manager's money ...

Lt. Col. Blackledge: The Program Manager usually doesn't know that that money was pulled off ... We don't tell him that! That's not been a problem. The problem has been things like 3400 money versus 3600 money type problems. The program office has not been unresponsive to us going out and setting up our separate tasks with the IV&V contractors.

Dr. Fischer: What programs have you tried that on?

Lt. Col. Blackledge: This is on the Global Positioning System user equipment, and we're also going to do it on another segment of GPS. Those are the ones we are working right now.

Another area which is essentially in the operational effectiveness area is the operator-machine interface. When I say operator here, I'm not talking about the computer operator. I'm talking about the user of the system. Was that software designed with the individual in mind that's going to use it? Our methodology here is a standard questionnaire that we have developed over the last two years that has about 95 questions on it. Here are some typical questions that we hit the evaluator with, and we have a standard handbook that we hand him which has each question on a page and a glossary and instructions. We're not talking to computer people here, we're talking to operators, electronic warfare officers, the guy at the C^3 console, whatever. In order to take our questionnaire, he has to have enough familiarity with the system to feel qualified to answer it, at least two or three weeks working on a new system. He's been out there, he's observed the system if it crashes, and so on.

Maj. Hammond: I noticed that your third bullet up there is "menu techniques are used to aid the operator in making decisions". Does that mean that the gospel according to AFTEC is that menu techniques should be used?

Lt. Col. Blackledge: Not gospel, but ...

Maj. Hammond: Some operators would contest that.

Lt. Col. Blackledge: Well, that may be. Yes, we are implying that it is easier if you have a menu to choose from than having to know or look up in the documentation what your next step is or what choices you have.

Maj. Hammond: Yes, it is easier for an unskilled operator, but it really slows down a skilled operator.

Lt. Col. Blackledge: We cover that in other places in the questionnaire, namely that the operator has control over how much help he's given. So, we hit that separately.

Maj. Hammond: I just wanted to make sure your test criteria were not biased.

Lt. Col. Blackledge: Toward an unskilled operator. We cover both areas. In summary then, under the effectiveness area, I think I've covered most of these. One thing I haven't covered is the support system. We run a separate evaluation, which we are just getting into, on software support facilities, for example for a large system having a separate software support facility. We found that these are not identical, you can't run one evaluation for all facilities. You can ask certain questions or find out certain things that are common to all facilities, but when you get down to whether they're making software for an operational flight type software or doing air cruise training devices, or they're doing automatic test equipment, then they get into specialties and you've got to make sure your evaluation covers those particular types of support facilities. So, we tailor our particular evaluation in that case, for each of those different types of facilities. Here's an example of what it looks like in the test plan. These are just the subobjectives that relate to these particular software areas that I've been going over. To give another example with the operator machine interface questionnaire, when we administered it on the EF-111, we found the system made a very low score relative to workload tests on the operator machine interface, and it turned out the electronic warfare officer had to punch in 1,000 key strokes before the aircraft could take off. The joke around the program office was (holding up a stub of a finger), "Hi there, I'm an EW officer on the EF-111." So, the software showed up poorly. The program officer agreed and the software was sent back to the contractor to make that change. They did make a change, we went back and tested it again, and it showed up much better the second time around. The operators obviously agreed, as that was what they were showing in the questionnaires.

In the maintainability area, our methodology is well structured and has been used for more years, about four years now. This is also a questionnaire type approach, not unlike that of the operator-machine interface. What we do here though, is we are asking the questions of individuals that are going to be maintaining that software, if it's going to be assigned to a particular air logistics center or if it's going to be assigned to the communications computer programming center out at Tinker AFB, or down at Warner Robbins. We draw in some typical software maintenance people from that organization, bring them in on temporary duty to where we've gathered up some listings from the system and the documentation for the system. We have them go over this questionnaire.

They don't need to know what the function of the software is. They do have to be experienced programmers and maintainers, as opposed to the individuals evaluating the operator-machine interface. Here we want programmer types. They go through this type of questionnaire with one questionnaire for a particular software module and one questionnaire for all the documentation in the program. What we do is go through question by question, we talk to them about the questions, we see if they have any misunderstandings, then we have them go through a calibration run on one module. Then we get all their answers and go back over them and see if there is a big disparity. The answers are forced into the choices from completely agree to completely disagree. The questions (as in the operator-machine interface) are not questions, but desirable characteristics. And, they're answering to what degree this particular software has those desirable characteristics. So, then when we take a look at what their scores are, if this guy gave a high score and the other 4 or 5 evaluators said it was a low score, we talk about it in the decalibration or debriefing session and make sure both groups are answering the same question or have the same understanding of the question. If the guy really felt that way strongly, obviously, we are not going to try to change his answer, but we want to make sure that he is thinking about that question the same way those other guys are. Maybe he's right and they're wrong.

Mr. Devlin: You might make a plug for your guidebooks or handbooks ...

Lt. Col. Blackledge: Thank you, I hate to do that because the maintainability one was written up in a national computer conference that had a maintainability panel, and a speaker brought it up there. Somebody wrote to us and asked for it and under the freedom of information we sent it to the somebody, the somebody turned out to be the reviewing editor for EDP Analyzer, and they reviewed it and gave a glowing review, and we got about 300 requests for that manual since last winter --- everybody from the London Stock Exchange to Procter & Gamble. So, I guess, software maintainability is a common problem.

Mr. Devlin: I'm one of them, and I haven't got it yet.

Lt. Col. Blackledge: I'll get one to you. We have a set that will eventually be six unique volumes in a set called Software Operational Test and Evaluation Guidelines. Some of them are designed for in-house use. The first volume, for example, is for the software test manager, the guys who work in my shop that help the test manager write his test plan. Other ones, though, are the evaluator handbooks; for example, volume three is for the maintainability evaluator. Volume four is for the operator-machine interface evaluator. Volume five will be that software support facility evaluation, and that one is not off the press yet. That software support facility evaluation is the area we are looking at here. A lot of times, you don't have a software support facility that is available, you're just looking at plans. That's one of the reasons we need this tailored type of tool. We may just be reviewing plans as far as the software support facility. Or, it might be a completed facility like the AFSATCOM System which is still in follow-on test and evaluation. It has this facility all there in place, set up at Tinker, and we can go out and get people that are actually running that system. So, we find everything from plans all the way to the final product.

Here is a sample of what would be in a test plan as far as the subobjectives for the suitability side. Both these examples were taken from the Tri-Tac system, the Tactical Communications System. From the Communications Nodal Control Element of the Tri-Tac system. What kind of results do we come up with, well, as I say, we force our (at least) 5 evaluators into a completely agree to completely disagree, and for the type of results that we can express from this type of work, we set up a goal and a threshold value for those averages. We try to force them into a normal distribution on their responses by guiding them away from "completely agree" and "completely disagree." This is an old slide that I love to show because it makes a point well. What I'm showing across here is the programs; we do these evaluations by modules, but you can group the modules into a program, you can group the programs into a subsystem, and you group the subsystems into a system. We used the EF-111 Tactical Jamming System and the F-4G Wild Weasel for individual programs. The top part is for the documentation, different characteristics that we look at under the documentation. The bottom part is for the actual source listings, the design of the code. You see some things that show up, others are old characteristics and they've been re-arranged a little bit, but the structure of the code shows up well. Blue being good, red being bad. What shows up interestingly in this particular one is that when you look across here, it turns out that this one particular module, called EXACT, has a super rating on it, and it turns out that the prime contractor did these two programs, and he subcontracted out this EXACT program. Apparently, he forced good programming practices on his sub, but not necessarily on his own people.

What are we doing now? What kinds of new things are we doing? Well, I've mentioned a couple of them, like the software support facility evaluation that we're working on. These are the types of things that we are trying to overcome. Other areas that we are working with are the event trace monitor, standardized test tools, and independent verification and validation applications. The event trace monitor is usually described as a logic analyzer, but it is more than that. We've taken a standard off the shelf commercial COMTEN 8028, I believe it is, and added on to it, what Hughes built for us: this universal selector component, they call it. You can dial in certain addresses from the software. This particular hardware monitor you can clip onto your operating computer.

Whenever you hit that particular address, it chunks it out to a tape and gives you a time stamp of when you hit it so every time the address is pulsed, you know about it. It looks like a DT&E tool, right? Well, it is pretty much a DT&E tool and that is why we're not talking about this too much, but it can be used as an excellent OT&E tool, if you have somebody that is intimately familiar with the software. You've got to have somebody that is really educated on the software. Where do you get them? Well again, you've either got to be on good terms with the developing contractor or the IV&V contractor, one or the other. We have used this this year for the first time, on two different programs. On the Over-The-Horizon Backscatter radar in Columbia Falls, Maine, we hooked up to a Univac 1616, and on the Tri-Tac program down at Ft. Huachuca, we used it on the equipment there, and we're still getting the data from that particular one. What can it tell you? If you suspect a particular area is a bottleneck, then you can dial in those particular addresses, like the entry point of a module where you say, "I'm worried that this area is going to be overworked", and it'll show you. You'll get your printout, or you get your tape, take it off-line, do your data reduction, and you get an indication of how often that module was hit. If you had another area, a background type of thing, a waste type of job, you can find out how much reserve time you have in the program. Those are DT&E type of things. So, if you really want to use it well as an OT&E tool, you've got to know what you want to check very well.

Dr. Fischer: Why do you use a tool like that? Is it because you at OT are trying to do a super job or is it because people in DT don't do their job?

Lt. Col. Blackledge: First reason, rather than the second. I sort of emphasize that we can keep that computer operating in its operational environment, and this monitor will not interfere with its operation, whereas if you put in some kind of software testing in there, it is going to change the operational environment of that system. We don't have it in a pod yet for mounting on an aircraft, and may never do that, but for command and control systems like Tri-Tac and OTH-B radar type systems, we can clip it on without affecting its operation at all. So, we can get more visibility into what's happening in that software and find out whether or not it's got bottlenecks, or if it's working right on the ragged edge of operational capability. In other words, if it were pushed a little bit more, if it's going to drop off the cliff, or whatever. We can get a little more visibility into what's happening when the system is actually running.

Mr. Devlin: That is over and above reports that are already available to the DT testers or programmers?

Lt. Col. Blackledge: Right. One company, Martin-Marietta, got kind of excited about this and they are looking at something like this as an IR&D project. We'd love to see that. We would much rather have somebody else do it in industry instead of us. We really don't want to get into the hardware business.

In the structured area, we encourage better software design techniques, of course, and we're strong advocates of Ada. We're going to start pushing that when we see the RFP's, we want to make sure they've got that all covered properly. 1679 is a MIL-STD, a Navy MIL-STD, that we think is excellent. The Joint Logistic Commanders didn't jump on it and say this is the way the Air Force should go, but until we get a better one, that's the one we're pushing. If a contractor follows 1679, he will make a high score on our maintainability questionnaire.

Mr. McOmber: The Navy is getting ready to issue a data call for updates to 1679. Are you interested in that?

Lt. Col. Blackledge: Sure. Send one to Col. Marciniak at RADC, too. I've already covered this pretty well. IV&V itself is also a DT&E tool. But, we feel that we in the OT&E community can gain by that experience that the IV&V'er built up over the life of the development of the program, if we can get him to identify for us what are the critical functions; perhaps when an aircraft goes into a particular climb, there is one module he thinks when looking at the software that "boy, if all these inputs come in at the same time, this little module is going to be overwhelmed and there's a possibility that it's going to degrade the output of it". OK, let us know that. What kind of things would come in? Well, if the pilot pulls back on the stick or whatever, then all these things happen at the same time. If you put that into your test scenario, you should be able to check out that particular critical function. That would help us, from all the infinite possibilities for test design, help us check out a couple that might tell us something.

Dr. Fischer: If you have a potential use for the work an IV&V contractor does, do you have any input into the evaluation of IV&V proposals?

Lt. Col. Blackledge: I thought you were going to sign us up to monitor IV&V. ... Yes, we do. What we found, of course, is that most systems don't do IV&V across the board. They do IV&V on a particular area like the mission data preparation system, or some particular area, some subset of the system. Our software test manager usually gets to take a look at what they are going to do on that area.

Maj. Hammond: This might be an appropriate time to point out that rather than your concept of having OT guys doing IV&V, current Air Force thinking is whoever is going to support that system do the IV&V. One of the reasons for that, is once the testers finish testing a system, that's the end of their involvement with it. There's no longterm payback period just in learning that software program. The supporting command is extremely active. So, the current letter guidelines from the Air Staff, that we are incorporating in our supplement to AFR 800-14, say that the program manager, once he has decided to do IV&V, set up criteria on how he makes that decision. He then gives primary consideration to whoever is going to support that, whether it's the operating command or the logistics command, or whoever. It doesn't have to go that way, but that's the way it's planned.

Lt. Col. Blackledge: Thank you for reminding me of that, Dave. There's an October 13th '81 Air Force RD/LE letter that makes IV&V a policy. I've got a few copies of it, if you'd like to get a copy from me. I have a few of them to pass out there. That, essentially, just went out with a 90 day suspense to Systems Command and Logistics Command to get back and tell Air Staff how they were going to set up their focal points and so on. That policy is something we have been pushing at AFTEC for a couple of years, and it has finally come about.

Maj. Hammond: And it's being included in the supplement to 800-14 virtually word for word.

Lt. Col. Blackledge: OK, in the supplement. We were pushing, hoping it would get into 800-14, itself, but apparently, that is not to be. [Editor's note: The latest revisions of AFR 800-14 will contain the IV&V policy.]

Mr. Watt: Does this mean that AFTEC will not be in on the IV&V, involved at all?

Lt. Col. Blackledge: Only as a strong advocate. What it says in there is that ideally IV&V would be run by the using command and the supporting command. If that can't work, then there should be some combination of the using command and the supporting command and an independent contractor or a federally contracted FCRC [RAND] type of outfit. But, AFTEC itself is not. General Leaf's position was that you'd have to double or triple his staff to really get into IV&V, and he wouldn't sign up for that. He was in favor of it but not for AFTEC to be running the show.

This kind of sums up what we've been talking about. We do put somebody out on each test team, and we try to tailor to each individual software evaluation on each program that we're working at. I've mentioned some examples as I went through. I also have some handouts of a paper that is going to be at NAECON, National Aerospace and Electronics Conference at Wright-Pat in May. I've got about 10 copies of that, which takes the program, goes through what I said, but gives it a detailed account of what was done on the air launch cruise missile competitive flyoff as far as the software evaluation goes. So, if you would like a copy of that, I've got about 10 of those. Any questions?

VIEWGRAPHS

USED BY

LT. COL. BLACKLEDGE

FOR THE

AIR FORCE

PRESENTATION

AIR FORCE TEST & EVALUATION CENTER

SOFTWARE TEST AND EVALUATION

SOFTWARE EVALUATION DIVISION

SOFTWARE TEST AND EVALUATION

- ● BACKGROUND
- ● AFTEC ORGANIZATION FOR SOFTWARE OT&E
- ● AFTEC APPROACH TO SOFTWARE OT&E
- ● FUTURE EFFORTS IN SOFTWARE OT&E

DEFINITION

SOFTWARE - A SET OF COMPUTER PROGRAMS, PROCEDURES, AND ASSOCIATED DOCUMENTATION CONCERNED WITH THE OPERATION OF A DATA PROCESSING SYSTEM

DEFINITION

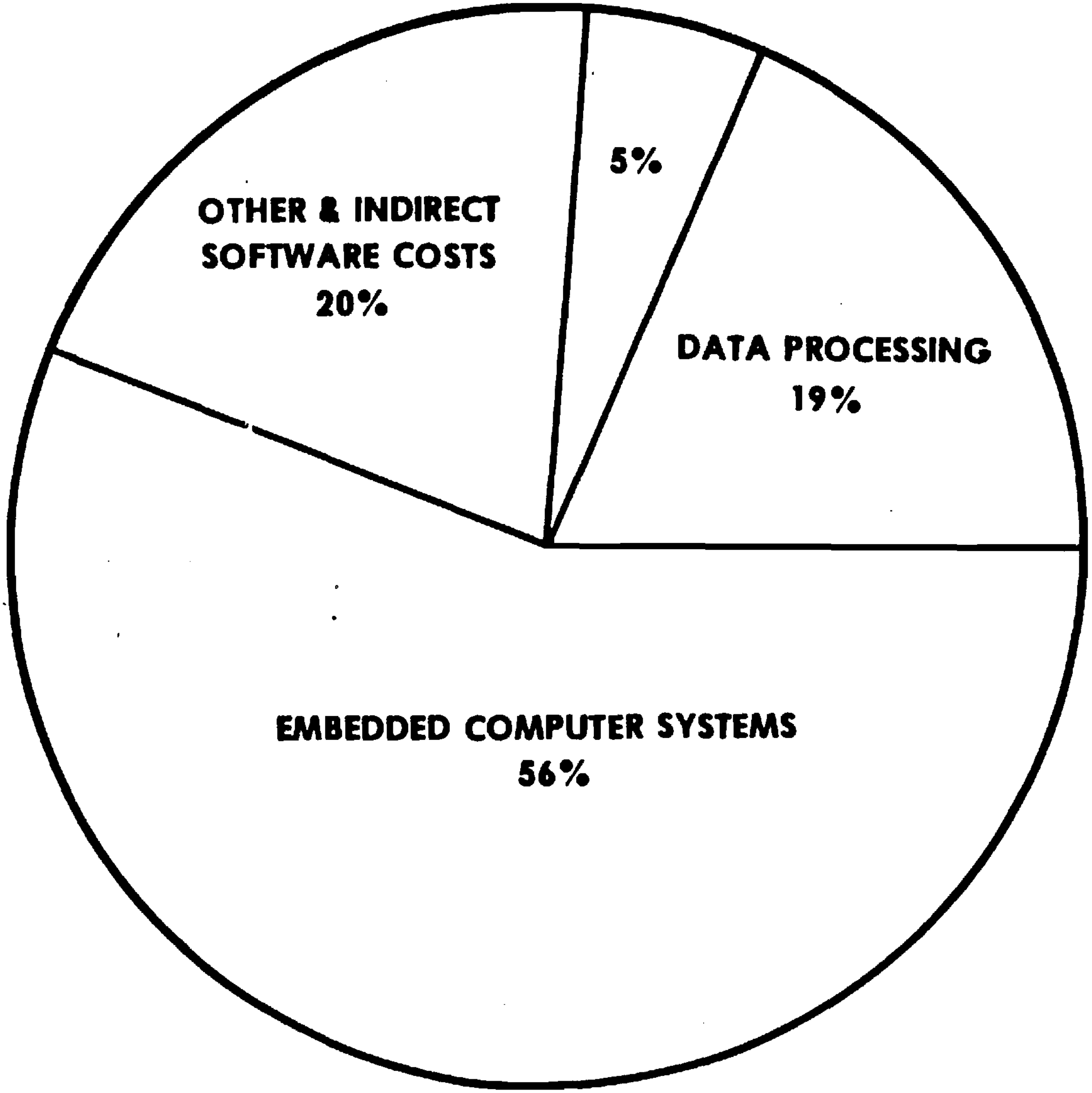
EMBEDDED COMPUTER SYSTEM - A COMPUTER SYSTEM THAT IS:

- •PHYSICALLY INCORPORATED INTO A LARGER SYSTEM WHOSE PRIMARY FUNCTION IS NOT DATA PROCESSING
- •INTEGRAL TO A LARGER SYSTEM FROM A DESIGN, PROCUREMENT, AND OPERATIONS VIEWPOINT

EMBEDDED COMPUTER SYSTEMS SOFTWARE PROBLEMS

- RISING COSTS
- DEVELOPMENT ERRORS

ANNUAL DOD SOFTWARE COSTS

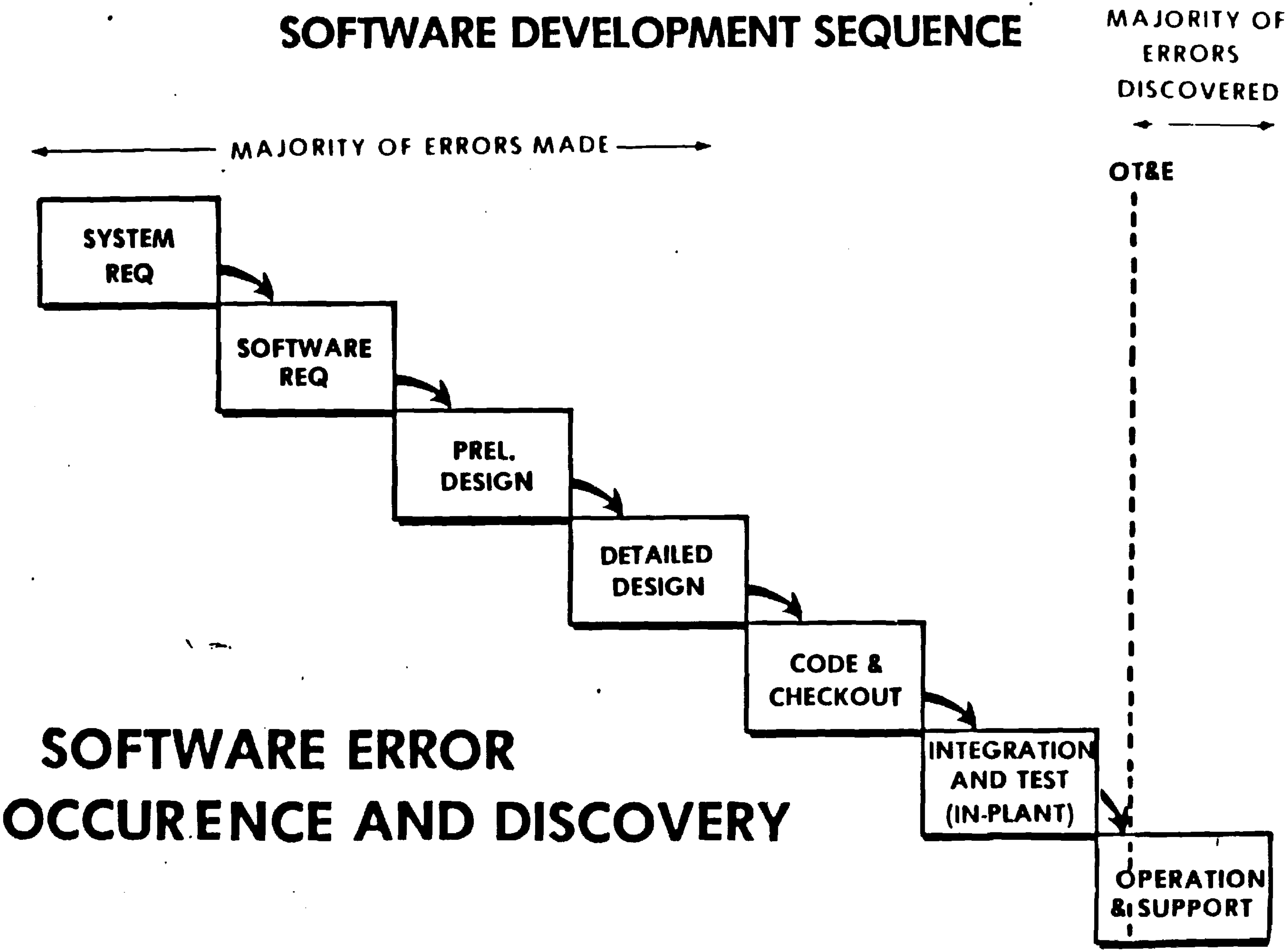


THE 1974 DEFENSE SCIENCE TASK FORCE ON TEST AND EVALUATION

THE TASK FORCE FOUND THAT:

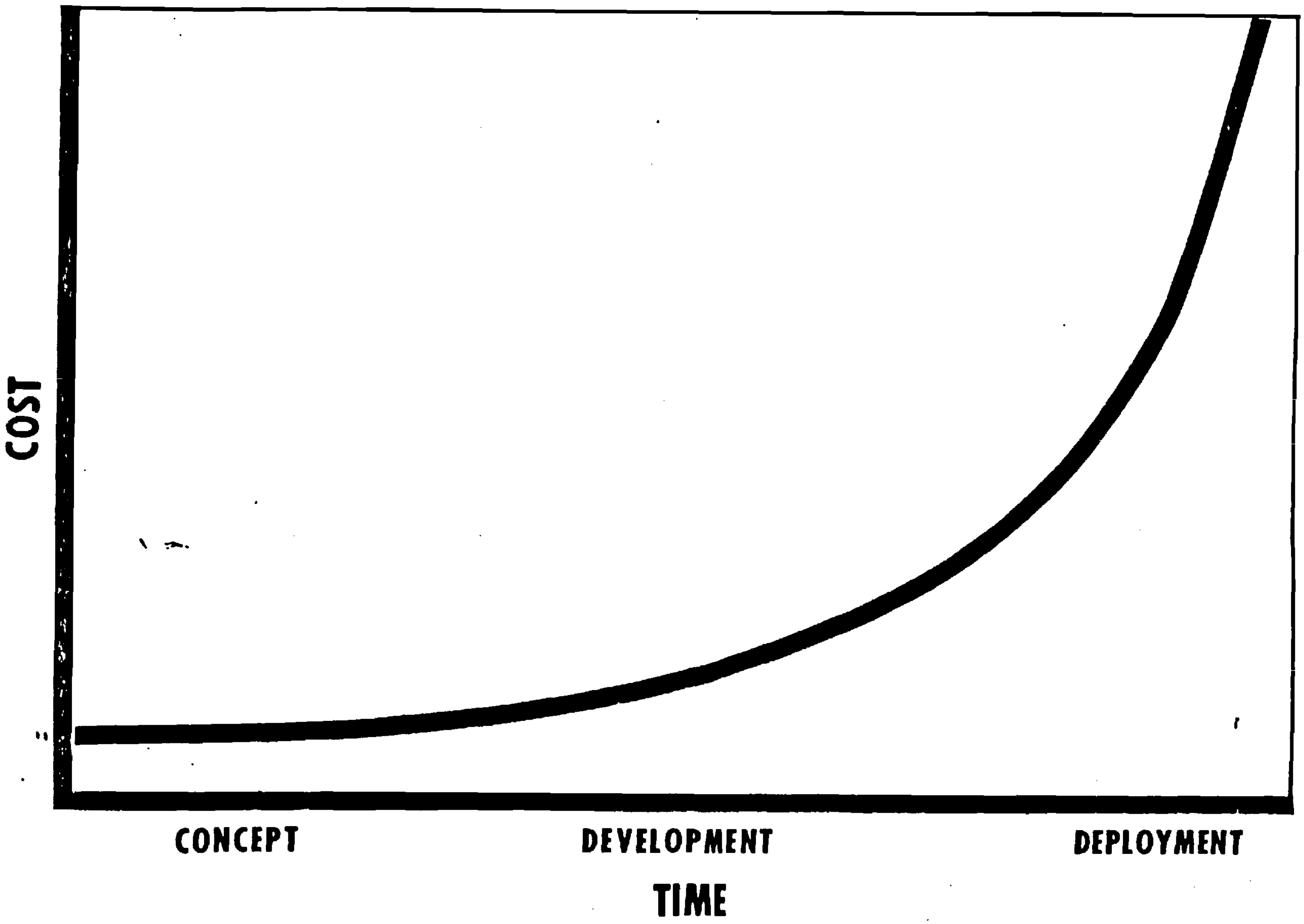
..................."WHEREAS THE HARDWARE DEVELOPMENT WAS FOR THE MOST PART SCHEDULED, MONITORED, TESTED, AND REGULARLY EVALUATED, THE SOFTWARE DEVELOPMENT WAS NOT."

SOFTWARE DEVELOPMENT SEQUENCE



SOFTWARE ERROR OCCURENCE AND DISCOVERY

TEST EARLY TO SAVE COSTS



EXCERPTS FROM DODD 5000.3
(26 DEC 79)

...PROVISIONS APPLY TO SOFTWARE COMPONENTS ...AS WELL AS HARDWARE COMPONENTS

...PERFORMANCE OBJECTIVES SHALL BE ESTABLISHED FOR SOFTWARE DURING EACH SYSTEM ACQUISITION PHASE...

...DECISIONS TO PROCEED FROM ONE PHASE OF SOFTWARE DEVELOPMENT TO THE NEXT WILL BE BASED ON... APPROPRIATE T&E.

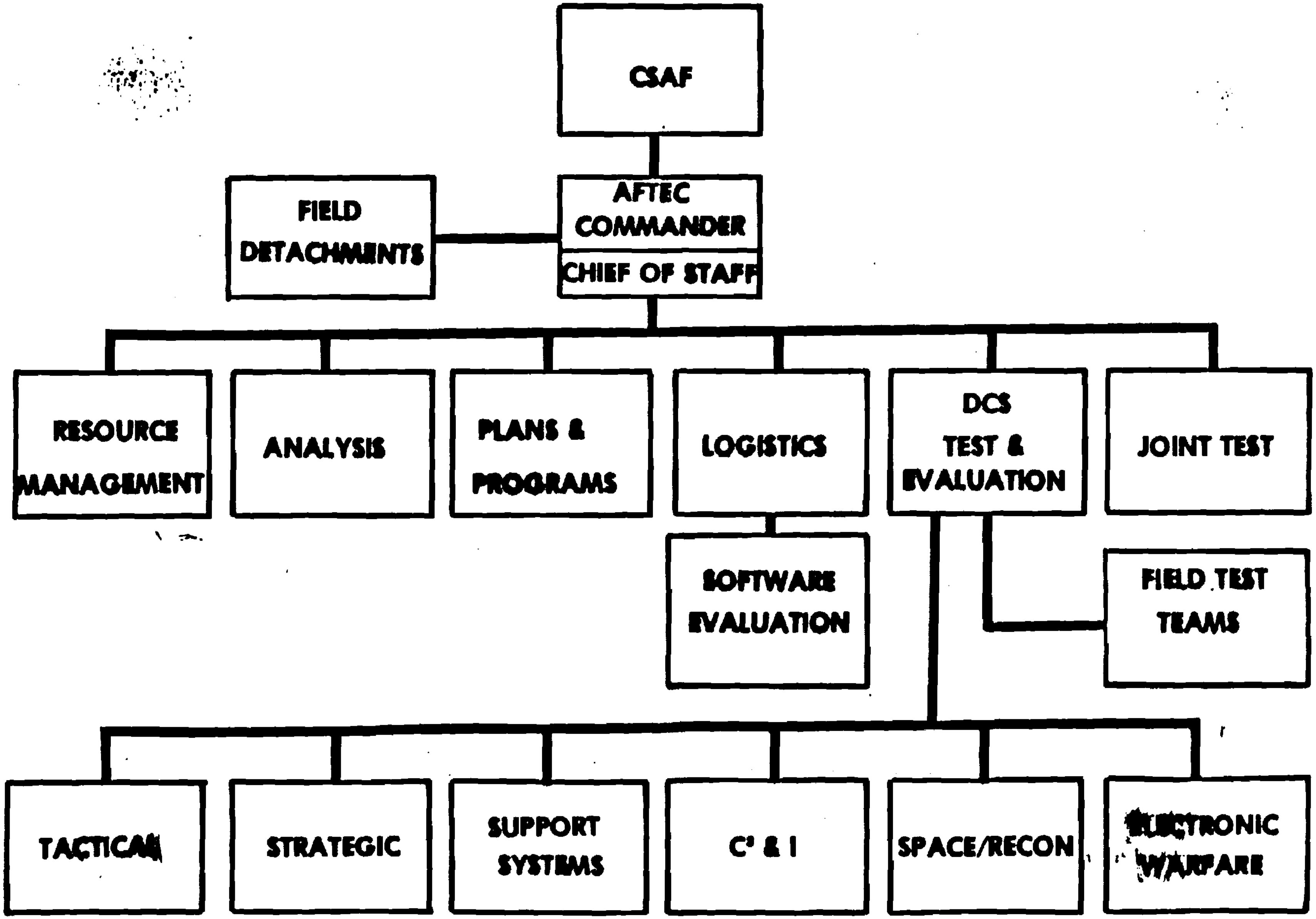
...SOFTWARE...SHALL UNDERGO OPERATIONAL TESTING...UTILIZING TYPICAL OPERATOR PERSONNEL.

...OT&E AGENCIES SHALL PARTICIPATE IN SOFTWARE PLANNING AND DEVELOPMENT TO ENSURE CONSIDERATION (OF THE) OPERATIONAL ENVIRONMENT AND EARLY DEVELOPMENT OF OPERATIONAL TEST OBJECTIVES.

SOFTWARE TEST AND EVALUATION

- BACKGROUND
- AFTEC ORGANIZATION FOR SOFTWARE OT&E
- AFTEC APPROACH TO SOFTWARE OT&E
- FUTURE EFFORTS IN SOFTWARE OT&E

AFTEC ORGANIZATIONAL RELATIONSHIPS



CSAF

AFTEC COMMANDER

CHIEF OF STAFF

FIELD DETACHMENTS

RESOURCE MANAGEMENT

ANALYSIS

PLANS & PROGRAMS

LOGISTICS

SOFTWARE EVALUATION

DCS TEST & EVALUATION

FIELD TEST TEAMS

JOINT TEST

TACTICAL

STRATEGIC

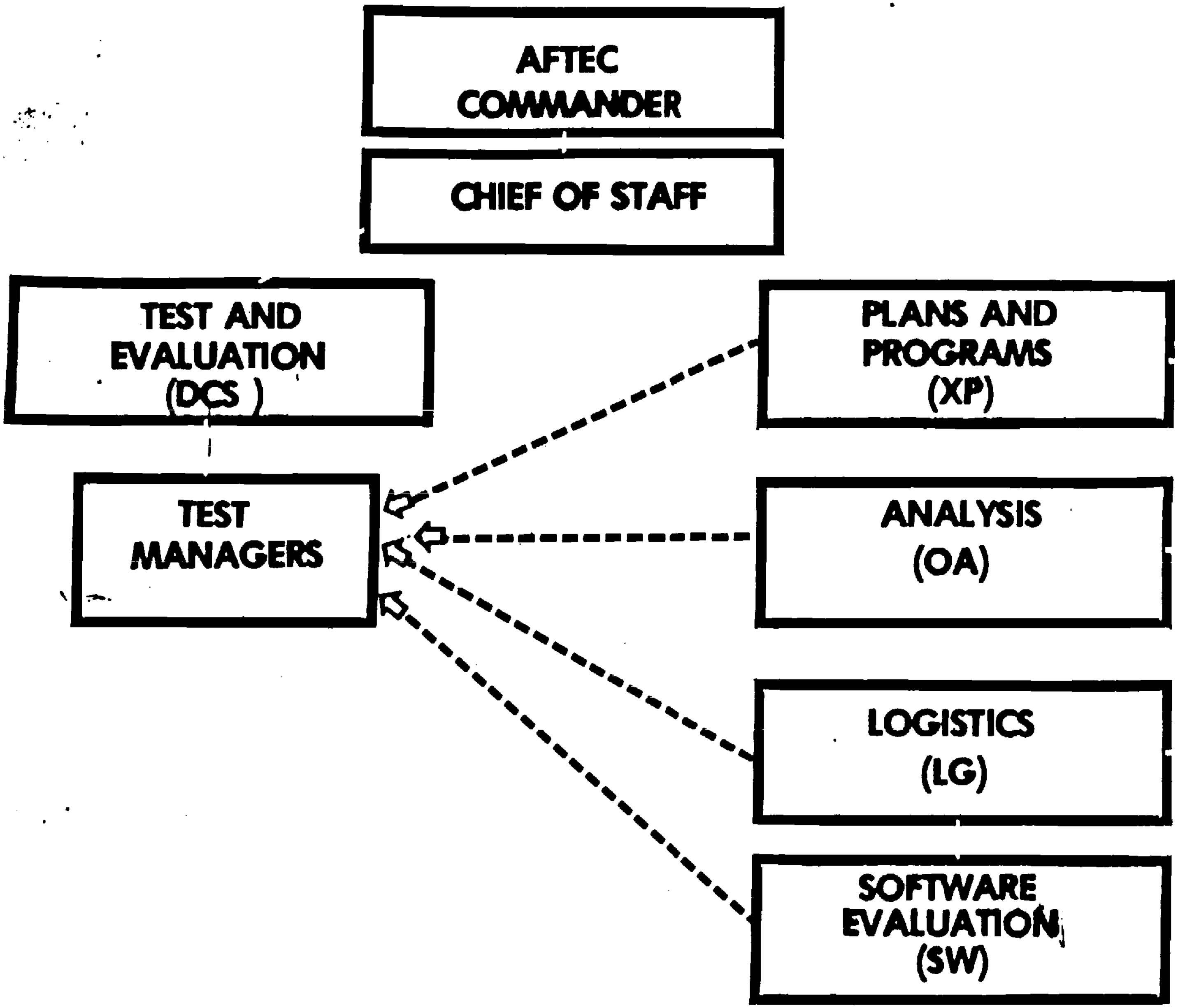
SUPPORT SYSTEMS

C^3 & I

SPACE/RECON

ELECTRONIC WARFARE

TEST MANAGEMENT APPROACH
(AFTEC HQ ELEMENTS)



AFTEC COMMANDER

CHIEF OF STAFF

TEST AND EVALUATION (DCS)

TEST MANAGERS

PLANS AND PROGRAMS (XP)

ANALYSIS (OA)

LOGISTICS (LG)

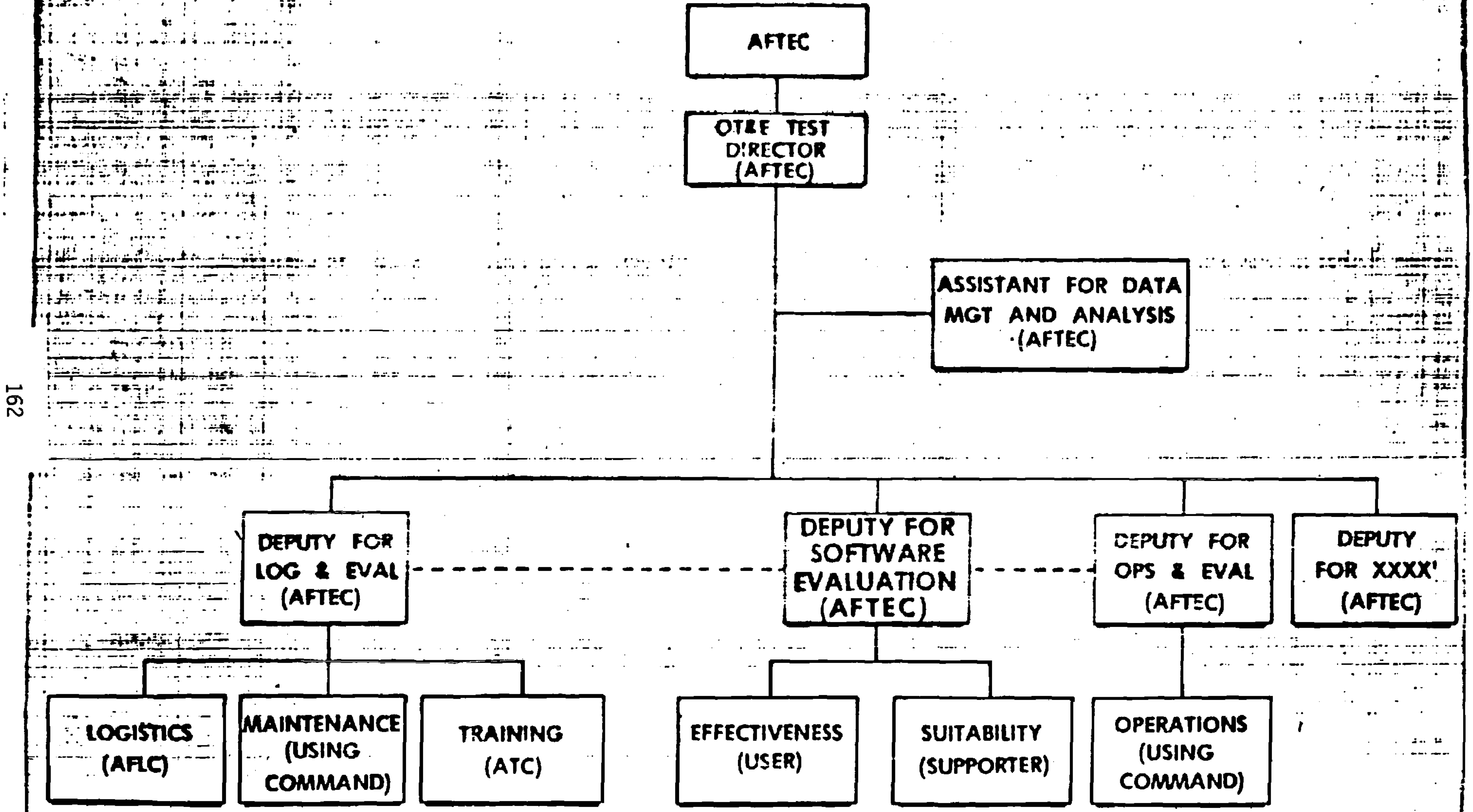
SOFTWARE EVALUATION (SW)

SOFTWARE EVALUATION

FUNCTIONAL SPECIALTIES

- ●AVIONICS/ELECTRONIC WARFARE SYSTEMS
- ●SPACE/MISSILE SYSTEMS
- ●C^3 SYSTEMS
- ●AIRCREW TRAINING DEVICES
- ●AUTOMATIC TEST EQUIPMENT
- ●DATA AUTOMATION SYSTEMS

OT&E TEST TEAM COMPOSITION


AFTEC

OT&E TEST DIRECTOR (AFTEC)

ASSISTANT FOR DATA MGT AND ANALYSIS (AFTEC)

DEPUTY FOR LOG & EVAL (AFTEC)

DEPUTY FOR SOFTWARE EVALUATION (AFTEC)

DEPUTY FOR OPS & EVAL (AFTEC)

DEPUTY FOR XXXX' (AFTEC)

LOGISTICS (AFLC)

MAINTENANCE (USING COMMAND)

TRAINING (ATC)

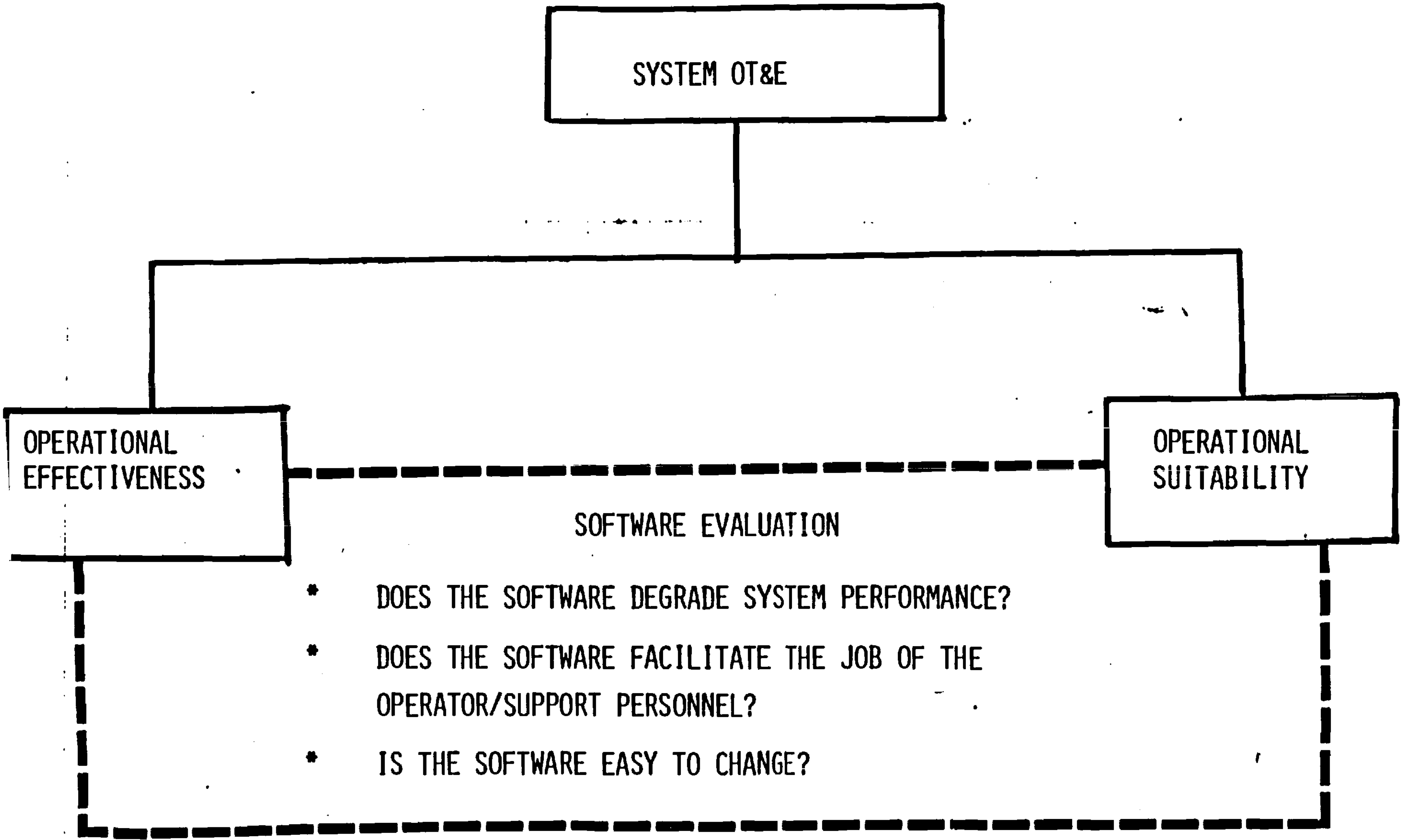
EFFECTIVENESS (USER)

SUITABILITY (SUPPORTER)

OPERATIONS (USING COMMAND)

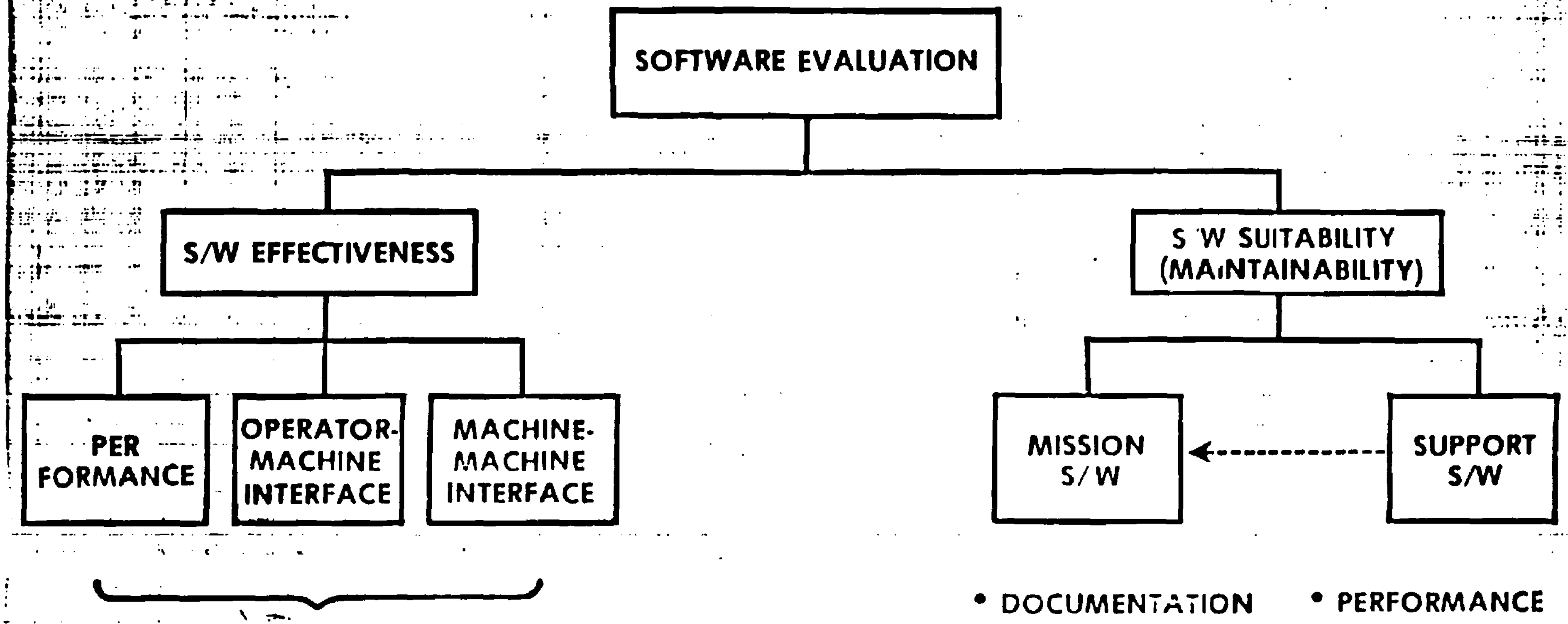
SOFTWARE EVALUATION

- ● TEST PREPARATION
  - •EARLY PLANNING WITH IMPLEMENTING, USING, SUPPORTING AGENCIES
  - •PREPARE OBJECTIVES, MEASURES, METHODOLOGY
  - •DESIGN REVIEWS, CRWG, TPWG
- ● TEST CONDUCT
  - •IN-PLANT TESTING
  - •ON-SITE TESTING
- ● EVALUATION
  - •TEST DATA ANALYSIS
  - •TEST DATA EVALUATION
  - •REPORT PREPARATION



SOFTWARE EVALUATION

- DOES THE SOFTWARE DEGRADE SYSTEM PERFORMANCE?
- DOES THE SOFTWARE FACILITATE THE JOB OF THE OPERATOR/SUPPORT PERSONNEL?
- IS THE SOFTWARE EASY TO CHANGE?



SOFTWARE EVALUATION

S/W EFFECTIVENESS

PER FORMANCE

OPERATOR-MACHINE INTERFACE

MACHINE-MACHINE INTERFACE

S W SUITABILITY (MAıNTAINABILITY)

MISSION S/ W

SUPPORT S/W

OF THE MISSION S/W

- DOCUMENTATION REVIEW

- PERFORMANCE
- OPERATOR-MACHINE INTERFACE
- DOCUMENTA TION REVIEW

DEFINITION

SOFTWARE MAINTAINABILITY - A MEASURE OF THE EASE WITH WHICH SOFTWARE CAN BE CHANGED

REASONS FOR CHANGE:

- CORRECT ERRORS
- ADD OR DELETE SYSTEM CAPABILITIES
- MODIFY SOFTWARE BECAUSE OF HARDWARE CHANGES

DEFINITION

USABILITY - THE EXTENT TO WHICH SOFTWARE DESIGNATED TO PERFORM A SUPPORT FUNCTION IS EFFECTIVE IN PERFORMING THAT FUNCTION AND IS USABLE BY THE AIR FORCE OPERATOR.

CONSIDERATIONS:

- FUNCTIONAL PERFORMANCE
- OPERATOR-MACHINE INTERFACE
- PERSONNEL/TRAINING REQUIREMENTS

SOFTWARE SUITABILITY OBJECTIVES EXAMPLE

OBJECTIVE 11. EVALUATE THE OPERATIONAL SUITABILITY OF THE CNCE SOFTWARE SYSTEMS.

(A) SUBOBJECTIVE 11-1. EVALUATE THE OPERATIONAL SOFTWARE FOR MAINTAINABILITY

(B) SUBOBJECTIVE 11-2. EVALUATE AVAILABLE SOFTWARE SUPPORT RESOURCES

} MAINTAINABILITY

(C) SUBOBJECTIVE 11-3. EVALUATE THE ADEQUACY OF OFF-LINE DIAGNOSTICS TO DETECT AND ISOLATE MALFUNCTIONS IN A TIMELY MANNER

} USABILITY

METHODOLOGY

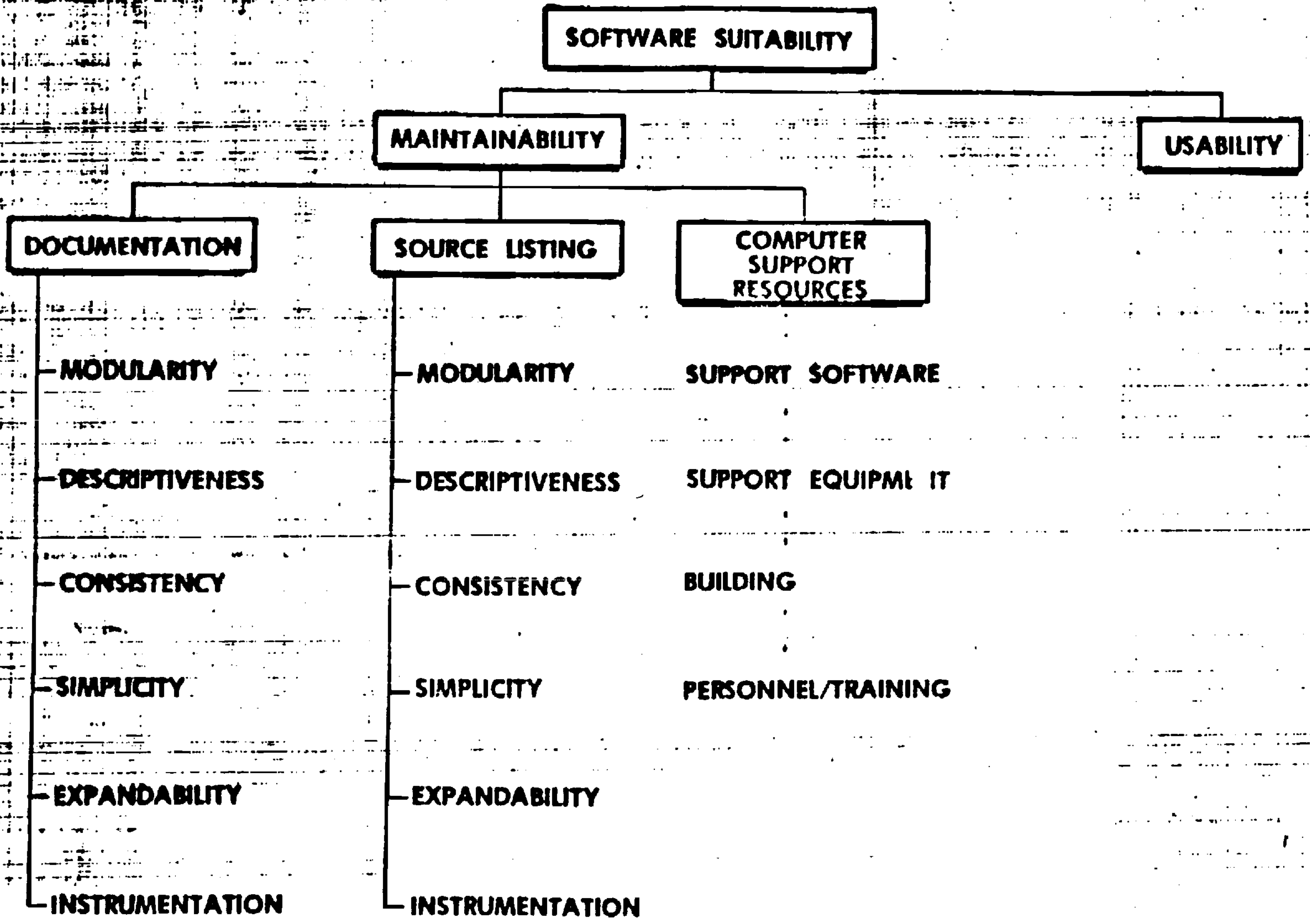
- QUANTIFY SUBJECTIVE EVALUATION

- SEPARATE EVALUATION OF DOCUMENTATION AND SOURCE LISTINGS

- STANDARD QUESTIONNAIRES FOR ALL SOFTWARE

- MEASURES PRESENCE OF DESIRABLE MAINTAINABILITY CHARACTERISTICS

- TECHNIQUE IN USE FOR SEVERAL YEARS



SOFTWARE SUITABILITY

MAINTAINABILITY

USABILITY

DOCUMENTATION
- MODULARITY
- DESCRIPTIVENESS
- CONSISTENCY
- SIMPLICITY
- EXPANDABILITY
- INSTRUMENTATION

SOURCE LISTING
- MODULARITY
- DESCRIPTIVENESS
- CONSISTENCY
- SIMPLICITY
- EXPANDABILITY
- INSTRUMENTATION

COMPUTER SUPPORT RESOURCES
- SUPPORT SOFTWARE
- SUPPORT EQUIPMENT
- BUILDING
- PERSONNEL/TRAINING

MAN-MACHINE INTERFACE EVALUATION METHODOLOGY

- **APPROACH:**
  - SAME AS MAINTAINABILITY
  - PROVISIONS FOR UNIQUE APPLICATIONS
  - OPERATOR VS SUPPORTER ORIENTED

- **STATUS:**
  - TEST FACTORS IDENTIFIED
  - QUESTIONNAIRE WRITTEN
  - DRAFT EVALUATOR'S HANDBOOK AVAILABLE

SOFTWARE OPERATOR-MACHINE INTERFACE

EXAMPLES:

- OPERATOR INPUT ERRORS DO NOT CAUSE SYSTEM FAILURES.
- THE SYSTEM SOFTWARE MAY BE RELOADED QUICKLY AND EASILY.
- MENU TECHNIQUES ARE USED TO AID THE OPERATOR IN MAKING DECISIONS.
- LEGITIMATE RESPONSES FOR ALL CONDITIONS ARE DOCUMENTED AND/OR PROMPTED BY THE SOFTWARE.
- MESSAGES REQUIRING ACTION BY THE OPERATOR ARE ALWAYS HIGHLIGHTED IN SOME FASHION.
- OPERATOR ENTERED INSTRUCTIONS ARE RELATIVELY SHORT.

SOFTWARE EFFECTIVENESS OBJECTIVES

EXAMPLE

OBJECTIVE 5. EVALUATE THE IMPACT OF THE CNCE SOFTWARE SYSTEM ON THE OPERATIONAL EFFECTIVENESS OF THE SYSTEM.

(B) SUBOBJECTIVE 5-2. EVALUATE THE ABILITY OF THE CNCE OPERATIONAL SOFTWARE SUBSYSTEM TO PROCESS NODAL TRAFFIC IN A TIMELY MANNER	PERFORMANCE
(F) SUBOBJECTIVE 5-6. EVALUATE THE INTEROPERABILITY OF THE CNCE SOFTWARE SUBSYSTEM WITH OTHER NETWORK SOFTWARE ELEMENTS	MACHINE-MACHINE INTERFACE
(H) SUBOBJECTIVE 5-8. EVALUATE THE CAPABILITY OF THE CNCE SOFTWARE SUBSYSTEM TO ASSIST THE CONTROLLER IN PERFORMING HIS RECORD-KEEPING FUNCTIONS	MAN-MACHINE INTERFACE

SOFTWARE EFFECTIVENESS EVALUATION

- SYSTEM CONTEXT
- FOCUS ON CRITICAL PATHS
- FULL SYSTEM/CASUALTY MODE OPERATION
- DETAILS OF LOGiC EXECUTION

SOFTWARE TEST AND EVALUATION

- BACKGROUND
- AFTEC ORGANIZATION FOR SOFTWARE OT&E
- AFTEC APPROACH TO SOFTWARE OT&E
- FUTURE EFFORTS IN SOFTWARE OT&E

PRESENT SOFTWARE TESTING SHORTFALLS

- LITTLE ASSURANCE THAT CRITICAL FUNCTIONS ARE EXERCISED
- DEFICIENCIES DISCOVERED LATE - COSTLY TO CORRECT
- SOFTWARE SUPPORT RESOURCES NOT AVAILABLE FOR EVALUATION
- SOFTWARE ENGINEERS NOT AVAILABLE FOR EVALUATION

IMPROVED TEST METHODOLOGY

- EVENT TRACE MONITOR
- ADDITIONAL STANDARDIZED TEST TOOLS
- MORE IV&V APPLICATIONS

SOFTWARE DESIGN TECHNIQUES

- MORE STRUCTURED, DISCIPLINED
- TOP DOWN PROGRAMMING
- HIGHER ORDER LANGUAGES

INDEPENDENT VERIFICATION AND VALIDATION (IV&V)

- FUNCTIONS IN OPERATIONAL ENVIRONMENT
- USEFUL DT&E TOOL (ANALYTIC ENVIRONMENT)
- APPLY TO OT&E
  - PROVIDES EARLY DATA
  - IDENTIFY CRITICAL PATHS

SUMMARY

- SOFTWARE MUST BE EVALUATED (EFFECTIVENESS AND SUITABILITY)
- SOFTWARE EVALUATION EXPERTISE ON TEST TEAM
- AFTEC TAILORS SOFTWARE OT&E TO EACH PROGRAM

Mr. Greenlee: Thanks, Mike. To complete our presentations by military components, we will have a presentation from the Defense Communications Agency. There are numerous defense agencies outside the Services which are involved in computers and their use, NSA, DIA, etc. Probably none is a bigger consumer of software than the DCA. To compound that, it seems like our comm programs are the ones that frequently have most substantive issues involving software and software testing. So, here to talk a little bit about one portion of the DCA point of view is Mrs. Caral Giammo. She has charge of the software testing for WWMCS at the DCA facility out in Reston, VA. Caral?

MRS. CARAL GIAMMO: DEFENSE COMMUNICATIONS AGENCY

Mrs. Giammo: I think what you are going to hear from me about software testing is a little bit different than what the other people have been saying because the software we test is based on commercially supplied, general purpose software. Most of what has been talked about is embedded software where the environment and the range of the use tends to be more restrictive. The software itself is easier to use and easier to specify than general purpose software, and we are involved with general purpose software. I've been listening, and everyone seems to assume near perfect specifications that are comprehensive, accurate, and precise and that, therefore, the fallibility lies in the implementation. I work in an environment where the specifications are probably nonexistant and the fallibility lies everywhere. I was interested in the broadening of the scope of the definition of embedded software to mission critical because you will see that our software is indeed mission critical. The area I'm with, and that the Defense Communications Agency plays a very big part in, is in the Worldwide Military Command and Control System - WWMCCS. We are in WWMCCS ADP which is 7% of WWMCCS. WWMCCS ADP is small yet it is that part of WWMCCS that makes the headlines, which Jack Anderson writes about, which the House Appropriations Committee chastises, and which is constantly being investigated by GAO. We have high visibility. We even had 7 minutes on Walter Cronkite.

The Defense Communications Agency is very heavily involved in WWMCCS. WWMCCS is slightly different from the systems previously discussed. We support all Services, JCS, unified and specified commands, and NATO. So, the systems that I'm going to talk about are extremely general purpose because we have such a mix of users. It is indeed mission critical.

The WWMCCS ADP system is what I'm going to talk about and what they look like today. WWMCCS ADP started in 1971 when the hardware buy was made. The basic computer is a Honeywell 6000 frontended by a Datanet 355. There are within this system at least 4 different type processors within the WWMCCS community. A site may have from 1-4 processors. Some have more. This slide is just a general picture. Some have more than one Datanet. Most have more than one Datanet. There are at least 5 different kinds of disk drives in the community and 5 kinds of tape drives in the community. The H6000 and Datanet was the original buy. In about 1974, people discovered that they really had to access 2 computers from remote sites, and they needed network processing capabilities. At that time, the Honeywell 716 (700, 725) was put onto the WWMCCS contract. I'd like to point out here that the software that operates in the Honeywell 6000 is basically the Honeywell commercial software. If you go to IBM and buy a computer, you get a whole pile of software that comes for free. In WWMCCS, if you buy a computer, if you become a WWMCCS site as the Defense Nuclear Agency has just become, the computer is from Honeywell, but the software is from the Defense Communications Agency -- operating systems, compilers, data management systems, the whole set that is normally free.

In addition, if you were buying from IBM and you were in the banking industry, you might decide to buy from IBM a set of software to handle banking applications. In WWMCCS, if you would like to have a joint reporting system, you also get that from the Defense Communications Agency. We supply to the WWMCCS community 23 standard applications. We're dealing with not only applications but systems software. And the systems software, as you'll see, keeps growing. Over 20 million lines of code.

The next thing that happened was that the Honeywell 700 became obsolete in Honeywell's product line and was replaced by the Level 6 computer. Again, the users are accessing hosts over communications lines. Computers in Panama are talking to computers at Tampa and one in Alaska to Cheyenne Mountain. Our computers are all over the world. WWMCCS ADP covers 17 time zones. We moved from stand alone hosts with remoted mini's, which is easy, to internetting our computers. We run a computer network, an ARPA type computer net. Twenty-two of the WWMCCS sites are internetted and NATO runs their own net of 3 computers. Our front end is an Interface Message Processor built on the Honeywell 716. Our communication lines also involve satellite links. We have 6 computer types within WWMCCS now, all one vendor. In a few months, we will have 7 computer types from 2 vendors.

The internetting is called the WWMCCS Intercomputer Network. This slide is the topology as of March 1981. You can see that we have Air Force sites in the Pacific (PACAF) with Korea internetted by a satellite into an IMP at PACAF and PACOM, coming back over satellites into MAC. SAC will be coming into the intercomputer network in a couple of months. The network runs all the way to Europe. The network goes as far south, which I don't believe is on here, to Panama which goes to REDCOM. From the north, we have the Alaskan Air Command. This is the software that I'm responsible for testing. We have 14 people who test the system software that is the basic operating system, communications, and network software for this network.

This next slide is from the WWMCCS modernization study. It is here to show you the diversity of our users. In looking at the WWMCCS modernization, the WWMCCS System Engineer found that there were 26 independent users of the system. They excluded NATO, the 4 Air Force Major Commands that are part of WWMCCS, and the early warning systems. Note the variance in the number of processors that the sites have and the types of applications. Our users are everyone from MAC who runs an airline reservation system to SAC to MTAC, the military transport command. The last one is our facility in Reston. That is the scope of applications.

Some interesting things come out of it, I think, that makes us a little bit different from previous presenters. We have 20 million lines of code on the systems software side, the bulk of which is commercial off-the-shelf software. If you have ever had to deal with commercial off-the-shelf software done to best practices, you know the problems. What kind of testing do you do? We view our role as everytime the vendor comes out with a new release of software, our role is to get the belly-ache rather than our 42 WWMCCS sites all getting the same belly-ache.

We have embedded into this software WWMCCS uniques, mainly in the security area. These are embedded in the software and permeate all of the software. We have, in addition, about 117 additional WWMCCS hang-ons. They are requirements of the users after having used the software and deciding that they needed something added. The biggest WWMCCS unique is the software for the WWMCCS Intercomputer Network. Our network handles file transfers and teleconferencing. Thus, we have a full range of software. I think the key here is that we are dealing with general purpose software. We don't know how our users will be using the software. Our users are the computer operators, the system software people, the application programmers, and the action officers who sit in a command post, and who have to get into the strategic systems or into the joint operational planning system. All of this is done through the WWMCCS Intercomputer Network. We have a heterogeneous user community both in the types of applications that they are using and the experience level, from very experienced computer people to very inexperienced people, and totally uncontrolled use. Another area of difference is that we have multiple vendors -- multiple hardware vendors and multiple software vendors. Even in the standard applications area, we have multiple vendors. So, it is a slightly different environment.

Instead of general requirements, I started to put motherhood requirements because that is about the only requirement that we have. We don't have well defined, unambiguous specifications. Our requirements are that the WWMCCS Intercomputer Network should be 99% reliable. Now, how can something which is over communications lines, modems, cryptos, etc. have 99% reliability with no back-up? Even if the reliability of each component is .99, by the time 100 components are hooked together, you have a very small reliability. Some requirements are conflicting. For example, two that came out of Congress recently said the WWMCCS is obsolete. Then it said, next time buy commercial, off-the-shelf software without bells and whistles. And, by the way, make sure it handles multilevel security. And, the last difference is that we have continual change. If the vendor, Honeywell, isn't changing the operating system on the H6000, then the Level 6 operating system is changing. We are continually finding errors in the systems. Our applications are updated twice a year. The sites are continually changing their use of the system. What do we do in this kind of an environment?

We use all of the techniques that everyone has spoken about. We get in early when we can. We obviously can't tell Honeywell how they should do their software development. Given that we have the authority, given that we have the dollars, given that we have the personnel, we get in when we can. We do component tests, very extensive component tests. We have a large range of test programs. We have the ability to choose tests based upon the changed parts of the operating system or different pieces of the equipment. We have gone into end to end tests, systems tests. The biggest testing is with the WWMCCS Intercomputer Network. If a user can't get into the system, who does he call? An exercise falls apart, who gets blamed? The only place where there is any system expertise in WWMCCS is within DCA. We're the people who have to try and do the post mortem and find the problem. The big question is where does software testing stop? When a system is more than hardware and software, who solves the problem?

We just went through a very expensive and time consuming problem solving exercise. Our site in Korea was running much slower than any other site. The first problem was that they didn't know they had a problem until my boss went out there and signed on a terminal. He tried to get into a teleconference and waited 5 minutes. He said, "this is not the way it is supposed to be". We have a small test facility with the ability to simulate satellite circuits or to go over live satellite circuits. We found out theoretically in an unloaded system what kind of throughput that Korea should have. Messages were sent back and forth -- check this, check that. The DCA Operations Center that controls the circuits checked the circuits. Checks occurred through the tech control facilities in Hawaii and the tech control facility in Korea. The problem could not be found. We finally put 3 people on an airplane with datascopes, and they started in Hawaii tracing the circuits. We believed that the problem was one of the tail circuits. The Hawaii circuits were clean. They went to Korea. The problem was a $1.50 wire, a grounding wire. It had been cut. No matter how many times that field engineer changed the boards, which had been done many times, the board still wasn't grounded. It was causing the errors -- retransmissions, loss of data, and the slowness. Is that a software problem? I don't know, but the user came to the software people to find the problem. That is what we are seeing today. Test personnel have to be multiskilled, and the real critical area is the multi-discipline ADP/communications specialist.

I would like to give to Dr. Fischer a few more problem areas. One is system test tools for computers and communications. Where there are mixed systems, you need systems test tools. Another area is system performance, I rarely see in software that is being developed that the requirement to be built into the software data collection tools which will help in the performance evaluation or in even performance data collection. The vendors don't do it in their operating systems.. This data would help in testing systems. The last problem area is diagnostic capabilities without taking the system off-line. We've got that big intercomputer network, and something is going bad. How can we find it? Loop backs through the comm circuits? Through the IMP's? Go into the Datanet maybe and into the 6000 and back to a bad terminal? But, you have to be able to do that without taking that site off-line. You have got to keep the computers up because the operational data is flowing.

That is all I have. Does anyone have any questions?

Dr. Fischer: The representatives from the 3 services distinguish operational tests from development tests, where development tests test to the engineering specifications and operational tests test to the users' needs. The testing you described„ what would you classify that as?

Mrs. Giammo: Well, we don't have any specs first of all. And, I'm very serious about that.

Dr. Fischer: Why?

Mrs. Giammo: We are getting commercial, off-the-shelf software. Do you ever get a spec from IBM? WIN I have specs for, not very good ones sometimes, but specs. If we have specs, we test to the specs. If we don't, we use the users manuals that the vendor provides. In the WIN, we have specs we test for. But the main thing we test to is the end user. Here is the user document, these are the capabilities he has. Does he really have those? The slide that I have left up there is that we ensure that the system operates as the end user believes the system will operate, that the documentation clearly explains the system to the user, and that any capabilities that he had in the previous version of the software still exists or the change is documented. We do a great deal of regression testing. If I have to fit somewhere, I have to fit in the operational testing. If we can get into the development phase, which we do with the WIN software, we do. I fit more in that user end. Any other questions?

VIEWGRAPHS

USED BY

MRS. GIAMMO

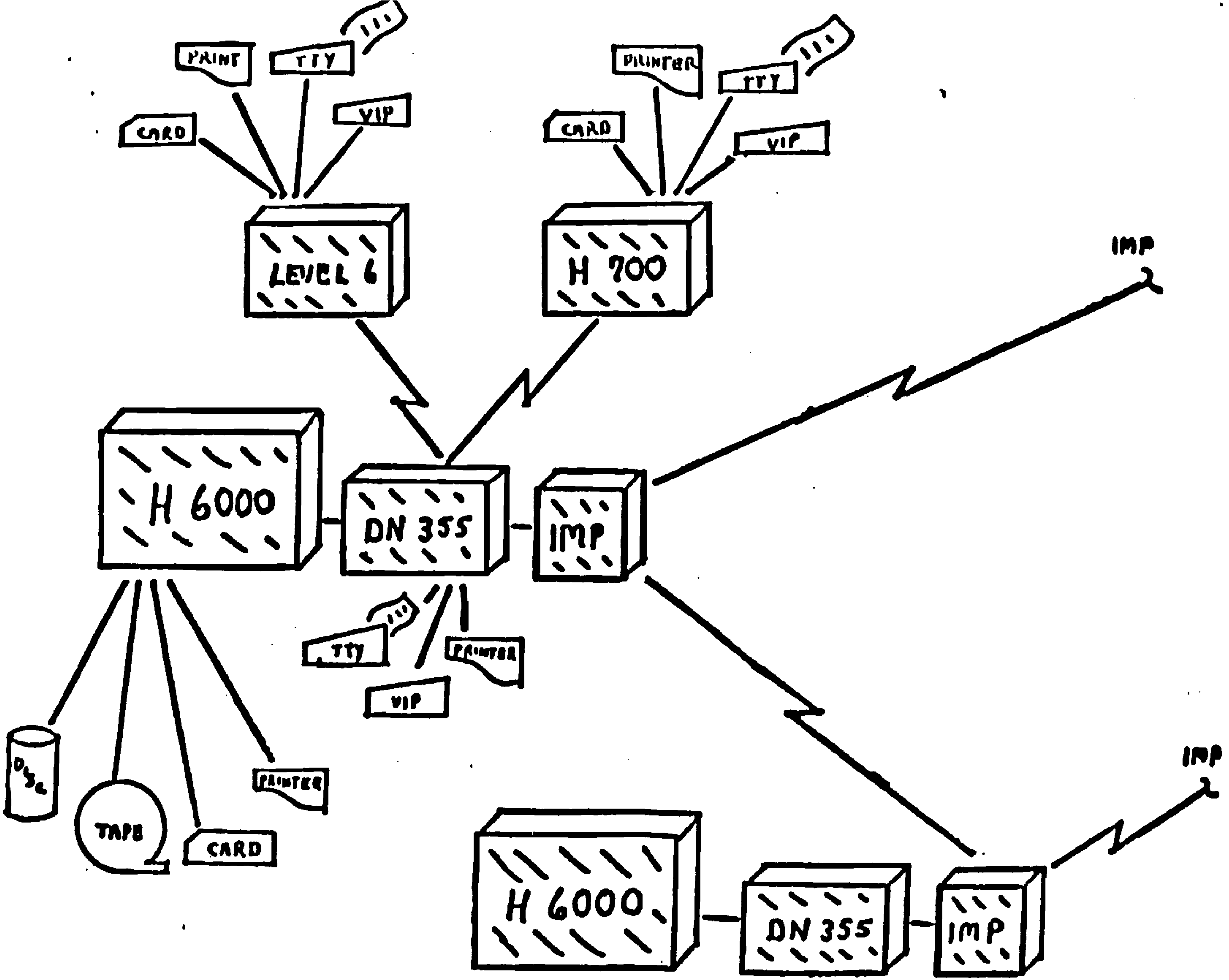
FOR THE

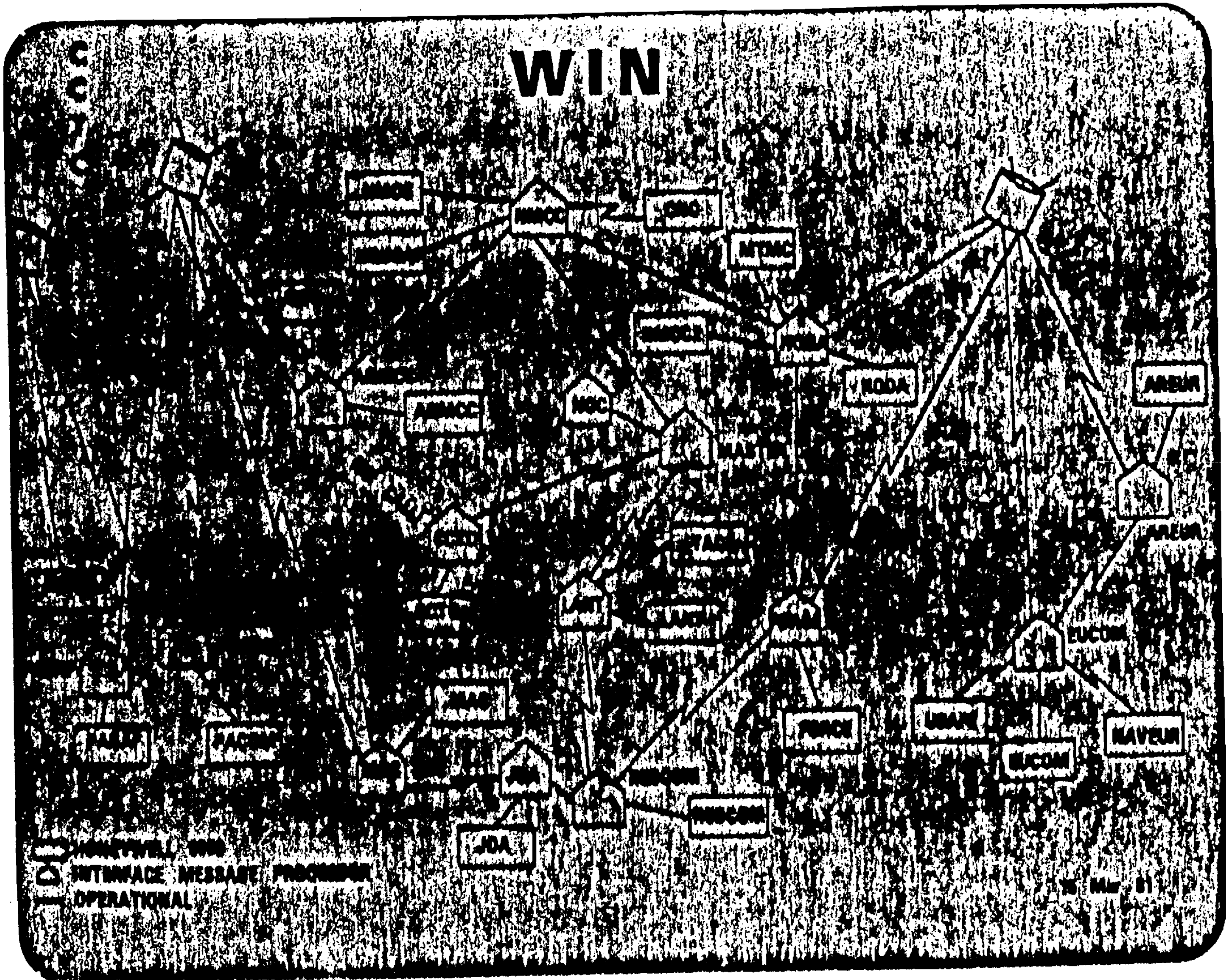
DCA

PRESENTATION

TEST AND EVALUATION OF SYSTEM SOFTWARE FOR THE WWMCCS STANDARD COMPUTERS

Slide 1


PRINT
TTY
CARD
VIP
LEVEL 6
PRINTER
TTY
CARD
VIP
H 700
IMP
H 6000
DN 355
IMP
TTY
PRINTER
VIP
Disc
TAPE
CARD
PRINTER
IMP
H 6000
DN 355
IMP


WIN
NMCC
NMIC
EUCOM
EUCOM
NAVEUR
JDA
INTERFACE MESSAGE PROCESSOR
OPERATIONAL

Slide 3

COMMAND LEVEL	SITE	LOCATION	TOTAL CPUs BY SITE	CONFIGURATIONS	CPUs PER CONFIGURATION
JCS	NMCC	Pentagon	4	Operations	2
				Development/Back-up	2
	ANMCC	Ft Ritchie, MD	4	Operations	2
				SIOP	2
Unified	USEUCOM	Vaihingen, GE	2	Operations/SIOP	1
				Intelligence (IDHS)	1
	PACOM	Camp Smith, HI	1	SIOP	1
		Makalapa, HI	1	Operations	1
	LANTCOM	Norfolk, VA	6	Operations	4
				Intelligence (IDHS)	2
	REDCOM/ JDA	McDill AFB, FL	4	Operations (REDCOM)	2
				Operations (JDA)	2
Specified	SAC	Offutt AFB, NE	7	SIOP	2
				Force Status (on-line)	2
				Development/Back-up	2
				MAJCOM Support	1
	NORAD/ ADCOM	Colorado Springs, CO	14	Intelligence (IDHS)	2
				Space Computation Center	2
				NORAD Command Center	2
				NCS Back-up	2
				Comm System Segment	2
				Off-site Development	4
	MAC	Scott AFB, IL	7	Passenger	1
				Cargo	1
				Operations (Top Secret)	2
				Operations (Unclassified)	2
				MAJCOM/Development	1
Sub-Unified	USFK	Taegu, Korea	1	Operations	1
Service Hq	Army (AOC)	Pentagon	2	Intelligence	1
				Operations	1
	Navy (NCC)	Washington Navy Yard	4	Operations	2
				Development	1
				Back-up	1
	AF(AFDSC)	Pentagon	1	Operations	1
System Supportive	Army War College	Carlisle Bks, PA	1	Back-up Operations	1
	Air Univ	Gunter AFS, AL	2	Back-up Operations	2
Component Commands	PACFLT (PACWRAC)	Makalapa, HI	2	Operations	2
	USAREUR	Heidelberg, GE	1	Operations	1
	FORSCOM	Ft Gillem, GA	2	Operations	2
	NAVEUR	London, England	2	Operations	2
	TAC	Langley AFB, VA	3	Operations	3
	PACAF	Hickam AFB, HI	1	Operations	1
	USAFE	Ramstein AB, GE	2	Operations	1
				NATO/US Support	1
Transportation	MTMC	Falls Church, Va	3	Operations (Top Secret)	1
				Operations (Unclassified)	2
Support/ Development	CCTC (Reston)	Reston, VA	4	Development	4
	ATC	Keesler AFB, MS	1	Training	1
	Navy	Pax River, MD	1	Navy Test Bed	1
Totals	26		83	49	83

Figure 2 WWMCCS ADP Sites, CPUs, Configurations
(January 1981)

Slide 4

SOFTWARE TEST AND EVALUATION

- OFF-THE-SHELF COMMERCIAL SOFTWARE
  - ● IMBEDDED WWMCCS UNIQUES
  - ● HANG ON WWMCCS UNIQUES
- FULL RANGE OF SOFTWARE FUNCTIONALLY AND GENERALITY
- HETEROGENEOUS USER COMMUNITY-UNCONTROLLED USE
- MULTIPLE VENDORS
- GENERAL REQUIREMENTS-CONFLICTING REQUIREMENTS
- CONTINUAL CHANGE

CCTC

C400
WWMCCS ADP
TECHNICAL
SUPPORT
DIRECTORATE

TEST OBJECTIVES

- ENSURE WWMCCS SYSTEM SOFTWARE FUNCTIONS AS DOCUMENTED

- ENSURE NEW FUNCTIONS ARE ADEQUATELY DOCUMENTED

- ENSURE CAPABILITIES OF PREVIOUS SYSTEM OPERATES IN THE SAME MANNER AND ANY CHANGES IN CAPABILITIES ARE DOCUMENTED

Slide 6

Mr. Greenlee: Our next and final presentation is by our contractor support team established to carry out certain technical details in the STEP program. We have obtained the expert services of Georgia Tech and CDC out of Atlanta. Their primary emphasis will be, in addition to smoothing and shaping the overall project activities and producing the reports and other hard products, developing the data base which is really composed of two parts, the technical survey on software tools, practices and procedures, as well as the management or administrative side, which will look at the current guidelines, standards, etc. under which software is developed and tested within the Department of Defense. The principal investigator is Dr. Rich DeMillo of Georgia Tech.

DR. RICHARD DEMILLO: GEORGIA INSTITUTE OF TECHNOLOGY

Dr. DeMillo: Let me tell you what's going on with the STEP contract at Georgia Tech and Control Data because there are some ways in which you all are going to be involved, I hope you are going to be involved later on. The contract itself is funded through OSD and ONR. The prime contractor is Georgia Tech. I have a group of people working with me, mainly graduate students at Tech. We'll be dealing mainly with the state of the art in software T&E. We have a subcontract to Control Data Corporation to deal with current DoD practices. You'll hear from a couple of the Control Data people later on. In addition to the contract work that's being done, there are going to be two additional sources of input. One is this meeting. I didn't really know what to call this group, so as far as I'm concerned, you're an advisory panel. Later on, we'll have a reconvened and expanded advisory panel that I decided to call the consultants. Some of you, in fact, may end up being the consultants. I'll explain in just a moment what the point of that second advisory group is.

The goal of the project is to give Don Greenlee's office some technical information on software test and evaluation. This slide shows some things that we'll certainly want to provide. We may get to other topics as time goes on. The things that seemed most important to me were these top three items. Assessing available technology. Assessing the state-of-the-art in software testing and surveying current practices both in DoD and industry. We would like that to be as clear a picture as possible of what the collection of people both in academic and research circles and in practice think of program testing. We want to take into account changes in technology that have relatively short horizons, both hardware technology and software technology. There was talk this morning, for instance about Ada, software tools ... That's a change in software technology that will certainly impact software testing. We want to account for special DoD problems. That's a euphemism for embedded computer systems. If we're going to target anything in particular to talk about that will be right up there on the list.

Finally, we want to solicit expert positions, opinions in effect on the things that have come before. Almost certainly, there are going to be differences of opinion on the meaning of the data that we collect on the state-of-the-art and the state-of-practice. What we would like to do is to get, within our resources, a range of positions that we hope will cover the kinds of controversies that are going to come up as this step takes place, as we give input to Mr. Greenlee for what were really phases II and III of his flow diagram.

We've set up a number of tasks. We're actually in the middle of Phase I of the project. Phase I of the project is 9-11 months, depending on when you count the beginning of this contract. Phase I is a data gathering phase. The assessment of the state-of-the-art, the assessment of the state-of-the-practice, is going to be summarized in a document that we are calling an overview document. This should be a relatively complete picture of the world of software testing, at least from our point of view. We're trying to be as objective as possible about the data gathering effort. I was thinking about that this morning. I don't usually sit so quietly in meetings like this. But, I guess I'm supposed to be objective in the data gathering effort. The second bullet under Phase I, convening of the advisory panel, is really this meeting. The Control Data people who will be visiting some DoD installations in the next few months are going to need a lot of help, a lot of input, a lot of patience on your part. We had hoped that this meeting would be a ... I don't know what you would call it .. an olive branch that we're holding out to make sure that that goes smoothly. Anyway, Phase I, this lengthy phase of the project will be mostly invisible to you. You will see the output of it at the end. That will feed into Phase II, the analysis phase. This is the non-objective part of the project. We'll select, with your help and input, a panel of ... I don't really know how many ... we had suggested 20 at one time ... we're open to suggestions on that ..., a panel of consultants or advisors whose job it will be to develop position papers based on the overview document. The overview document will be distributed to them. The position papers should hit the points that Don Greenlee mentioned this morning and some of you have talked about since then. We would like postion papers on the notion of risk assessment with regard to program testing, status and limitations current technology. Presumably academic people will be involved developing those position papers, suggestions for future research and development. I think what we will find is that there will be lots of suggestions for future research and development in order to get the kind of T&E technology that we would all like to see. The economics of program testing. And input on feasible policy formulation. Those position papers will be developed over a 3-4 month period. I'm not particularly concerned that we get contradictory positions, in fact, I hope that we will get some contradictory positions so we can lay out a range of opinion on the matter. We will then hold a workshop at the end of Phase II at which the position papers, and we hope, extensive discussion will be presented. This entire Phase II project should be completed, once it begins, in 3-4 months.

The overview document that we are now in the midst of preparing, as I said, has 2 parts to it. One is an assessment of the state of the art, the other is an assessment of current practices both within DoD and industry. The topics in the state of the art survey, I just picked a few from the outline, and I'll talk through some of these in a little more detail in just a moment. The state of theoretical knowledge, the role of software metrics, what methodologies are available for software test and evaluation, what methodologies are available for test data generation, what tools, automated tools, are available, what technological advances will mean to T&E ... someone suggested this morning that being able to migrate some software tasks into hardware would have an impact on it ... Finally, the relationship of applications to software testing. The Control Data portion, the current practices survey is data gathering in the true sense of the word, I think. Ronnie Martin and Don Miller have been gathering material on existing regulations for T&E, project experience from various military organizations, effective procedures or ineffective procedures, as the case may be, and finally, data on such things as the kinds of errors that are reported in existing systems and additional statistical data.

I'd like to just show you the kinds of things that we are going to talk about in the state-of-the-art portion of the overview and then, when I'm done, Ronnie and Don will tell us about what you can expect from the current practices portion of the overview document.

The state-of-the-art survey is going to cover three aspects of software testing. On the next slide, I'm going to limit the world of software testing for you, so we shouldn't have any definitional problem. We'll cover the current state of theoretical knowledge, the current status of tools that we know about for test and evaluation, and these will be both commercial products and things that are in the public domain, and finally, take care of that issue of technological forecasting, as it effects T&E.

Unlike a couple of the previous speakers, I'm not talking about operational testing. Right now, I'm talking about this kind of testing which is really the only thing that is dealt with very well in the technical literature. It is really development testing. The picture you should have, I think one that you all are familiar with, is one in which the proposed methodology will somehow take test data, to be generated, and a specification document and compare the results of executing the program on that test data with the expected results from the specification document for two purposes, one to discover errors ... Vance, I think this morning you had in the Ada kernal diagram, debuggers sitting at the MAPSE level, I guess I'd move those into the program testing realm insofar as debugging helps you discover errors ... for error discovery and for confidence building. If you don't find errors what does that tell you about the reliability of the program? So, the state-of-the-art survey is going to be concerned with the techniques that are available for these two aspects of program testing and finally, the sort of surrounding infrastructure, the management of that task.

Dr. Fischer: Is this going to be limited just to software testing or also to integration testing where we put software on a chip and test that?

Dr. DeMillo: We'll say some things about that only to say that we don't know much about it from a research point of view. One of the things that we will do is identify research directions. I think that is one that has to be put on that side of the board.

I'm going to show you some topics so you can see the things that we will be dealing with. When I talk about testing techniques, here is a list of techniques that is not inclusive but it is the kinds of things that we will be looking at. The applications areas .. I don't think anyone here feels left out by that. Tools ... and really the hardest section for us so far is the assessment of new technology.

When Don Greenlee saw the outline for this section of the overview document, he thought it would be a good textbook, and I hope that, at least, if not a textbook, at least a handbook that is encyclopedic of what is known today about software testing. I don't know what we want to do about questions, I'm willing to take questions now and then turn over things to Control Data.

VIEWGRAPHS

USED BY

DR. DEMILLO

FOR THE

SOFTWARE T&E PROJECT STATUS & PLANS

PRESENTATION

THE GEORGIA TECH

SOFTWARE TEST AND EVALUATION PROJECT

(STEP)

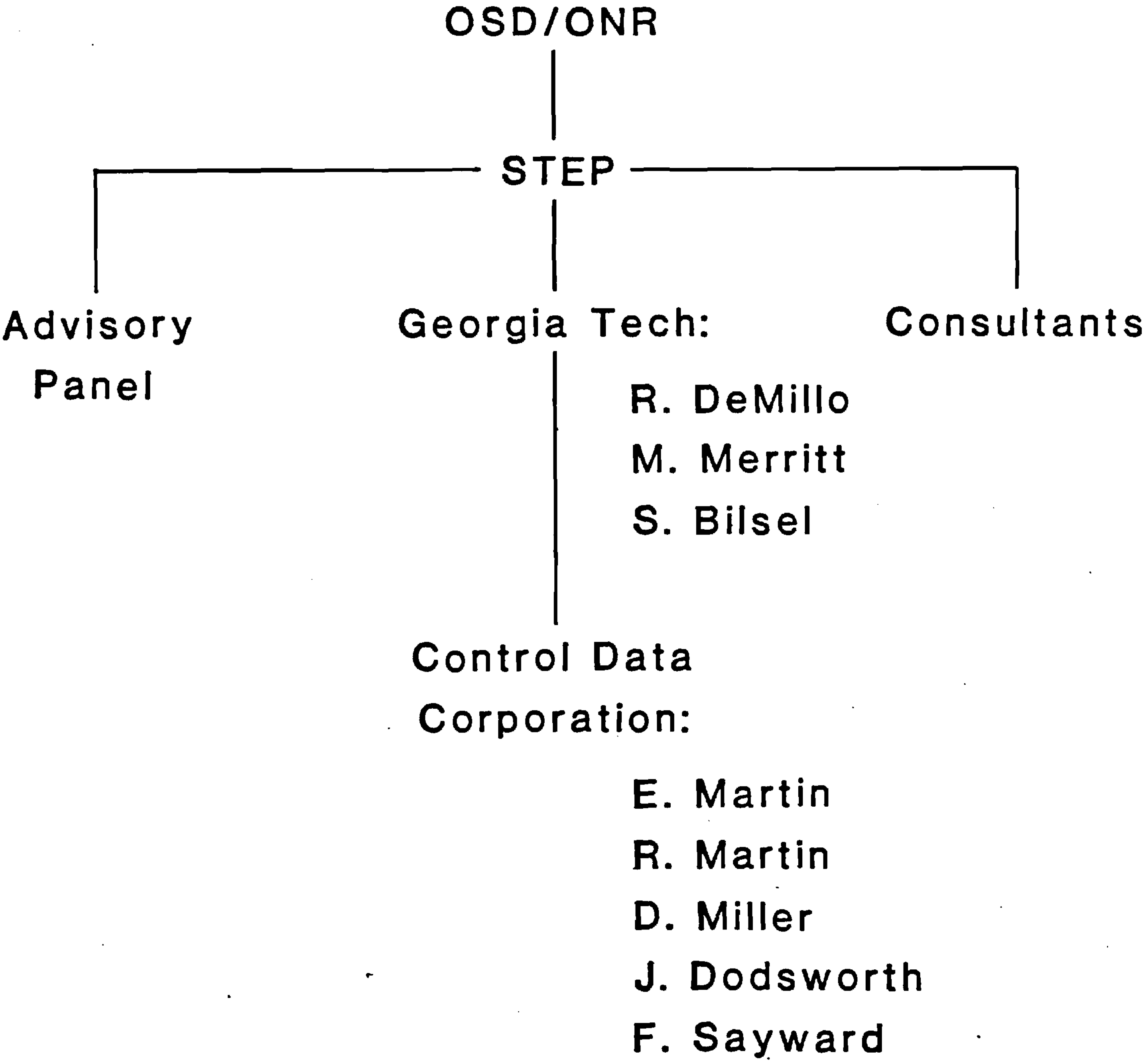
Richard A. DeMillo

School of Information and Computer Science

Georgia Institute of Technology

Atlanta, GA

STEP CONTRACT:



OSD/ONR

STEP

Advisory Panel

Georgia Tech:

R. DeMillo
M. Merritt
S. Bilsel

Consultants

Control Data Corporation:

E. Martin
R. Martin
D. Miller
J. Dodsworth
F. Sayward

PROJECT GOAL:

Provide Technical Information on Software Test and Evaluation

- Assess Available Technology
- Survey Current Practice
- Assess State-of-the-Art
- Forecast Future Technology
- Account for Special DoD Problems
- Solicit Expert Positions

- Input for Policy-Level Decision-Making

PROJECT TASKS:

Phase I: Data/Information Gathering

- Overview Document Preparation
- Convening Advisory Panel
- 9-11 Months

Phase II: Analysis

- Distribution of Overview Document
- Selection of Expert Panel
- Development of Position Papers
    - Risk Assessment/Management
    - Status and Limitations
    - R&D Needed
    - Economics
    - Feasible Policy Formation
- Workshop
- 3-4 Months

OVERVIEW OF SOFTWARE T&E:

1. State-of-the-Art (Georgia Tech)

   - Theoretical Knowledge
   - Metrics
   - Methodologies
   - Test Data Generation
   - Tools
   - Technological Advances
   - Application Areas

2. Current Practices (Control Data)

   - Regulations and Standards
   - Project Experience
   - Effective Procedures
   - Data

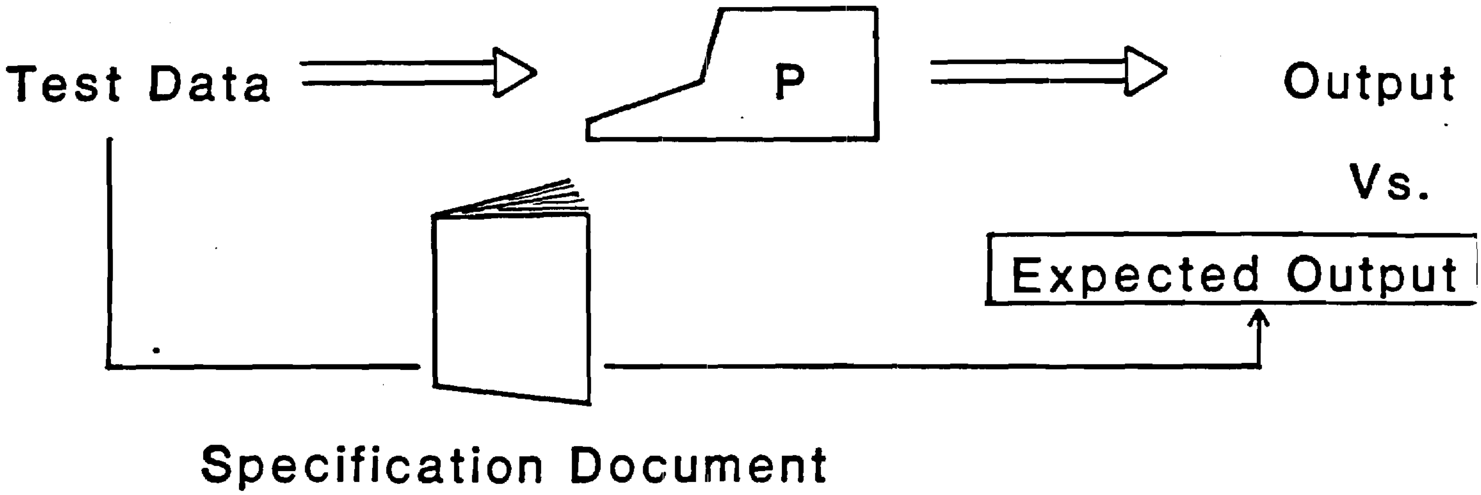
STATE-OF-THE-ART:

I. Theory of Software T&E

II. Software T&E Tools

III. New Technology

SOFTWARE TEST AND EVALUATION:



- Error Discovery

- Confidence-Building

- Management of the Task

APPROACHES TO SOFTWARE T&E:

1. Concepts
   - Black Box/White Box
   - Structured Methodologies

2. Test Case Design and Generation
   - Black Box Methodologies
   - White Box Methodologies

TECHNIQUES:

1. Static Analysis
2. Symbolic Evaluation
3. Instrumentation
4. Compiler-Based Techniques
5. Mutation
6. Domain Testing
7. Functional Techniques
8. Path Analysis
9. Algebraic Techniques

APPLICATION AREAS:

1. Embedded Systems

2. Communications

3. Real-Time Systems

TOOLS:

1. I/O Behavior
   - Test File Generators
   - Test Data Generators
   - Output Comparators
   - Mutation-Based Systems
2. Un-Augmented Tools
   - Code Auditors
   - Static Analyzers
3. Augmented Tools
   - Dynamic Assertion Processors
   - Dynamic Execution Verifiers
   - Self-Metric Instrumentation
   - Symbolic Evaluators
4. Productivity Considerations
5. Experimental Systems

NEW TECHNOLOGY:

1. Software Technology
   - Operating Systems
   - Compilers
   - Data Bases
   - Languages
2. Hardware Technology
   - VLSI
   - Memory
   - Graphics
   - Architecture
3. Communications Technology
4. Applications Technology

MR. DON MILLER: CONTROL DATA CORPORATION

Mr. Miller: I am Don Miller. Ronnie Martin and I will be doing this task of surveying DoD and its components, gathering data. We work for, as you know, Control Data Corporation. I always like to start out with wiring diagrams. This will give you a fast idea of the approach we're going to use in gathering this data. The command and control lines are the solid lines. The lines that I'm interested in are the dotted lines which are the coordination lines for the lines from which we will be gathering data. We'll probably follow the chain of command. We'll start at the top level and follow the lines down to the program managers. We are interested in the Headquarters levels of the Services, the Materiel Commands, the Operational Test and Evaluation Agencies, and the Program Managers, etc. This is basically what we are going to be doing. We are going to be looking at the regulations and policies as far as software test and evaluation. Then, we're going to survey the military installations on how the regulations, procedures, etc. are implemented.

Mr. Devlin: How are you going to determine the implementation, by examination as a question or are you going to examine the contracts...?

Mr. Miller: My partner is going to get into that in just a few minutes, into the details of the methodology that we are going to use. Any questions?

Dr. Leathrum: It seems there might be a little concern with finding ways where the regulations were ignored or not ignored.

Mr. Miller: Well, hopefully, we'll find that out. We will definitely be reporting our findings, and we'll be seeing you all again, real soon.

VIEWGRAPHS

USED BY

MR. MILLER

FOR THE

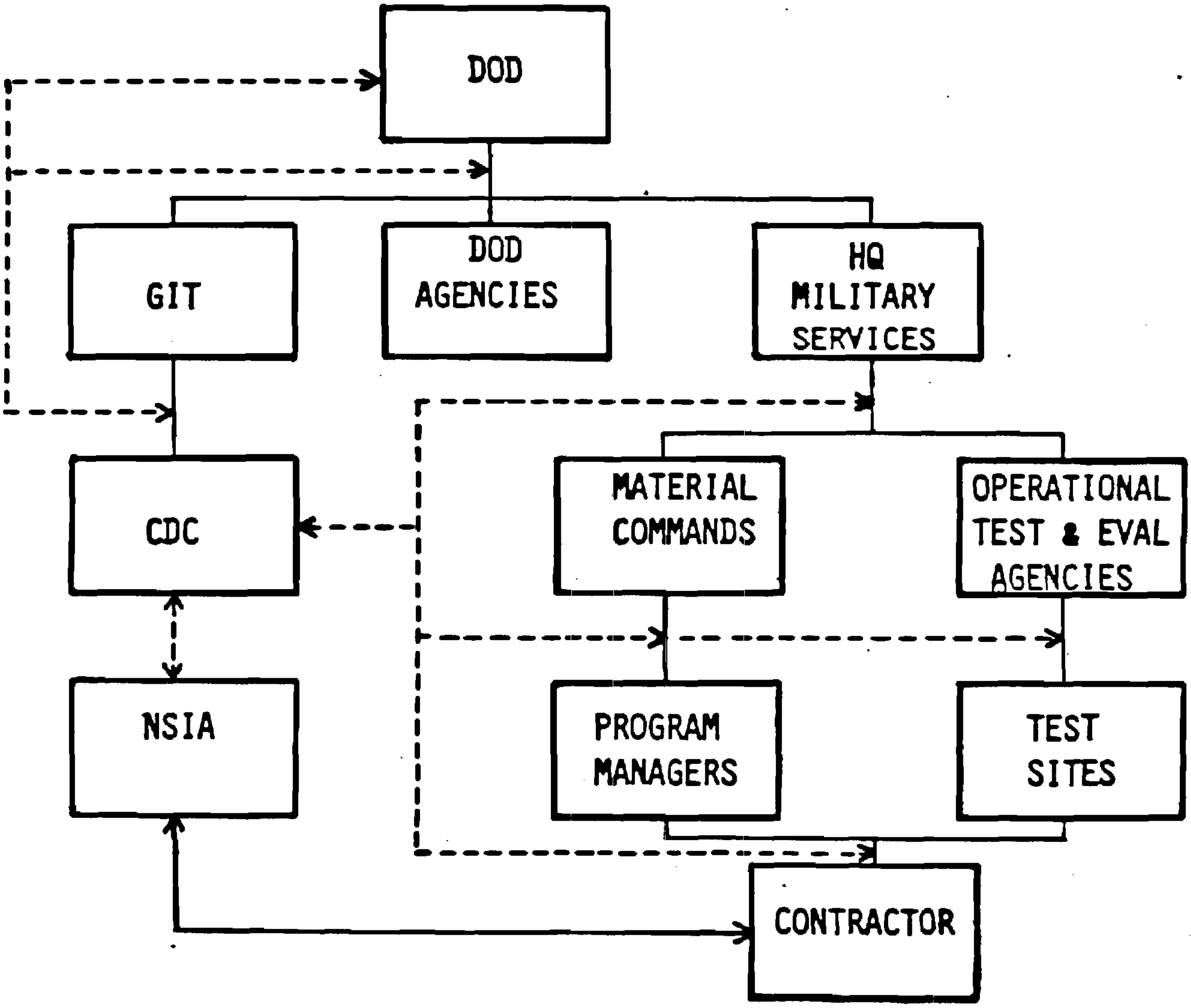
SOFTWARE T&E PROJECT STATUS & PLANS (CONT.)

PRESENTATION

DEPARTMENT OF DEFENSE

SOFTWARE TEST & EVALUATION ACTIVITY

COMMAND, CONTROL & COORDINATION


DOD

GIT

DOD AGENCIES

HQ MILITARY SERVICES

CDC

MATERIAL COMMANDS

OPERATIONAL TEST & EVAL AGENCIES

NSIA

PROGRAM MANAGERS

TEST SITES

CONTRACTOR

DOD PRACTICE:

What Regulations, Etc. Exist?

DoDD 5000.3

MIL-STD-1679

. . .

How Are the Regulations, Etc. Applied?

	Army	Navy	Air Forc
HQ	?	?	?
Development Cmds (& Contractors)	?	?	?
OT&E	?	?	?
IV&V	?	?	?

MS. RONNIE MARTIN: CONTROL DATA CORPORATION

Ms. Martin: As it was mentioned before, my name is Ronnie Martin. I work for Control Data Corporation.

The basic methodology we're going to use in trying to gather our data on the current practices in DoD and industry as far as software test and evaluation is ... Don and I have put together a whole group of Data Gathering Guides which we will use to interview people when we go on our visits to various military installations. The guides are specifically tailored according to whoever it is we're talking to. We have one for the Headquarters and Materiel Commands, we have a separate one for the Program Managers, and the Government Programming Shops, Industry Programming Shops, OT&E Agencies, and IV&V Organizations. The most detailed one of those guides is the one for the Program Managers because that's where we want to get the detailed information according to what application the people are working on, how they test the software for that application, how they evaluate the software, what do they do as far as developmental testing, operational testing, and so on.

These are some of the areas that we'll be looking into. The items that I gave you are overviews of the Data Gathering Guides. A lot of the sections look the same under each of the different guides, but there are some slight differences. For instance, if we talk to an IV&V organization, we're not going to ask them everything in the world about operational testing. It wouldn't be appropriate there. When we talk to the operational testers, we're not going to ask them all of the detailed questions about developmental testing. We will ask them to what extent they participate during the development stage but we won't ask them detailed questions on it.

Mr. Devlin: Are you going to ask about a specific program, say any given program, X program? Are you going to start out with the program sponsor, say at the Headquarters level and then go the the PM and then go the OT&E and then go to the IV&V on one program?

Ms. Martin: We're going to hit more than one program, but we want to get as much information about the programs that we choose to visit as possible. Initially, what we want to do is talk to the Headquarters people. The type of information we want there is basically, what regulations and standards exist. We want guidance from those people as to what programs are at a good stage for us to visit. If something is way back in concept, they're not going to have a lot of details on how it's going to be tested. They'll have some ideas, but we wouldn't be able to get nearly the amount of information there as we would at a project that's in the process of being tested or one that was just deployed 6 months ago. This is the type of information we want at the Headquarters level. And, similar information from the Materiel Commands.

The next level down from that would be the Program Managers. The Program Managers we will visit will be the ones for the specific programs we want to look at. If he can't give us all of the information we would like in the detail we would like, then, hopefully with the help of NSIA, we'll go to the contractor who actually did the programming. We'll go and talk to the operational testers who are involved in it. If there is an IV&V organization, a contractor involved there, we'll talk to them. We'll try to get a total picture of all of the test and evaluation that is done on that specific program. We're hoping to hit multiple programs, I don't know exactly how many ...

Mr. Devlin: With what success rate? Are you going to look at the people who survive? The people who didn't survive? ... That's what I was wondering, the validity of the case ... you can pick up any contract and see almost any regulation that there is invoked.

Ms. Martin: The important question here is, we want to know what regulations exist and we want to know how they are implemented. It's not a witch hunt, we're not trying to point fingers at anybody. We just need as much information as we can get so we can give a realistic picture of what the current practices are.

These are the various areas that we want to look into. Background Information. Basically, the type of thing we want to know there is what programming language is being used, what type of application this is. Is it radar ... is it C^3I ... what is this? What regulations and standards exist, the people that work on this project, what they have to follow? Industry testing standards vs. DoD practices. What is required for industry to follow? Do you have any methodologies or any special techniques within your corporation that go above and beyond what's required by the government? What exactly are the standards in industry vs. those in the government? Pre-testing activities are things like design reviews, code reviews, walkthroughs, inspections, requirements analysis, various areas like that. Developmental Testing and Evaluation, people have talked about a lot today. Integration Testing, once you've tested the various modules to make sure they're OK, how do you test that they work with each other? How do you test that they work with the hardware? We want to look at Independent Verification and Validation, Operational Testing. Acceptance Testing, exactly what do you consider to be your acceptance testing, is it part of the developmental testing, do you have a separate group that does that, exactly how that is handled. How do you document the tests that you do? How do you maintain that media so that you can retest things later? Another part of each of these sections where appropriate is regression testing. Given a specific change to a program, how do you decide how much testing you have to do. What are your procedures in those areas? What quality assurance programs do you have? What kind of risk assessment do you do for software to decide how much testing should be done given how critical this piece of software is? Finally, we want as much information as possible from people as to what they see the new technology trends to be and if you're doing anything to prepare for those new technologies, if there are any engineering studies or anything like that as far as how we're going to be ready to test according to these new technologies.

This is a list of some of the types of supporting documentation that we may be asking for. Organization charts, that gives us an idea of how everybody fits together in the big picture. We need Operations and Functions Manuals so that we're sure we don't miss somebody that's important, that could give us some good information, and to know whose responsibility is it to do this, whose responsibility is it to do that. What regulations and supplements you have, since as you go down the chain in the military, people supplement and supplement, and there goes a pamphlet ... we need to know what's out there. We need to know what the guy down at the bottom has to follow. What Standard Operating Procedures do you have? Descriptions of the applications that we're looking at. Not anything real, real detailed, because I don't have an engineering degree to understand that, and that's not what I'm interested in. I just want to know basically what applications are involved in this program, so that if we have some application specific techniques, we can label them as such. We would like to see some test plans and procedures because a lot of the questions we will be asking relate to how you came up with those. What method did you use for your test case design? Finally, if it's possible, we would like to see some error data. By that I mean, what kinds of errors have been found once the system has been deployed or in the testing what kinds of errors have been found. What are your ideas as far as what kind of testing might have found that?

Another thing that we'll be asking all the way through is when we're talking about specific techniques, we'll ask you what do you like about this technique and what do you dislike. We believe that for any technique to be really useful, the people that have to use it have to like it. So, we'll be asking those types of questions. This is everybody's opportunity to get their input in as far as what they think would be good, what's worked for them, what hasn't work.

Dr. Fischer: Are you aware of the Data and Analysis Center for Software that RADC has? They have file drawers full of error data.

Ms. Martin: That might be a good source of information, but the error data all by itself won't be worth much if we don't know how they tested in the first place. So, we need it all as a package to be able to figure out what it's worth.

Finally, this is an outline of what our part of the overview document that Rich was talking about will look like. It has a background section. We'll explain in there again how we went about our data gathering. The important part is the results which we will organize according to the state-of-the-art format so that if you read the front part of the overview where it's state-of-the-art, if you read about specific techniques you can turn to a similar section for Current Practices and see what is really being done in those areas. That will include a listing or some kind of a write-up about current standards and procedures. Then, the practice of testing and evaluation rather than the theory of it, this will be how it's really done. What philosophies were used, test case design, testing techniques, evaluation techniques. What tools are available for people. That will be an important part of the questions we'll ask. What tools do you have available? What tools do you use? Do you have any available that you don't use, and if so, why not? This is to try to find out what problems there are with tools, what the needs are.

We'll have a section on applications. Again, we want to know if any techniques are application-specific and we'll be asking questions about that. We'll also have a section on new technologies. That will depend upon what information we can get.

We're completely dependent upon all of you people for all of the help that you can give us. The more information you give us, the better job we can do with this whole thing, and the more likely it is that it will succeed. If you don't give us any information, we've got nothing.

Finally, we'll have a summary at the very end of it. This will be it. The State-of-the-Art part, the current practices part, that'll be the overview document for the panel of experts to come in and look at and decide what's good, what isn't, and where we should go from here. As Rich said, we're going to be as objective as possible throughout this. We're just gathering data so that we can report it, and the experts can decide what to do from there.

VIEWGRAPHS

USED BY

Ms. MARTIN

FOR THE

SOFTWARE T&E PROJECT STATUS & PLANS (CONT.)

PRESENTATION

DATA GATHERING GUIDES:

I. HQ & Materiel Cmd Visits

II. Program Manager Visits

III. Government Programming Shop Visits

IV. Industry Programming Shop Visits

V. OT&E Agency Visits

VI. IV&V Organization Visits

AREAS OF INTEREST:

Background Information

Regulations and Standards, Etc.

Industry Testing Standards Vs. DoD Practices

Pre-Testing Activities

Developmental Testing and Evaluation

Integration Testing

Independent Verification and Validation

Operational Testing and Evaluation

Procurement-Acceptance Testing

Test Documentation Procedures

Quality Assurance Program

Risk Assessment

New Technology Trends

SOFTWARE TEST AND EVALUATION
- STATE-OF-PRACTICE DOD:

Background and Overview

Data Gathering Methodology

Results Organized According to State-of-the-Art Format

A. Current Standards and Guidelines, Procedures, Etc.

B. Practice of Testing and Evaluation

1. Testing Philosophies Used
2 Test Case Design and Generation Techniques Used
3. Testing Techniques Used
4. Evaluation Techniques Used

C. Testing and Evaluation Tools

D. Applications

E. New Technologies

Conclusions - Summary

SUPPORTING DOCUMENTATION:

- Organization Charts
- Operations and Functions Manuals
- Regulations and Supplements, Etc.
- Standard Operating Procedures
- Program Descriptions
- Test Plans and Procedures
- Error Data

Mr. McOmber: I don't have any questions, but I have a couple of observations. You may or may not be aware of it, but DoD is in the process of conducting a survey exactly like this and gathering a lot of the same information regarding software problems. A year or so ago within the Navy - in fact Bill Smith, who is here today - conducted a survey asking for a lot of the same information. Admiral Williams, the Chief of Navy Material, commissioned our office to conduct a survey of the entire Material Command, including all project managers and Navy development activities, collecting data which will be a superset of your data. Number one, it seems like the world is being surveyed to death. Number two, we decided to go in-house with our survey, because two years ago, when the project was being planned, we had a contractor already signed up, and our contract control people turned it down because the contractor was also involved in the development of Navy software, and they thought that this was inappropriate. That's my comment. I'm not being negative about the whole thing, just a couple of observations.

Mr. Greenlee: Well, I understand and appreciate your comments. It is our firm commitment not to replow any new ground. What you have heard described is the from-scratch data gathering effort, but obviously, we are committed to getting the most mileage out of everything that has already been compiled. Nothing would please us more than if this were simply a capstone effort to pull together existing data. We are not going to go out and ask a single question more than we have to or take up anybody's time where their thoughts have already been recorded elsewhere. We are aware of many of these survey efforts, and I'm sure we will turn up more as we go along. Our objective is not to compile raw data. It is to reach conclusions. We will welcome surveys and inventories and existing and ongoing and planned efforts of this type. We know about a lot of efforts that are going on within and between Services, even NATO, for example. The CDC objective, although they have described it as a from-scratch effort, is really to pull together existing information to the extent possible. Caral?

Mrs. Giammo: When I was thinking about what I was going to say here, I tried to put together a chart that I hope you people can put together, because I don't know how. I tried to develop a pictorial representation of kinds of software, the states of the software development process, the controls over the development process, including requirements, and tools that are available for software testing. There would be guidelines or something that says if you are up here, here is a test tool that is very good, and if you are down in this part, here are other things. I find an awful lot of what I do is trying to explain why some technique that is really super good doesn't at all apply to the work I'm doing. It doesn't technically make any sense. I don't know how to do it.

Mr. Greenlee: You may recall from the first flow diagram this morning, that one of the tasks we have set for ourselves is to attempt to not only survey and draw together test techniques and tools and practice but to make some kind of a knowing assessment, an evaluation of these and try to determine in what areas certain ones are good, what ones are not good, etc. We do intend to attempt to assess as well as simply gather raw data on the value of software tools.

Dr. Leathrum: A number of speakers said this morning that what we do depends on the capabilities of the people we train. It makes me wonder if CDC's survey shouldn't include educational institutions as far as what techniques are being taught ...

Mr. Greenlee: I think that is a good point, and we are committed as you have seen from our displays to bring in the ideas from the university community as well. We have a suspicion that what government asks for, industry provides, and colleges teach are not all at the same level.

Unidentified person: Don, you mentioned that one of the hopeful outcomes of this effort is to write a guideline for the test and evaluation of software and that will become part of DoDD 5000.3 and this effort will take about a year. What is the current status with revising 5000.3?

Mr. Greenlee: To answer your first question, I don't think a decision has been made on the medium by which any guidance which is developed will be promulgated. 5000.3 is the likely choice of vehicle because it is the T&E document that we live by, and it does call out software testing as a special section. There are other alternatives. We will certainly not make the next revision of 5000.3 which I think will occur rather shortly. 5000.1 and .2 are very close to being issued now, and we expect to provide whatever additional change or extra material is needed by 5000.3 very shortly, but certainly we will not be in a position to infuse any of this effort into that.

If there are no further questions or comments, I would like to bring the workshop to a close by thanking all of you who attended. Your participation made this meeting a very interesting and useful forum for the exchange of information and ideas on software testing. We owe special appreciation to the DSMC for serving as our institutional host on this occasion.

The material presented here today and the ensuing discussions were far too broad and diverse to summarize briefly, so I won't attempt to. However, I believe we have reaffirmed the importance of effective software T&E. Furthermore, it appears that there is room for improvement in the technical state of the art in software T&E, i.e., tools and techniques, or at least in the dissemination and application of existing methods. Finally, it seems clear that the management side of software T&E, i.e., the policies, practices and guidelines under which software is tested, is also a candidate for enchancement. These are exactly the subjects of subsequent STEP efforts.

You have heard the planned STEP methodology described. We are proceeding to gather information from government, industry and the academic world. A second workshop with wider representation will be held at a later date to promulgate and discuss findings to date. An expert panel will develop papers on specific topics. It is hoped that this activity will ultimately lead to conclusions which will be specific enough to serve as useful guidance in software T&E yet general enough to encompass all or most DoD embedded computer resources. Your continued interest and involvement are welcomed and appreciated.

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

VOLUME 5
REPORT OF EXPERT PANEL ON SOFTWARE TEST AND EVALUATION

SUBMITTED TO
THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION
AND
THE OFFICE OF NAVAL RESEARCH
ONR CONTRACT NUMBER N00014-79-C-0231

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

FOREWORD

This volume is one of a set of reports on Software Test and Evaluation prepared by the Georgia Institute of Technology for The Office of the Secretary of Defense/Director Defense Test and Evaluation under Office of Naval Research Contract N00014-79-C-0231.

Comments should be directed to: Director Defense Test and Evaluation (Strategic, Naval, and C^3I Systems), OSD/OUSDRE, The Pentagon, Washington, D.C. 20301.

Volumes in this set include:

Volume 1:	Final Report and Recommendations
Volume 2:	Software Test and Evaluation: State-of-the-Art Overview
Volume 3:	Software Test and Evaluation: Current Defense Practices Overview
Volume 4:	Transcript of STEP Workshop, March 1982
Volume 5:	Report of Expert Panel on Software Test and Evaluation
Volume 6:	Tactical Computer System Applicability Study

Volume 5

Report of Expert Panel on Software Test and Evaluation

TABLE OF CONTENTS

PAGE

An Integrated Software Test Technology Research Plan 1
by Edward F. Miller, Jr.

Rapid Prototyping and Refinement: An Approach to Software . . 8
Testing by Richard J. Lipton

Integration Testing of Software by James F. Leathrum 10

Status and Directions for Software Testing and Evaluation . . 14
Tools by Leon Osterweil

Assessing the Software Product Qualities of Correctness . . . 25
and Reliability by Victor R. Basili

Impact of Software Testing Issues on Future Software 33
Engineering Environment by Leon G. Stucki

Application of Software Metrics During Early Program 34
Phases by Ralph C. San Antonio, Jr.

Managing Software Testing Using Reliability Estimates 42
by Martin L. Shooman

Software Error Studies by Carolyn Gannon 50

Experience in Testing Large Embedded Software Systems 57
by John B. Bowen and Marion F. Moon

The Economics of Software Testing - An Introduction 65
by Raymond J. Rubey

Programming Languages, Testing and Reusability 69
by Peter Wegner

Impact of New Hardware Technology by D. A. Giese 80

Software Testing Standards: Policy and Applications 99
by Marilyn J. Stewart

Software Quality Assurance and Acquisition Policy 105
by Capt. William P. Nelson

Air Force Testing/Acquisition Policy 112
by Major Arthur E. Stevens

Acquisition Problems Influencing Software Development 116
and Operational Testing by Lt. Col. Michael A. Blackledge

Perspectives: Software Test and Evaluation Project - 123
A Progress Report by Richard A. DeMillo

Panel Discussions 133

An Integrated Software Test Technology Research Plan

Edward F. Miller, Jr.
Software Research Associates
580 Market Street
San Francisco, CA 94104 USA

Phone: (415) 957-1441 -- Telex: 340-235

ABSTRACT

It appears clear that the essential issue for the computing community in the 1980's is to develop the technology to assure successful creation of complex software systems at the necessary high level of Quality but with acceptably low Cost. Already, some overseas competition has begun to offer software with "extended warranties," something that domestic manufacturers of software have not yet attempted.

Research and development for the past decade has made substantial strides toward the dual goals of Quality and productivity; yet much work remains to be done. Many of the technical approaches that have been identified in the past are, apparently, effective only on relatively small projects. Methods have to be devised to extend "laboratory scale" approaches to "commercial scale" problems.

Among major software engineering problem areas, software testing and analysis has become a major bottleneck, due to a lack of understand of the technology, a minimal understanding of the underlying theory, and inadequate tools base. This paper describes a systematic program of advancement in theory, technology, and necessary support tool functions designed to provide for software Quality needs for the remainder of the century.

INTRODUCTION AND BACKGROUND

It is becoming clear that the main issue in software engineering in the 1980's will be "quality" in one form or another. Quality in software must be delivered with acceptably low cost, particularly in view of the growing international competition for high-quality high-capability software.

Significant strides have been made in the 1970's toward achieving reliable software. These advances have spanned a range of technical areas, and the software test and evaluation area has experienced great growth and development. However, to meet the challenges of the remainder of the decade new approaches must be made to achieving better quality software at higher-than-ever production rates.

The Software Test and Evaluation Project (STEP), (Reference 1), is an example of the kind of program that can have a significant benefit if carried through properly. An earlier workshop, held as part of that Project (Reference 2), provided some initial thinking on how best to attack the software test and evaluation problem.

This paper describes how, in rather general terms, software quality is actually achieved, and discusses how quality management is made to fit into the software life cycle. Based on this perspective, we then define a series of research needs in broad technical areas. A goal of identifying these needs is to focus attention on those few important technical areas which could benefit from accelerated development.

The research needs identified form an integrated package of technical requirements that could for the basis for a long-range technology development plan.

HOW IS QUALITY ACHIEVED

Quality is "installed" at varying cost throughout the life cycle of a software system by means of a number of technical devices, not all of which are completely explicit. Effectively, testing in one form or another accounts for the majority of error discoveries. Gross error rates range from 0.1% through 3.0% for unaudited code; it is unknown how many other errors exist that may cause problems. Some of these errors are found on purpose, and some are not. In fact, some are found only by field tests made by the final users!

The feedback process involved in perfecting a software system often involves developers, quality assurance experts, beta-site evaluators, and the first group of actual customers. While this technique of distributing costs may efficient in its way (the least number of people spend the least possible money producing the minimally acceptable product), the old formulas for software production do not apply in an era of software guarantees and, possibly life-critical systems fully dependent on software.

A range of techniques currently exists to analyze computer software. This range includes the following, where the Quality Management (QM) Level indicates groupings within the underlying quality assurance technologies:

QM 0.1: Design Walkthroughs and Guidelines: Procedures and checklists designed to assist in analysis of software design statements.

QM 0.2: Code Walkthroughs and Guidelines: Procedures and checklists designed to assist in analysis of source level software listings.

QM 1: Static Analysis: Automated tools that try "perfect induction" methods in an attempt to monitor source level programs for defects.

QM 2.1: Dynamic C1 Analysis: Systematic testing aimed at achieving a high level of segment coverage.

QM 2.2: Dynamic Ct Analysis: Systematic testing aimed at achieving tests for each equivalence class of program flow (this is related to domain testing).

QM 2.3: Dynamic Mutation Analysis: Systematic error seeding and testset analysis.

QM 3: Symbolic Evaluation: Treatment of the source level of a system in detail at the symbolic level (related to Proof of Correctness).

QM 4: Proof of Correctness: Mathematical verification of software properties using mechanical proof methods. Often, this is done by reference to data type abstractions of various kinds.

Level 0 Quality Management methods are largely manual approaches, although there is some augmentation with automated tools. The level 1 method relies on the use of specially built static analyzers, which are normally both language and system dependent. Level 2 techniques involve the use of dynamic analysis methods of varying kinds. The C1 level, which is the most basic, is becoming a standard (see below). Ct, requiring source level analysis of the control structure to recover the software system structure in order to identify the equivalence classes, is more thorough, and emulates proof methods.

QUALITY MANAGEMENT IN THE LIFE CYCLE

Delivery of software quality control and management technologies must be phased carefully with the software life cycle. This can be done by phasing the technique with the corresponding phase of the life cycle.

For example, Level 0 techniques relate (obviously) with the design and coding phases of the life cycle. The Level 1 method must relate directly with the post-coding stage, since the source code must exist and be processable by the static analyzer.

Similarly, Level 2 methods can only be applied at the point where code and test data exist. Note that the mutation analysis phase requires that some test data exist, or else there is now way to "retire" mutants.

Level 3 methods apply, if appropriate in cost and effort levels, only to working programs. Level 4, involving potentially the highest costs for the smallest returns, must be used only sparingly. Notably, techniques based in Level 4's technology, primarily involving the application of abstract data types, is finding application in the requirements analysis phase.

What is conspicuous by its absence is the fact that few of the current software quality management methods have direct application during the early-on phase of requirements/specification. This is a lack that must be remedied in the future.

RESEARCH NEEDS

All of the foregoing requirements suggest a range of "technology needs" that, if achieved, would serve to augment the capability to systematically test and evaluation computer software. In this section we have identified some 30 different technical needs in a variety of areas. In part, this identification is based on an earlier paper (Reference 3), except that the range of needs considered is expanded and the context is updated.

The technology needs fall into a number of general areas, roughly outlined as follows:

Theoretical Foundations: The technical developments that support all of the other areas, based on mathematical understanding of the underlying technical models that describe computer software. The theory must preceed development of applicable methodologies.

The state of the art in testing theory is good, but not complete. Tests can be shown reliable only if the error class is effectively restricted (according to Howden, to action errors only). No basis exists (yet) for mutation analysis, even though the experimental results for it are attractively good.

Methodology: These are the organized procedures that bind theoretically sound methods and techniques into a workable strategy for accomplishing some useful work. The methodologies actually used must be based on successful empirical evidence of success.

Most organizations now have some Quality Management methodology in place, but these cannot be considered "strong" systems in all cases.

Empirical Experience: This area deals with past experience, with experiments that can yield new information about particularly interesting quantities, and with applying available data to problems at hand. Only with a knowledge of theoretical foundations can good experiments be designed.

The data exists, but it has only be analyzed in very limited form. And, much more data needs to be examined.

Automated Software Support Tools: Automated tools provide a means of carrying out the evaluations that are prescribed by the methodologies developed earlier. Good tools can contribute significantly to the overall quality of a software system by lowering the cost while maintaining a given quality level and/or increasing the capability of the system at the same cost.

Automated tools continue to grow in number, but not necessarily in effectiveness (Reference 6). Some systems (notably the Unix (tm) environment) seem to provide a wide range of capabilities at relatively low cost. Current programming environment research work has yet to produce usable products.

New Technical Approaches: Among the range of current ideas, a few appear to deserve significant new treatment because of the possible high payoff they promise.

The subsections below will investigate each of these areas and propose some new techniques as well as comment on old techniques.

Theoretical Foundations

While theoretical work was advancing rapidly in the early 1970's, progress appeared to slow down in the latter 1970's. Quite probably this was due to the fact that the "easy" results had been obtained and published, and what remained were difficult and messy investigations.

The relationship between proof of correctness and testing is, at last, reasonably well understood (Reference 4). It is known that software testing cannot show the absence of software defects, and that finite sets of tests can never replace the use of full-blown proof methods.

The strong reliance on software structure, both in proofs and in testing-level analysis, leads to the first Need:

Need 1: Devise a general relationship between a programming structure and the content, effect, nature, and/or behavior of the corresponding program.

Such a structure would permit a uniform treatment of program structures independent of language and execution environment. Once this general structure is established it should be possible to develop general rules for identification of software faults.

Need 2: There needs to be a general theory of formal testing that relates the extent to which a test along a particular path can, or cannot, discover particular kinds of defects.

Once it exists, this theory would permit predicting whether or not software contains certain kinds of defects, provided only that the guidelines and limits of the theory had been met. The problem, however, is how to do this with a few tests, rather than a large number (when it is likelier that the defects would be found).

Need 3: There needs to be a better relationship between the kinds of tests that are run and the kinds of defects that are discovered, so that it can be shown in advance that certain tests are sure to prevent certain kinds of defects.

This may not be possible in every case, since the execution of computer programs is extremely complex when viewed from the level at which software defects must be sensed and reversed.

A common means to handle a problem is to divide it into pieces "small" enough to be useful. At present, a "test" has to be viewed as a single monolithic element to be analyzed. This can be particularly troublesome at the system test level.

> Need 4: The relationship between a test of a system and the tests of a subset of the system is not understood. There needs to be a general investigation into the relation of a test to its "subtests".

If this can be accomplished then system level tests could be sub-divided into sub-system and maybe even module-level tests. The result would be to save a great deal of testing work.

Even so, however, the techniques used now for system testing are crude and not particularly effective. Techniques like cause-effect graphing, functional decomposition, and state-space analysis seem to be limited to a few hundred "items" (i.e. causes or effects, functions, or internal states).

> Need 5: There needs to be a general theory of system testing to match that currently known (and evolving) about unit testing. This theory should provide for development of functionally based tests, possibly in terms of specifications (see below), that have error discovery and/or preventive properties.

In such a theory, if it can be developed, a test could be related directly to a function or requirement that the software system must have, and as such would lead directly to a "validating" step. This would lead to a situation in which software requirements, software design statements, and source code, could be treated in an common way. Such a development would go a long way to improving the effectiveness of formal and semi-formal methods of requirements setting.

> Need 6: There needs to be more development of methods that can be used effectively to "execute" software requirements or specifications.

Interpretation of specifications should be possible by a technique akin to that of symbolic evaluation. The result would be the ability to determine the exact nature of a specification, long before it was converted into code. Hence, errors could be prevented before being bound into code.

> Need 7: More theoretical work must be done on the notion of "program mutations," small changes to programs at the source level. If possible, a spanning set of mutations should be developed.

At present, mutation analysis lacks a theoretical basis to explain why particular sets of mutation operators are chosen, and how that choice affects the level of protection mutation analysis achieves. This does not mean, it is important to note, that mutation analysis should not be performed as a valuable adjunct to the testing process.

Methodology

Needs in the methodology area focus on the development of a range of applications of known methods, as before timephased carefully with the software life cycle.

Earlier we mentioned the need for a way of connecting specifications to tests, but even in the lack of that capability there is still much that can be done without a fully developed formal basis.

> Need 8: There must be some better method of relating specifications to software tests, so that defects can be discovered early in the software life cycle.

Even if this method is "informal" or de facto in some sense that would be far better than the techniques used today.

The integration between the so-called "production" or "synthesizing" life cycle stages and the "analysis" stages must be made more effective if testing is to provide the benefits of lower cost higher quality software.

> Need 9: Testing must be integrated into the design and coding process more thoroughly. There needs to be a method of evaluating the problems of testing during software design and/or coding, as well as methods for expressing planned-for tests.

If this can be done one can anticipate higher quality designs and, as a result, higher quality source code.

Even so, there need to be stronger methods to prevent incorporating software defects into source code. One way to do this is to have stronger and more effective design and code check rules.

> Need 10: Specific rules must be developed to check features of software design and/or source representations to impede instatiation of software defects.

Most software quality work in the past has dealt with the easiest of the problem classes, e.g. a simple batch-oriented system. Little thought has been given to the issues surrounding real-time operation.

> Need 11: Methods must be developed to deal with the questions of the quality of software which must operate in a real-time environment. This implies a requirement for both new techniques for handling communications problems and methods that work for timing-sensitive program executions.

There may be some connection between the work on communications protocols based in proof of correctness and this testing need.

The ASSERT idea, introduced by a number of writers in the early 1970's (Reference 4), is a way for programmers to indicate their beliefs about the (future) behavior of a software system. Attractive as it is, this notion is not yet fully exploited.

> Need 12: There must be more experimentation with the ASSERT idea, better in-use methodologies developed, and a body of experience made more widely available to support this powerful method of annotating programs. Also, the ASSERT notion must be extended to the software system level.

Empirical Experience

Developing and interpreting statistics has never been the most pleasant part of working with computers. For the software testing area, however, in which so much depends on the benefit/cost effects of various methodologies, getting detailed information about how varying techniques work would seem essential.

> Need 13: There must be better overall effectiveness data on every aspect of software testing and evaluation, if the intuitively clear benefit/cost rations are to be supported quantitatively.

The essential "product" of software testing -- the affirmation of existence of function combined with the demonstration of defects -- also has not yet been fully analyzed.

> Need 14: More precise error and defect discovery rates have to be developed and analyzed. In particular, the relationships between each kind of defect, its preventive tests or diagnostic procedures, and the cost of discovery must be understood in great detail.

Once these rates are known the design of effective methodologies will be somewhat easier, particularly when combined with good discovery and repair cost data.

The mutation systems in experimental development need more attention too.

> Need 15: More empirical information must be gathered on the mutation system idea. The goal of the analyses must be demonstration of benefit/cost ratios, and only secondly the development of improved tools.

The application of fault trees (see Reference 7) produces some very good effects when studying hardware/software system interactions.

> Need 16: More information and experience must be gained on the fault-tree approach to software and system quality analysis.

The most powerful method for ridding "raw" code of many subtle and potentially dangerous defects is with a static analysis system. The "lint" system on the Unix (tm) environment is an excellent example of how such a system should be built (Reference 8).

> Need 17: More experience must be developed about the use of static analyzers as a way of finding source-level defects. A primary goal should be to clearly show the benefit/cost ratios for use of static analyzers before requiring their use within the general computer software development community.

Although static analysis finds many defects, other classes of defects have yet to be analyzed in detail.

> Need 18: More experience must be collated and analyzed from areas such as: operating systems, databases, and real-time operation. The goal of this information gathering is to provide some feedback on the possible use of new approaches.

Automated Software Support Tools

Tools capture proven methodologies in specific implementations that serve the given function with a high degree of efficiency.

The difficulty is: if the methodology is not quite the one wanted, or if the environment is not the best one, then automated tools tend not to be used.

The commonest form of software testing tool, a coverage analyzer, needs much more work.

Need 19: Extended coverage analysis systems, that handle the "testing arithmetic" for more complex measures than C1 or P1, must be developed and distributed widely.

At the other end of the spectrum, there is a growing need for use of rather advanced systems.

Need 20: Symbolic evaluation systems, until recently only a laboratory-level system, should be made widely available, with two goals: (1) improvement of the user interface, and (2) additional of necessary interactive manipulation capabilities.

A simpler tool that would have a high impact is one that analyzes only a program's structure and suggests a set of test paths for that program that would meet some established coverage criteria (such as C1 or Ct).

Need 21: Portable and relatively standardized software testcase generators based on analysis of the structure of software should be developed and distributed.

Test data is often very expansive, and has to be managed with a great deal of care.

Need 22: Test control and data management systems that are particularized to the needs of software testing and analysis have to be be built and experimented with. At present, only the most rudimentary of systems have been tried.

Such a test data management system would have a number of freestanding features and facilities, but it would be even more effective if it could be included in a portable test environment.

Need 23: A fully portable software test environment should be developed. ToolPack, an initial effort in this direction, should provide valuable insights on how to organize such a system.

What happens when the software to be tested runs only on a machine for which the cost of transporting an interactive test environment is too great. One answer is to "emulate" the target hardware using a universal emulator.

Need 24: The use of dynamic microprogramming architectures should be considered as a means to minimize the cost of test analysis of one-of-a-kind systems.

Work by Clark at the University of Massachusetts in the symbolic evaluation area has led to development, almost as an offshoot, of a limited capability for automatically generating test data from the source programs.

Need 25: A practical automated test data generator, based on the use of symbolic evaluation and interactive path selection techniques, needs to be developed and distributed to a selected group to learn its effectiveness and cost.

And, in keeping with the new emphasis on fault-tree processing, there is a corresponding need for tools.

Need 26: Support and analysis tools for fault tree analysis of software and hardware/software systems must be developed.

New Technical Approaches

Some new ideas have arisen over the last decade that have never really had the opportunity for full exploitation.

Need 27: Approximations to computer arithmetic, involving limited range numbers that "fully characterize" software behavior, need experimental development.

Such an arithmetic would make it possible to "execute" the program with "real values" to determine what would happen in the more general case. "Approximate arithmetic" falls midway between symbolic evaluation and full direct execution of the tested program.

More work needs to be done on the basic implementation of systems to support domain testing, and the use of interaction with a skilled user should be emphasized.

Need 28: Domain testing, a promising technique, needs to be integrated into an interactive test environment as a way to enhance its capabilities.

The notion is an old one, and related to random tests for hardware systems. But, if successful, there could be a very high payoff.

Need 29: Statistical test systems, which choose their test data based on random number generator patterns, need to be developed on an experimental basis.

Lastly, there is an important business area that should now be addressed.

Need 30: Investigate the requirements for guaranteeing software performance, and providing for the indemnification of software against the effect of failure.

SUMMARY

Some 30 technology "needs" have been identified, in categories involving theoretical matters, methodology development, empirical analysis, and automated tools. These needs are one opinion as to what is most important to accomplish in the long run. It is clear that, even if these needs are met, much work will remain to be done.

REFERENCES

1. "ODDTE Software Test and Evaluation Project (STEP)," Viewgraphs, Panel Orientation Meeting, 29 September 1982.

2. "Transcript of Proceedings, Software Test and Evaluation," Workshop, Georgia Institute of Technology, Atlanta, Georgia, 18 March 1982.

3. E. F. Miller, Jr., "Program Testing Technology in the 1980s," The Oregon Report: Proceedings of the Conference on Computing in the 1980's, IEEE Computer Society, 1979.

4. E. F. Miller, Jr. and W. E. Howden, Software Testing and Validation Techniques, IEEE Computer Society Press, 1982.

5. The Fourth Convention of the Israel Society for Quality Assurance, Proceedings of Software Quality Assurance Sessions, 18-20 October 1982, Israel.

6. Software Research Associates, "Software Engineering Automated Tools Index," San Francisco, California, 1982.

7. H. Hecht, "Validation of Fault Tolerant Software by Means of Fault Trees," In Reference 5, (see above).

8. S. C. Johnson, "Lint, a C Program Checker," Bell Laboratories, July 1978.

"RAPID PROTOTYPING AND REFINEMENT: AN APPROACH TO SOFTWARE TESTING"

Professor Richard J. Lipton *

EECS Department
Princeton University
Princeton, New Jersey 08544

ABSTRACT

One of the possible approaches to handling the increasingly difficult testing of software is by new development methodology. We purpose an approach to software based on rapid prototyping and step-wise refinement. This approach promised to make higher quality software that is both easier to test and modify.

1. INTRODUCTION

Software development is very expensive. What we believe is that the current approaches to the development of large software systems is not likely to greatly reduce these huge costs. Further, we believe that a new methodology is needed if these costs are to be reduced. Without such fundamental changes in the way software is produced we feel that only small improvements in its costs are possible.

The key idea that we propose is that software development be viewed in a more *evolutionary* way. That is that even large software systems be viewed as evolving from other earlier systems. The central claim is that such an evolutionary approach has the potential for greatly reducing the costs of software: both in its production and in its maintenance.

2. SOFTWARE EVOLUTION

First, we must understand exactly what we mean by software evolution. By this we mean that the specification of the new system be given in a new way. In the past specifications of nontrivial systems took 100's if not 1,000's of pages of detailed specs. Such huge documents are clearly not easily used by the very people they are created for, the programmers. However, what we propose here is that they be replaced by incremental specifications. An incremental specification would *not* say what must be done, rather it would explain what changes must be done to a existing system to create the new one. That is the specs are viewed as a pair of objects: an existing system and a set of *changes*.

It is important to note that such a view is used in many other areas of specification. As humans one of the ways we can control the specification

* This research is being supported by ARO Contract #DAAG 29-80-K-0090.

of very complex systems is to not specify them from the ground up but rather by such an incremental way. Even software is informally specified in this way; what we are suggesting is it be done on a formal basis.

An incremental specification is then a way of expressing our needs in a very concise way. Examples of such specs are: (1) "make the current foo command take an arbitrary number of files instead of just one" (2) "the foo command is not fast enough; make it at least twice as fast as it is currently" (3) "the system should also keep track of the goo attributes; add a this to the data base". The key to their success is that we can envision the current system changing to respond to this new specs without having to redefine the entire system.

It is important to note what we are also not saying with this new approach. We are not saying that the new system is just a simple change to the existing one. The incremental specs can be very large themselves. of course still much shorter than the normal specs. However, the new system need not share any code at all with the old system. We are not suggesting that the new system cannot be totally new as far as its code is concerned. What we are saying is that it should not be specified from the ground up. It is important to note that incremental specification may point the way to where we can share or use preexisting code. One of the most obvious ways to control software cost is to attempt to avoid recoding the wheel, and our approach may help avoid this.

3. RAPID PROTOTYPING

What if there is no current system that is close to our needs? Then it appears that the size of the incremental specification will be no smaller than a usual one. The answer to this is rapid prototyping. Rapid prototyping is the coding up of a system, often in a very high level language, to fill exactly this role. Once such a prototype is up and running we can then begin to see what changes must be made to satisfy our needs.

The rapid prototype can be used to determine the final specification. Often it is impossible to predict what features are needed in a system without a running version. Another advantage of a rapid prototype is that we need not worry about its speed. We are solely interested at this point in the functionality of the system: what features are useful? which features are missing? which features are not needed? and so on. Later we can begin to address the issue of efficiency.

4. MAINTENANCE

Another advantage of the incremental approach is that it meshes very well with software maintenance. By all accounts maintenance is the dominating cost in large software systems. The key to why our incremental approach is so powerful is that we can view the maintenance process exactly as an application of incremental specification. Maintenance is precisely responding to incremental changes in the software. Since our approach addresses directly such specs it should be able to reduce greatly the costs of this process.

5. RESEARCH ISSUES

We feel that the incremental view of software is the key to great reductions in its cost. In order to make such an approach successful we must begin research into tools that we support such an approach. We need editors, compilers, testers and other software tools that work well with our incremental view. We also greatly need formal specification languages that allow us to easily specify systems in an incremental way.

INTEGRATION TESTING OF SOFTWARE

by

James F. Leathrum
Professor
Department of Electrical and Computer Engineering
Clemson University
Clemson, S. C. 29631

ABSTRACT

The technology of integration testing for software is considered in the context of similar problems arising in other disciplines. The impact of Ada and the Ada program support environments upon integration testing developed in terms of the reduction of risk associated with strong typing of user defined types. The case for shared testing technology is established in relationship to other technologies which have not openly shared details of successes or failures.

INTRODUCTION

The purpose of this paper is to outline some problem areas and recent developments associated with integration testing of software. Rather than a survey of research trends or a discussion of the state-of-the-art in integration testing, this paper is an appeal for technology transfer and an appeal for a new role for professional societies in integration testing.

The problem of integration testing of software can be characterized by presuming that the various modules have been verified and tested and these modules are to be "integrated" into a system. The testing issues which arise at such a juncture are of two important classes:

1) Testing the interface specifications of the modules
2) Testing against the unquantified requirements.

Testing and verification of interface specifications for modules represents an area of on-going research and rapidly developing insight due to the impact of Ada as a software design tool. The testing against unquantified requirements is an area to which we must bring some insights from other technologies.

UNQUANTIFIED REQUIREMENTS

In dealing with the issue of unquantified requirements for software, we need to acknowledge similarities and differences between software design and other design disciplines. First, we recognize that most other design disciplines share the integration and integration testing problem. The lack of quantifiable requirements is also not unusual. A bridge may be expected to operate at acceptable safety levels. A nuclear reactor may be required to have a minimum of environmental impact.

In the case of software, the unquantified requirements are likely to arise in areas where the verification methodology is not well developed. In short, the mathematical foundations have not been established. These areas are likely to be associated with real-time operation, total throughput requirements, and response time requirements. Of increasing frequency and concern as the technology develops are issues associated with system survival in the face of hardware or software component failure. Although the modules have been verified and tested, and the system meets all unquantified expectations, we are still left with the question of what happens when a memory segment fails or what are the conditions under which resource (i.e. memory) fragmentation may be fatal?

If we look to other design disciplines for guidance in dealing with unquantified requirements, we find that each such problem is characterized by:

1) An acknowledged risk in the first system integration;
2) Over-design in the earliest prototype;

and 3) A well established investigative body to follow-up on failures.

We have acknowledged and accepted risks throughout the development of flight and space travel. A certain amount of risk is acceptable in the interest of progress. It is unrealistic to expect software design and integration to be otherwise, but one is often led to believe that the so-called "structured" techniques will obviate this risk. An appropriate view of the impact of verified modules is that recovery and repair of the system is facilitated in such a design. (We have come to take such designs for granted in automobiles.)

A close corollary to this view of system integration as a risk taking activity is the assessment of reducing software to code as an experimental activity. The personal and organizational flexibility to throw away code in the interest of design improvements has proven to be very difficult to achieve. It is here that compatible design and coding media such as Ada will help to put the risks into perspective. Backing-up and re-creating a "package body" in Ada will seem like a far less serious risk than that encountered with older programming tools.

The over-design of prototypes is an issue that is difficult to treat in the isolated software context. In the overall system view of the problem we find that extra memory or extra data channels are often committed to a design as safety factors. The designs often become cumbersome and inflexible as a result. In the next section, the case of the Army's Tactical Computer System will be considered as an example of such an overdesign.

The recognition of the need for an investigative organization for software failures is perhaps the most serious and most immediate concern in software integration testing. Even the most superficial view of the problem in the context of other disciplines would lead to the conclusion that the need is obvious. Those scientific disciplines which have been most open in the exchange of such information have also developed the most rapidly. Most notably, the communications and the electronics industries have established open forums for the exchange of what would appear to be potentially proprietary information. The automobile industry, on the other hand, has not been characterized by open sharing of technical information. Although the point is not immediately relevant to the investigation of failures, the commitment to the exchange of technical information is a key element to the success of such investigations.

The mechanism by which we manage the investigation of software failures is also an issue for which some insight is available from other disciplines. One finds three common modes of review:

1) A panel established by the appropriate professional society.
2) A government review board.
3) A "blue ribbon" panel. Such a panel was formed to investigate the Three Mile Island nuclear reactor failure.

In case of software, the first two mechanisms need careful consideration. The "blue ribbon" panel would probably only be invoked in the case of a failure of most serious consequences. The most likely professional society to take a role in such investigation would be the ACM since they already have in place the means of publishing the results, and the ACM has already established such panels to discuss curriculum matters. The most obvious model of a government panel is the National Transportation Safety Board. Given the DoD commitment of software and the pending DoD sponsored software initiative, it would appear that the panel should be established within the DoD.

THE TACTICAL COMPUTER SYSTEM: A CASE STUDY

Occasionally a systems development project arises which provides a useful case study in how pathologies occur during the software life-cycle. The U. S. Army has had a Tactical Computer System (TCS) under development for nearly a decade. Both the hardware and software which have been designed as part of the project have been viewed throughout as prototype, generic models. Thus, nothing that is written here should be construed as being critical of the designers themselves.

The software for the TCS was developed during early phases of the system development using tools which by present day standards would be judged exceedingly primitive. The modules were subjected to extensive laboratory verification and testing. These modules included various modules which were subordinate to the operating system such a message formatting and memory management. Subsequent to further hardware design which included militarization of the hardware, applications software development and development testing commenced. During the development testing, the system was judged inoperable, and the decision was made to redesign the software under a new contract.

One could approach the TCS experience with a "so what's new?" assessment. But, the interesting aspect about this experience is that it presents an excellent opportunity to examine the scenario which led to the failure of the first software. Even, if for contractural reasons, it is agreed a priori that the original design was as good as could be expected at the time it was executed, there remains a great deal to be learned from investigating what happened.

A few overall conclusions can be made directly from the TCS experience. First, the software failure was clearly a failure in software integration, and the failure was not recognizable until integration testing was undertaken. Second, early perceptions of integration problems which were addressed by adding more hardware were the same actual integration problems which proved fatal to the system. Third, over-design of the communications hardware proved to be an unfortunate diversion of the technical manpower from the systems implementations problems.

The problems just cited were further complicated by the progression of technological developments which were occurring concurrently with the TCS development. Since the original software was implemented many years before the hardware was militarized, the tools which were used were primitive (assemblers) and the rationale for the software design was either lost or it never existed. By the time the system reached development testing, it was clear to almost all concerned that if any re-design was needed to make the system operational, the whole software system would need re-designing not just the bad parts. And so, it happened.

Without a careful examination of all that happened to the TCS, it would be inappropriate to draw sweeping conclusions. However, it is possible to project the general types of recommendations which might be forthcoming from such an examination. One might expect recommendations regarding standards of design documentation and design rationale to be formulated. It is reasonable to suppose that project management guidelines might be established to insure that the software and hardware life-cycles are kept in technological phase with each other. Finally, one would expect to find additional support for the use of high level tools for software design and implementation.

THE IMPACT OF ADA UPON SOFTWARE INTEGRATION

Since one of the issues which arose in the previous section and throughout the notes of the preliminary session of this workshop was the role of higher level tools such as Ada on software design and testing, it is appropriate to speculate on the role of Ada in integration testing. The issue is particularly relevant since Ada is designed to specific "encapsulation" requirements for software. To meet this requirement, the language contains a construct called the "package".

When viewed in the context of other design disciplines, the issue of encapsulation seems to be a new concept. However, we would note that other designs such as structures, electrical systems, and plumbing systems are treated in a modular fashion. The interface issues are resolved by utilizing the laws of nature as embodied in the properties of materials such as strength, insulation, etc. The important feature of such designs is that the laws of nature are scale independent, and thus they can be confirmed in the laboratory. The materials can be expected to perform according to the results of the small scale tests.

In the case of software, we do not have any such laws of nature at hand. We have very little choice but to minimize the interfaces between the various modules, and most importantly, we must build the interface laws into our design medium. Thus, we have encapsulation and packages.

In assessing the encapsulation features of Ada, one finds a mixed blessing with respect to system integration problems. The simple fact that the package body (i.e. the implementation) is a separate entity from the package specification (i.e. the interface design) offers the possibility of repair of parts of a system during integration. Since the various parts are not dependent upon the details of the implementation, it is apparent that those details can be refined without upsetting the integrity of the whole system. Although system failure is not precluded by such a linguistic feature, it is now possible to keep a system alive much longer and thus, learn much more before the ultimate failure occurs. It is the risk associated with the next prototype which is reduced by the Ada package.

In addition to the package feature of Ada, one finds that Ada also lends the designer some assistance in managing the evolution of the modules of a software system. The library management features of the language and the plans for a well defined program support environment will make it possible for the designer to ensure that proper versions of each modules are selected during the preliminary "system builds". As an added side effect of these features, users have found that they tend naturally to create only small modules in Ada.

The other side of the impact of Ada derives from features of the language which preclude detailed compiler enforcement of interface designs. Most notable, the "overloading" feature allows a name to take on a number of different meanings at a particular point in a design. This is an acknowledged very powerful feature of the language, but it is also a potential source of over confidence in a system integration. This over confidence may not be illuminated except by the most careful and detailed integration testing. Out of fairness to the designers of Ada, it should be observed that it is this same overloading mechanism which allows the extension of a software system at any level of abstraction without polluting the name space to be managed by the programmer. Thus, complex arithmetic may be added without inventing new symbols for the arithmetic operations.

Ada offers additional protection to the systems designer through the generalized type definition mechanism. In Ada, it is possible not only to control the objects associated with a type, but it is also possible to obtain explicit control over the allowable operations. Several of the older programming systems included the former capability, but it seldomly aided the programmer prior to integration testing in avoiding inappropriate use of objects. The "private type" and the "limited private type' of Ada allow the designer to specify strict control over permissible use of objects, and furthermore to have the compiler enforce the restrictions.

The advent of Ada has presented another opportunity to test ideas about software testing which emerged during the 1970's. The Department of Defense has proposed to validate Ada compilers by subjecting them to a test suite. This test suite contains about 1300 programs each of which is specific to a particular feature of the language to be implemented by the compiler. The design of the test suite is based upon the presumption that the major features of the language are orthogonal and free of interference with each other. The likelihood that compilers will pass this kind of test suite and still remain inoperable in practice needs to be recognized. The problem comes back to one of dealing with integration of all the features of the language into a single compiler. On the more positive side of the on-going testing of Ada compilers, it should be noted that one class of tests (Class L) will confirm that the systems integration tools of the language are implemented.

CONCLUSIONS

The preceeding discussions have highlighted some of the problems which can be foreseen in the development of a systematic approach to integration testing of software. The most pressing need is for technology transfer from other disciplines with respect to ways of dealing with design failures. Closely allied with this conclusion is the expectation that the software integration problems which we have may not be solved through sharply focussed research, but instead, may be solved by a combination of risk taking and careful examination of failures. The continued development of Ada deserves attention for what it offers to the designer for the resolution of integration testing problems.

ABOUT THE AUTHOR

James F. Leathrum is a professor of computer engineering at Clemson University. He is currently consultant to the Western Digital Corporation where he has participated in the design of an Ada compiler and continues to participate in the integration testing of the same compiler. He is also a consultant to the U. S. Army Material System Analysis Activity where he has participated in the design and analysis of software for fire control systems. He is the author of Foundation of Software Design, a book which will appear soon through the Reston Publishing Company.

STATUS AND DIRECTIONS FOR SOFTWARE TESTING AND EVALUATION TOOLS

Leon Osterweil

Professor and Chairman
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309

Telephone: (303) 492-8787 or
(303) 492-7514

ABSTRACT

This paper advances the point of view that there are a number of very promising tools and techniques for the testing and evaluation of computer software, but that at present, for a variety of reasons, it is difficult and often impractical to exploit their innate capabilities effectively. The major obstacles to effective exploitation are seen to be the difficulty and capital required to bring superior tools into widespread and common use, the current predisposition of large organizations in the private sector to refuse to disclose, no less share, their best tools, and a lack of understanding of how to integrate the best of current and proposed tools into effective total systems for the support of testing, verification, documentation and evaluation.

The paper goes on to suggest that what is most needed now is a mechanism for facilitating experimentation with the best currently available tools and techniques. It seems that this can best be done through the construction of collections of tools integrated in such a way as to facilitate the rapid configuration of new and/or competing tools and approaches through an architecture stressing the composition of larger tools out of smaller tool fragments, and the centering of the testing activity around a central data base. Such an experimental test bed for testing and evaluation tools should facilitate the process of deciding which tools and techniques are superior and should also facilitate the widespread familiarization of large Dod communities with the tools and techniques. It is expected that this would lead to more rapid adoption and implementation of superior tools and techniques.

EXECUTIVE SUMMARY

At present it seems that the greatest need in the area of software testing and evaluation tools is for a concerted program of development, packaging, evaluation and integration of these tools. The past decade has seen the creation of a large number of such tools whose approaches to the underlying problems of testing, analyzing, verifying and documenting software items has been refeshingly broad. Virtually all of those tools remain either experimental curiosities or special purpose tools, of value only in very restricted contexts. Some of these tools show clear promise of being exceedingly helpful in detecting errors in software or in demonstrating the absence of errors. Some have even been applied with considerable positive effect. Few, if any, however, have been broadly distributed and exploited. Those that have, tend to have been the most simple-minded.

Some of the reasons for this unfortunate state of affairs are not hard to discern. Software tools are themselves very sophisticated items of software. Thus they are expensive to produce and package to the point at which they can be considered suitable for large user communities. Many of the most innovative tools have been produced in University contexts or in private research laboratories, where there has been little incentive and/or little financial support for such packaging. Adequate funding for development of these tools could bring them to the point at which they could be used and evaluated by broad based user communities. Many would doubtlessly prove to be useful and effective.

Perhaps even more serious, there has been little opportunity for carrying out the sort of definitive, comparative analyses which are necessary if potential users are to successfully select appropriate testing and evaluation tools. As noted earlier there is a very wide range of approaches represented by currently available tools. This is gratifying from a scientific point of view, but represents a serious obstacle to the systems analyst or project manager who wishes to apply just the right tool or combination of tools to a given software item or project. Differences of effectiveness and efficiency between comparable tool capabilities have not been measured definitively in scientific studies. More important, however, there is currently only a dim understanding of the relative strengths and weaknesses of the different approaches to testing and evaluation. There is some understanding of how dynamic testing and static analysis techniques can complement each other, for example. This is a far cry, however, from the needed understanding of which technique or combination of techniques to apply to meet certain very specific testing and evaluation objectives. This, in turn, is itself a far cry from knowing which specific tools to acquire and apply. In fact, it seems clear that, although there is currently a very large variety of tool capabilities available, such systematic scientific testing and comparison of them will certainly reveal that there are some important capabilities that are still in need of development. For example, there is clearly a shortage of tools for testing and analysis of real-time and concurrent software.

Recent work in the area of software development environments seems pivotal in that it is effecting the creation of standard harnesses within which diverse testing and analysis tools can be installed for comparison and measurement. Some of these environments go even further in enabling the creation of tools as sequences of lower level tool fragments. The flexibility of this approach should encourage the creation of new classes of tools in a context within which they can be evaluated and either rejected or improved. This approach should also facilitate the sorts of experimentation needed in order to better understand the interrelations among testing and evaluation capabilities.

1. TOOLS FOR TESTING, EVALUATION, VERIFICATION AND ANALYSIS OF SOFTWARE

This section will attempt to present a brief overview of the broad classes of tools for testing and evaluation of software that have been produced, and will attempt to compare and contrast the approaches which have been taken. Specific tools will be named and referenced in places.

1.1 Class One - Dynamic Testing and Analysis

The terms dynamic testing and dynamic analysis, as used here, are intended to describe most of the systems known as execution monitors, software monitors and dynamic debugging systems ([Balz 69], [Fair 75], [Stuc 75] and [Gris 70]).

1.1.1. Testing Tools

In dynamic testing systems, a comprehensive record of a single execution of a program is built. This record -- the execution history -- is usually obtained by instrumenting the source program with code whose purpose is to capture information about the progress of the execution. Most such systems implant monitoring code after each statement of the program. This code captures such information as the number of the statement just executed, the names of those variables whose values had been altered by executing the statement, the new values of these variables, and the outcome of any tests performed by the statement. The execution history is saved in a file so that after the execution terminates it can be perused by the tester. This perusal is usually facilitated by the production of summary tables and statistics such as statement execution frequency histograms, and variable evolution trees.

Despite the existence of such tables and statistics, it is often quite difficult for a uman tester to detect the source or even the resence of errors in the execution. Hence, many ynamic testing systems also monitor each stateent execution checking for such error conditions s division by zero and out-of-bounds array refernces. The monitors implanted are usually prorammed to automatically issue error messages mediately upon detecting such conditions in rder to avoid having the errors concealed by the ulk of a large execution history. The monitors re positioned so as to assure that any occurrence f errors will be detected immediately before it uld occur in the actual execution of the proram. To a human observer it is often obvious at many of these probes are redundant. There are ways in which automated analysis can be used to suppress such probes.

Some systems ([Fair 75], [Stuc 75]) additionally allow the human program tester to create additional monitors and direct their implantation anywhere within the program. The greatest power of these systems is derived from the possibility of using them to determine whether a program execution is proceeding as intended. The intent of the program is captured by sets of assertions about the desired and correct relation between values of program variables. These assertions may be specified to be of local or global validity. The dynamic testing system creates and places monitors as necessary to determine whether the program is behaving in accordance with asserted intent as execution proceeds.

These assertions are designed to capture the intent of the program and explicitly state certain non-trivial error conditions, to which the program may be particularly vulnerable. It should be clear that dynamic assertion verification offers the possibility of very meaningful and powerful testing. With this technique, the tester can in a convenient notation specify the precise desired functional behavior of the program (presumably by drawing upon the program's design and requirements specifications). Every execution is then tirelessly monitored for adherence to these specifications. This sort of testing obviously can focus on the most meaningful aspects of the program far more sharply than the more mechanical approaches involving monitoring only for violations of certain standards such as zero division or array bounds violation.

It is important to observe, that the benefits of dynamic testing can only be derived as the result of heavy expenditures of machine storage and execution time. The next subsections will show that storage and execution time costs can be effectively reduced by employing static analysis and symbolic execution techniques in conjunction with dynamic testing. More important, however, is the observation that, because dynamic testing focuses on the minute examination of the history of a single program execution, its results are relevant to that execution, but may not be relevant to other program executions. Hence dynamic testing is able to detect the presence of errors, but it is not clear that it is a useful technique in demonstrating the absence of errors.

Because it is assurance of the absence of errors that would seem to be most important, a great deal of effort has been devoted to studying how sets of dynamic tests of a piece of software can be devised to at least raise the level of confidence in the correctness of the software to an acceptable level. At least three different approaches to doing this are recorded in the literature, although only one of the approaches seems to be adequately supported by tools. These are summarized next.

1.1.2. Testing Strategies

The first approach, and the one which seems to have the most straightforward appeal is advocated by Howden [Howd 80]. This approach, called Functional Testing, suggests that an adequate set of tests for a program can only be created by very careful examination of the functional specifications for the program. The highest level functional specifications must be determined and then carefully broken down into specifications for the lower level functions which are used to actually effect this highest level functionality and which are implemented in the highest level routines of the software to be tested. These lower level functions are in turn themselves effected by still lower level functions implemented in still lower level routines. These functions must also be carefully specified. This process of determining how the highest level functions are implemented by successive layers of lower level functions is the necessary prelude to the process of determining just what test data sets must be fed to the software program in order to adequately test it.

The test data sets are chosen so as to exercise thoroughly each of the functions at each of the successive levels of the software. Howden describes criteria to be used in determining just how to decide when a function or subfunction has been tested thoroughly. What is less clear is how to automate this process with tools. More significantly, it is still less clear how to use tools to help in the process of decomposing higher level functions into lower level subfunctions. This appears to be, in essence, the software design process. Thus the issue of providing adequate tool support for Howden's Functional Testing approach seems to be very much intertwined with the issue of providing support for the earlier phases of software development, especially the design phase. This issue will be addressed again in a later section of this paper.

A second major approach to the problem of creating thorough test sets is what might be called the Structural Approach. This approach involves modelling of the software as a graph or coordinated set of graphs. As such it has elements of static analysis, a technique to be characterized in the next subsection of this paper. The major element of static analysis which structural testing employs is reliance upon the creation and analysis of a program representation called the flowgraph. The flowgraph is a structure in which each procedure or subprogram of the subject software is modelled by a set of nodes and edges. In particular, each of a procedure's execution units is modelled as a node and each possible transition from one execution unit to a successor is modelled as an edge.

Most authors who advocate structural approaches agree that a set of tests of a program should not be considered complete until and unless the set has assured the execution of every node and edge of every flowgraph representing the various procedures and programs comprising the subject software. This seems a minimal testing regimen, but it is agreed that it is far from exhaustive enough to offer good assurance of the absence of error from a piece of software. In particular, it only assures that the functionality embodied in each statement will be exercised at least once and that the logic embodied in every flow of control alteration statement will be exercised only enough times to assure that every possible flow of control alternative will be selected at least once. Perhaps the most serious flaw in this approach to assuring thorough testing is that it is far too mechanical, treating a piece of software as being solely a structural object and not a functional object. The most appealing aspect of Functional Testing (just described) is that it recognizes that a piece of software is created to perform certain functional transformations. Testing should be directed towards seeing that those functions are correctly implemented.

Some investigators (e.g., [Rich 81], [Whit 80], and [Weyu 80]) have attempted to incorporate into the structural testing context a recognition of the need to treat software under test as a functional transducer. They have observed that a piece of software can be thought of as a functional transducer which performs a different transformation on each of a number of different subspaces of the software program's input space. The decomposition of the input space into subspaces is performed by the logic of the program's flow of control alteration statements, and the different functional transformations are achieved by composition of the various executable statements in the program in different orders.

These investigators suggest that thorough testing can be achieved if the decomposition effected by the program is first determined, and then used as a guide to the creation of test data sets which assure that the functional transformation performed for each input subspace is carefully exercised. Their papers suggest the tools that are needed to do this. Symbolic execution methods (to be described more fully in a subsequent subsection) are central to determining the set of input subspaces. They are also useful in creating the data sets required to exercise the code which implements the various functions computed for each of the various subspaces. As will be seen in the subsequent subsection, there are some research tools which have been built which are capable of assisting in the process of constructing such data sets. There are substantial obstacles, both pragmatic and theoretical, to the creation of truly effective tool supports, however.

The third approach to assuring effective testing is called Mutation Analysis [DeMi 78]. This approach is quite novel in that it offers an effective way of quantitatively assessing the thoroughness of the testing which is achieved when a program is exercised with a given set of test data. Mutation Analysis assumes that a given piece of software is to be tested by a given set of test data. This method entails the creation of a very large set of "mutants" of the original software, which are carefully created to reflect the gamut of errors which a "reasonable programmer" might commit. This gamut of mutant versions of the original program is fed all of the test cases in the given input data set. The outputs obtained are then compared to the outputs of the original (presumably correct) program. As soon as any difference is observed, the mutant program is discarded. If some mutants remain undiscarded after all sets of test data are executed then these mutants are examined carefully to see if they are actually functionally different from the original program. The number of functionally different mutants which remain undiscarded has been found to be a very reliable measure of the thoroughness of the test data set. If an unacceptably large number of nonequivalent mutants has remained undiscarded, more test data must be added to the original set, with the goal of causing some of the undiscarded, nonequivalent mutants to give different execution results from the original, thereby causing them to be discarded. This process of creating new test data sets to "kill off" mutants is to be continued until an acceptably low number of undiscarded nonequivalent mutants remains.

Tool systems for supporting the creation and testing of mutants have been built and experimented with extensively. Thus, this approach in contrast to the others, is well supported by tools. The tool sets do not, however, address the problem of automatically creating test data sets as directly as is suggested by the structural testing approach. It seems that there is a good opportunity for combining the strong points of these two approaches into a system for supporting the effective generation and evaluation of test data sets. This issue will be addressed in a subsequent section of this paper.

In the next subsections of this paper we describe two alternative approaches to testing. These approaches attempt to assure the absence of errors in a piece of software by analyzing its structure instead of exercising it with testcases.

1.2 Class Two - Static Analysis Tools

In the category of static analysis tools, we include all programs and systems which infer results about the nature of a program from consideration and analysis of a complete model of some aspect of the program. An important characteristic of such tools is that they do not necessitate execution of the subject program yet infer results applicable to all possible executions.

A very straightforward example of such a tool is a syntax analyzer. With this tool the individual statements of a program are examined one at a time. At the end of this scan it is possible to infer that the program is free of syntactic errors.

A more interesting example is a tool such as FACES [Rama 75], Softool 80 [Mehl 81], or RXVP [Mill 74] which performs a variety of more sophisticated error scans. These tools, for example, perform a scan to determine whether all procedure invocations are correctly matched to the corresponding definitions. The lengths of corresponding argument and parameter lists are compared, and the corresponding individual parameters and arguments are also compared for type and dimensionality agreement. By comparing every procedure invocation with its corresponding definition in this way it is possible to assure that the program is free of any possibility of such a mismatch error. Note that this analysis requires no program execution, yet produces a result applicable to all possible executions. This sort of analysis, requiring a comparison of combinations of statements, can also be used to demonstrate that a program is free of such defects as illegal type conversions, confusion of array dimensionality, superfluous labels and missing or uninvoked procedures.

Data flow analysis is a still more sophisticated form of static analysis which is based upon consideration of sequences of events occurring along the various paths through a program. As such it is capable of more powerful analytic results than combinational scans such as those just described. The DAVE System [Oste 76], [Fosd 76] is a good example of such a tool. This system examines all paths originating from the start of a Fortran program and is capable of determining that no path, when executed, will cause a reference to an uninitialized variable. DAVE also examines all paths originating from a variable definition and is capable of determining whether or not there is a subsequent reference to the variable. A definition not subsequently referenced is called a "dead" definition. Hence DAVE is also capable of showing that a Fortran program is free of dead variable definitions.

Data flow analysis is based upon examination of a flow graph model of the subject program. The flow graph of every program unit is created and its nodes are annotated with descriptions of the uses of all variables at all nodes. Nodes representing procedure invocations cannot be annotated in this way immediately. For such a node a data flow analyzer like DAVE would first determine the presence or absence of uninitialized variable references and dead variable definitions in the procedure represented by the node. This can be done by using data flow analysis algorithms such as LIVE and AVAIL [Hech 75] to efficiently determine the usage patterns of the program variables along the paths leading into or out of the procedures start node. Having done this, it is possible to complete the data flow analysis of the calling program. The details of this procedure can be found in [Fosd 76].

Thus static analysis can be used to determine the presence or absence of certain classes of errors and to produce certain kinds of program documentation. Hence it is useful as a complement to a testing procedure and offers some limited verification capabilities. It is also useful in supplying limited forms of documentation (e.g., the input/output behavior or a procedure's parameters and global variables). There is currently ongoing research which indicates that static analysis, particularly data flow analysis, can be used to both verify and test for wider classes of errors, such as concurrency errors, (e.g., [Tayl 80]) as well as to produce additional forms of documentation.

Of particular interest is the possibility of using static data flow analysis to suppress certain of the probes generated by dynamic assertion verification tools as part of a comprehensive test procedure. Many of these probes generated by dynamic test aids are redundant. Their presence adds to the size and execution time of a test run yet has no diagnostic value. Hence an automatic procedure which removes them makes testing more efficient. It also serves to focus attention on the importance of exercising the remaining probes. Sometimes it is possible to remove all the probes generated by an assertion or single error criterion. In this case, it has been de facto demonstrated that the error being tested for cannot occur, and this aspect of the program's behavior has been verified. This perspective shows how testing and verification activities can be coordinated with each other, through the integrating medium of static analysis.

Although the synergism of these two techniques seems apparent, no tools to exploit this synergism have yet been constructed. This would seem to be an important step to take, and it is expected that it would be facilitated by the creation of testing and analysis environments containing coordinated modular tool fragments.

1.3 Class Three - Symbolic Execution Tools

Symbolic execution is the process of computing the values of a program's variables as functions which represent the sequence of operations carried out as execution is traced along a specific path through the program. If the path symbolically executed is a path from a procedure start node to an output statement, then the symbolic execution will show the functions by which all of the output values are computed. The only unknowns in these functions will be the input values (either parameters in the case of an invoked procedure or read-in values when a main program is being symbolically executed).

A small number of symbolic execution tools has been built [Howd 78], [King 76], [Clar 76]. These tools mechanize the creation of the formulas and maintain incremental symbol tables. They employ formula simplification heuristics in an attempt to forestall the growth in size of the generated formulas and foster recognition of the underlying functional relations. (It should be noted, however, that these simplifiers do not take roundoff error into account and, therefore, may misrepresent the actual function computed by a sequence of floating-point computations).

The foregoing indicates that symbolic execution is an excellent technique for documenting a program. Symbolic traces provide documentation of the actual functioning of a program along any specific path. In order to use symbolic execution as a technique for testing and verification however, it is necessary to augment the technique with a constraint solving capability.

In order to clarify this, let us begin by observing that the above described functional behavior occurs only when the given path is executed. In general, however, a given program can execute an (often infinite) variety of paths, depending upon the program°s input values. The conditions under which a given path is executed can often be determined by symbolic execution and constraint solution. A given path will be executed if and only if all of the predicates attached to all of the path edges are satisfied. Unfortunately, a simple textual scan will express these constraints only in terms of the variables within the statements. Thus the constraints will in general not show their underlying interrelations. If the constraints are expressed in terms of the formulas derived through symbolic execution of the path, then a set of constraints all expressed in terms of the program's input values is obtained. Any solution of this set of constraints is a set of input values sufficient to force execution of the given path. This process of solving simultaneous constraints generated by symbolic execution is the process alluded to in Section 1.1.2. which can be used to decompose the input space of a program and test the functions executed for each subspace.

It is important to observe that some constraint systems are unsatisfiable, indicating that the path spawning them is unexecutable. This is important information as static flow analyzers sometimes detect "errors" along unexecutable paths. These errors are ephemeral and should not be reported. No less important is the observation that the problem of determining a solution to an arbitrary system of constraints is in general unsolvable. Hence we must not expect that this potentially useful capability can be infallibly implemented. Experimentation has indicated, however, that for an important class of programs the constraints actually generated are quite tractable [Clar 76]. A great deal more of this sort of experimentation is urgently needed. Testing and verification capabilities can also be achieved by attempting to solve constraints embodying error conditions and statements of intent. Thus we see that the symbolic execution/constraint solving technique is a powerful testing aid. It should be noted that the ATTEST system [Clar 76] implements most of the capabilities just described.

Perhaps the most important use of symbolic execution/constraint solution is as a technique for verifying assertions of functional relations between program variables. We saw that static analysis is quite adept at inferring all the possible sequences of events which might arise during execution of a program, and that by comparing these with specifications of correct and incorrect sequences, testing and verification capabilities are obtained. When the statements of correct behavior are couched as predicates involving program variables, however, symbolic execution/constraint solution is most useful. This is not surprising, as symbolic execution is a technique for tracing and manipulating the functional relations between program variables.

Using symbolic execution it is sometimes possible to synthesize recurrence relations among program variables, which might then be solved to yield closed form formulas relating the values of variables. These formulas could then be compared to assertions of intent. This capability rests heavily upon being able to draw on results from finite mathematics. Cheatham has created a tool with impressive inferential capabilities of this sort [Chea 78], although the problem of determining the closed form of a recurrence is in general intractable. Also required here is the ability to recognize when two formulas are equivalent. This problem is likewise intractable in general.

Another drawback to the use of symbolic execution is that it generally employs a simplistic model of real arithmetic under which the expressions X/2.0 and 0.5*X are considered equivalent. Because of the peculiarities of floating point hardware, however, the two formulas will often evaluate to different values. Hence the results of symbolic verification and dynamic verification may differ.

Despite these various limitations it seems clear that symbolic execution/constraint solution can be used to yield impressive documentation, testing and verification capabilities. Perhaps these limitations can be put in better perspective by observing that symbolic execution and constraint solution are the basic techniques used in formal verification or so called "proof of correctness" ([Elsp 72], [Lond 75], [Hant 76]).

In formal verification the intent of a program must be captured totally by assertions imbedded according to the dictates of a criterion such as the Floyd Method of Inductive Assertions [Floy 67]. The correctness verification is established by symbolically executing all code sequences lying between consecutive assertions and showing that the results obtained are consistent with the bounding assertions. The consistency demonstration is generally attempted by using predicate calculus theorem provers rather than constraint solvers as discussed here.

It is crucial to observe, that these theorem provers are subject to the same theoretical limitations discussed earlier. The undecidability of the First Order Predicate Calculus makes it impossible to be sure whether a theorem is true or false. Hence we cannot be guaranteed of an answer to the question of whether or not a symbolic execution will yield results consistent with its bounding assertions. Furthermore, the symbolic execution may make simplifications and transformations of real formulas which do not recreate the functioning of floating point hardware. These and similar limitations of formal verification have long been acknowledged. Yet still formal verification is rightly regarded as a useful technique capable of increasing one's confidence in the functional soundness of a program. This is sort of the sense in which the symbolic execution/constraint solution technique just discussed should be considered worthwhile, as well.

In fact, this technique is of more worth to a practitioner than formal verification, because of its flexibility. As already observed, formal verification requires a complete, exhaustive statement of a program's intent. The technique just described focuses on attempting to justify or disprove the validity of individual assertions. This gives the practitioner the ability to probe as many of the various individual aspects of a program as may be desired. From this perspective formal verification can be viewed as the logical, orderly culmination of a process of verifying progressively more complete assertion sets.

2. REASONS FOR THE LACK OF WIDESPREAD USE OF EFFECTIVE TOOLS FOR TESTING AND EVALUATION OF SOFTWARE

It is unfortunate and surprising to most observers that, despite the impressive list of capabilities just described, software tools have met with poor acceptance. Thus the promise of computerization of software production has, for the most part, remained just a promise. In order to understand this, it is important to carefully examine the notion of what a tool really is. According to most dictionary definitions a "tool" is a device which is comfortably and conveniently useful in facilitating or multiplying human work. While it seems clear that the multitude of tools which we have produced are capable of facilitating or multiplying the work that users will do, it is also clear that most of the tools which have been built are either uncomfortable or inconvenient or both.

There are some important classes of software tools which are successful and are, in fact, tools in the truest sense of the word. Compilers, loaders, assemblers, and operating systems are certainly tools. They perform the invaluable service of enabling us to write programs in a higher level language, to reuse libraries of already written procedures, to access and store large files of data, and to share access to the computer. All of this is done through the use of software which clearly multiplies and facilitates human effort. Moreover, most users of this software rarely think too much about when and how to use it. Despite occasional nasty surprises when this software fails, or periodic occasions upon which some unfamiliar features of the software must be learned, we generally use these software systems pretty much without a great deal of conscious thought. This software is comfortable to use, and thus these systems deserve to be called software tools.

It is important to reflect upon why these systems have achieved the status of tools in order to understand what must happen in order for the large universe of testing and evaluation support systems to achieve the status of software tools. It is clear that to a large extent compilers and operating systems have become comfortable and familiar simply because of their longevity. At first, these software systems were new and unfamiliar to users. At that time they experienced the same sort of rejection that we see in the case of many software - assistance systems today. Over a period of decades, however, their benefits became recognized, their proper utilization became better understood, and the corresponding increases in comfort and convenience led to acceptance. It is important to observe, also, that during this period the quality of the software tools themselves was slowly improved. It is a rare software product indeed which is reliable, robust and well documented right from the start. Early compilers, loaders and operating systems were no exception. Although relatively reliable today, they were not so at first, and their acceptance and transformation into tools took place only after a period of many years during which they were made robust, reliable and well documented.

Thus it seems that our present crop of testing and evaluation support systems is destined to evolve into a set of software tools given the time in which to improve and in which the using public will come to understand the true merits and proper application of this software. Here, unfortunately, we arrive at a problem. The user public was willing to tolerate years of poor compilers and operating systems because it understood the role and purpose of these systems, and because it, on balance, believed that, when perfected, these systems would lead to major productivity gains. The same cannot be said for many of our systems today. The sheer variety of such systems poses a problem, as does the more fundamental lack of understanding and systematization of testing, verification, documentation and evaluation as disciplines. Compilers and are clearly useful to humans because they assist in the necessary but tedious dealings with the actual hardware. As such they are common denominators, as all software writers need this aid at a well-understood, agreed upon time during software production. Our more modern test and evaluation aids have been built to address the galaxy of problems which arise before, during, and after the actual execution of the coded program. As such they form a bewildering array whose proper times and modes of application are neither widely understood, nor widely agreed upon. In this, they are designed to help with activities that have previously been largely the province of humans guided and assisted only by intuition. Thus it should be expected that there will have to be a lengthy period during which the bounds of their efficacy are studied and delineated. It appears that this period is still just beginning.

Thus it seems that if our software-assistance systems are to achieve the status of true software tools, it will be necessary to be sure that the proper usage contexts for these systems have been established and agreed to. If this can be done, then it is likely that the using public will have the patience to wait through the laborious and necessary process of incremental improvement which will ultimately lead to systems whose quality is adequate to assure acceptance.

It is evident that at present, due to lack of adequate utilization of testing and analysis aids, it is impossible to be sure of the best usage contexts for these aids. The discussions of these aids in section 1 of this paper should, however, have made clear that the testing, analysis and verification capabilities described there complement each other in important ways. More specifically, the discussion in that section strongly suggested ways in which these different capabilities can be synthesized in support of such major software development activities as testing, verification and documentation. It is urgently necessary that specific configurations of these tool capabilities be made available to users for experimental evaluation so that we can begin to understand what tool combinations are most effective and are the best prospects for production development and application.

Hence, in the next subsections there will be brief expositions about how to support documentation, testing and verification with combinations of the capabilities described in Section 1. Section 3 will then address the question of how frameworks for experimental evaluation of such tool capabilities can be established.

2.1 Documentation

A complete set of program documentation must fully describe the structure and functioning of the program. Clearly such a set must describe a wide variety of aspects of the program. At present it seems that certain of these items of description must inevitably be supplied by humans. The previous section of the paper has shown, however, that some documentation can be generated by tools. This documentation is, moreover, probably more reliably and cheaply done by such tools. In addition, if some documentation is done by tools, the remaining documentation is likely to be done more carefully by humans, thereby suggesting the possibility of greater quality and reliability.

The first section of this paper suggests that static analysis tools should be used first to create such documentation as cross reference tables, variable evolution trees, and input/output descriptions of individual variables and procedures. Symbolic execution tools can be used next to create descriptions of the functional effects of executing various paths through the code. With constraint solution, a complete input/output characterization of the code could be obtained. Performance characteristics can be measured and documented with the aid of a dynamic testing tool. In the next section of this paper it shall be proposed that all this documentation be stored in a central data base, forming a skeleton of the complete documentation. Editors and interactive systems might be used to gather from humans such additional items of documentation as text descriptions of variables and procedures.

2.2 Testing

We have seen that probe insertion tools can be very effective in instrumenting software for the automatic detection of wide classes of errors. We have also seen that tools can also be used to design and evaluate the effectiveness of a testing regimen. Tools can also be used to focus the testing effort on paths and situations which appear to be more error prone. This is done by elimination of probes which were created to test for common programming errors and for adherence to explicit assertions. We saw that many probes can be removed by application of progresssively stronger (and more costly) static analysis. Some remaining probes may be removed as a result of symbolic execution/constraint solution. We saw that these probes are likely to be the more substantive ones, monitoring for adherence to asserted functional intent. Their removal constitutes significant verification, but it can be expected that the cost of this will be relatively high.

Finally dynamic test tools can also be used to gather definite information about the existence and sources of error in the program. As already noted, testing can only show the presence of error in a test case, and even a simple program may have an infinite number of possible test cases. Hence the tool aided procedure just outlined has added importance in that it helps suggest test cases - namely those designed to exercise probes not analytically removed.

Clearly the foregoing summary indicates the power and importance of combining static analysis and symbolic execution tools with testing aids. No current systems do this effectively, and there is need for embarking on such large scale tool consolidation immediately.

2.3. Verification

Verification is the process of demonstrating the absence of errors. As such it should not be undertaken until and unless testing has failed to uncover errors. Thus it is a less frequent, more critical process, usually warranting greater expense and thoroughness.

A verification activity should start out like the testing activity just described. The first step is to suppress error testing probes and probes resulting from assertions. Static analysis can be used to suppress some probes, but the most significant probes probably can be removed only by symbolic execution. Verification is achieved on an assertion-by-assertion basis only when all probes generated by a single assertion have been removed. In this way stronger more complete verification can be obtained incrementally at greater cost and effort. Complete formal verification can be attempted if desired as the culmination of this process.

A final word should be said about the need for both verification and testing. It has been observed that testing cannot demonstrate the absence of errors. Hence verification should be attempted. We have also observed that the verification process has its own risks. The most important risk is that an assertion verification attempt may end inconclusively because of the failure to determine the consistency of constraints or the truth of a theorem. As already noted, this does not necessarily signify the falsity of the assertion, just that the verification attempt ended inconclusively. Another important risk is that the verification may be successful but rely implicitly upon false assumptions about the semantics of language constructs. As an example of this, we saw that symbolic executors generally make incorrect simplifying assumptions about the functioning of floating point hardware. As a result even a complete formal verification of program correctness may not completely rule out the possibility of an execution-time error. Hence it seems that both testing and verification should be considered techniques for raising the confidence of project personnel in the software product. Each is capable of bolstering confidence in its own way, and neither should be employed to the exclusion of the other. Comprehensive, integrated systems of tools for supporting both capabilities must be constructed and made easy to use.

3. NEED FOR SOFTWARE ENVIRONMENTS TO FURTHER THE DEVELOPMENT AND DEPLOYMENT OF TESTING AND EVALUATION TOOLS

The previous sections have made clear that we are currently at a very rudimentary stage in the development of the needed overall view of how existing testing, analysis, verification and documentation aids should be merged together into a powerful unified system capable of continuous and effective support for the software testing and evaluation process. It seems clear that a long period of broadly based experimentation with a wide variety of tools and approaches is necessary in order for us to reach the point at which the most effective tools can be identified and effectively integrated with one another in support of agreed to testing and evaluation objectives. It seems, further, that the best way to facilitate such experimentation is to contrive a large and flexible framework in which tools can be installed, configured, measured, evaluated, reconfigured, and reevaluated.

The previous sections have shown that different tool capabilities can complement each other rather effectively in support of these various functional objectives. Therefore it seems clear that the suggested framework should be organized in such a way as to facilitate coordination and cooperation among different tools. In addition, because there is inadequate experience with specific individual tools, it seems that such a framework should be built out of smaller, modular pieces of tool functionality which facilitates the process of creating new tool capabilities and altering the functionality of existing capabilities.

An obvious strategy for having tools complement each other in this way is to arrange for them to all access and update a single repository containing the information needed in order to support desired objectives. This repository could also serve as the point of collection for the statistics and data needed to help compare and evaluate the various tools and approaches. This suggests that the most important need now is for a careful study of what bodies of information are needed in order to carry out effective testing, evaluation, verification, and documentation activities. We should be thinking about what is needed in order to create information utilities aimed at the creation of data bases of information that are adequate to support the critical objectives of testing, analysis, verification and documentation. Certainly tools and tool fragments are indispensable to the creation and maintenance of such databases, but by focussing on the needed data items and files we will be better able to focus the process of comparing and evaluating tools. By creating a database-centered body of tools we will also be anticipating the time when we will be able to confidently create the sort of information utility which will be effective support for the needed testing and evaluation activities in the future.

A flexible set of smaller tool fragments built around a central database of shared information, and accessed by means of a friendly user interface language is called a software environment (see |Oste 81|). Thus it seems that the central locus of testing and evaluation tool development and evaluation should be the design and construction of software environments which emphasize the integration of testing, analysis, verification and documentation tools and fragments.

There are a number of research activities currently in progress aimed at the creation of integrated sets of tools for the support of software development. Some examples of such environments are Mentor [Donz 80], Interlisp [Teit 81], The Cornell Program Synthesizer [TeRe 81], and Toolpack [Oste 82, Oste 82a].

Of these research activities, it seems that the one which is closest to the above described orientation and objectives is the Toolpack activity. This project aims to build a sequence of increasingly ambitious prototype environments for the construction, testing, documentation and maintenance of Fortran programs. The tool capabilities offered in the early releases will include a comprehensive dynamic test probe insertion tool and run time monitoring system, called Newton; a syntax analyzer; a static semantic analyzer; a flowgraph consructor; a callgraph constructor; and a data flow analyzer. These tool capabilities are built out of a more or less standard set of smaller tool fragments, all of which are imbedded in an integrating framework, called the Integrated System of Tools (IST). The tools are to be invoked and controlled through a user interface language which attempts to focus the user's attention upon the manipulation and maintenance of a central system of files of information about the programs being created, analyzed, tested and maintained. This file system is capable of holding an elaborate structure of the source versions of the user's programs, as well as such derived versions as the parse trees, symbol tables, flowgraphs, and parameter list descriptions.

The Toolpack project intends to make this set of tools and the entire IST system freely available. Because a great deal of effort has been put into making it easy to use and powerful, it is hoped that a broad base of experience with this system will be obtained. It is hoped that this experience will then form the basis for better understanding of the data items and tool capabilities that users will most profit from having at their disposal. It is expected that the Toolpack design and IST architecture will be sufficiently flexible to allow for the alteration of IST into a system offering those desirable capabilities not already present.

Specifically, it is expected that Toolpack and the IST will be able to support the implementation of the sorts of integrated testing and documentation and verification activities suggested in Section 2 of this paper. Current plans for IST do not include a symbolic execution capability, or a constraint solver. Further, present plans do not call for the creation of the integrated capabilities described in section 2. The architecture of IST, however, should make it rather straightforward to incorporate these new capabilities and effectively integrate them.

For example, the incorporation of a symbolic executor should be very much facilitated by the IST architecture. The symbolic execution capability rests importantly upon the presence of parse tree and symbol table representations of the program. These are already supplied by existing IST tool fragments. In symbolic execution, the symbolic values that are evolved as the values of variables are considered to be bound as values of the variables. The IST also includes an attribute table, which is considered to be related to the symbol table, but is stored separately from it. The attribute table contains a large amount of semantic information about each variable. The symbolic value of a variable could well be considered to be yet another entry in the attribute table's packet of information for that variable. Thus it appears that the creation of a symbolic execution capability and coordination of it with other analysis, testing, evaluation and documentation capabilities is facilitated by an architectural decision to construct all of these capabilities out of smaller tool fragments.

The centering of these tool fragments around a data repository which is structured like the IST file system is also very helpful for the implementation of a symbolic execution system. In IST, the file system allows for the storage and easy management of various versions of a program's source code, derived images (symbol table, parse tree, etc.), input data and test output. It would not be difficult to also store the different paths which are of interest to someone desiring to perform symbolic execution of a program. The symbolic execution of a given program using a given path specification will result in a set of symbolic values for the program's variables. These sets of values could be considered to be different attribute tables corresponding to the fixed symbol table for the program which had been previously generated by the standard parser tool fragment in IST.

Thus, it is not hard to imagine the incorporation of symbolic execution within a database-centered system of tools which also contains a powerful dynamic testing capability and a comprehensive static analysis capability. This would enable the integration of these capabilities along the lines suggested in section 2. For example, the use of static analyzers and symbolic executors to remove dynamic testing probes would require only the alteration of a small number of existing tool fragments or the creation of a small number of new ones. The results of mutation testing could be stored in the central file system and coordinated there with related data objects. In particular, the outcomes of various structural testing regimens could also be stored in the central repository and might prove to be useful in fashioning new test cases designed to eliminate additional program mutants.

Finally it is important to observe that the sorts of environments which have been discussed all support testing and evaluation of program code, although it is generally agreed that the most significant and costly errors are those which are committed at the stages preceding the coding phase. Thus it is important to note that the principles of database-centering and use of small tool fragments are equally applicable to the design and architecture of environments capable of support of these earlier phases of software development as well. In particular, it is important to recall that Howden's concept of Functional Testing seems to be best thought of as the process of capturing the orderly design process as a sequence of design refinements described as sets of functions. There is surely no reason why this sequence of function refinements could not be captured and stored as part of large structured file system such as has just been described. Howden's scheme goes on to imply that these functions must all be tested carefully using guidelines which he supplies. This testing process could be managed most effectively with such a structured file system. Test executions corresponding to each of the functional refinements could be created as specific program instrumentations and associated input data sets. The outputs from these input sets could be caprured and stored as well. Thus, certain elements of the design of a program could profitably be captured in a central database, where they might be coordinated with associated elements of code and testing results. This indicates how a software production environment can begin to span the design process as well as coding, testing and documentation. There appears to be little reason why requirements specifications might not be incorporated profitably as well.

REFERENCES

[Balz69] R. M. Balzer, "EXDAMS: Extendable Debugging and Monitoring System," Proc. AFIPS 1969 Spring Joint Computer Conference 34 AFIPS Press, Montvale, N. J.

[Chea78] T. E. Cheatham, Jr. and D. Washington, "Program Loop Analysis by Solving First Order Recurrence Relations," Harvard Univ. Center for Research in Computing Technology, TR-13-78.

[Clar76] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, SE-2 pp. 215-222 (Sept. 1976).

[DeMi78] R. A. DeMillo, R. J. Lipton and F. G. Sayward, "Program Mutation: A New Approach to Program Testing," Infotech State of the Art Report on Software Testing 2, pp. 107-128 (Sept. 1978).

[Donz80] V. Donzequ-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming Environments Based on Structured Editors: The Mentor Experience," INRIA Research Report No. 26, INRIA, Rocquencourt, France, 1980.

[Fair75] R. E. Fairley, "An Experimental Program Testing Facility," Proc. FIrst National Conf. on Software Eng., IEEE Cat. #75CH0992-8C pp. 47-55 (1975).

[Floy67] R. W. Floyd, "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science 19 J. T. Schwartz (ed.) Amer. Math. Soc. Providence, R.I. pp. 19-32 (1967).

[Fosd76] L. D. Fosdick and L. J. Osterweil, "Data Flow Analysis in Software Reliability," ACM Computing Surveys 8 pp. 305-330 (Sept. 1976).

[Gris70] R. Grishman, "The Debugging System AIDS," AFIPS 1970 Spring Joint Computer Conf., 36 AFIPS Press, Montvale, N. J. pp. 59-64.

[Hant76] S. L. Hantler and J. C. King, "An Introduction to Proving the Correctness of Programs," ACM Computing Surveys 8 pp. 331-354 (Sept. 1976).

[Hech75] M. L. Hecht and J. D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," SIAM J. Computing 4 pp. 519-532 (Dec. 1975).

[Howd78] W. E. Howden, "DISSECT - A Symbolic Evaluation and Program Testing System," IEEE Trans. on Software Eng., SE-4 pp. 70-73 (Jan. 1978)

[Howd80] W. E. Howden, "Functional Program Testing," IEEE Trans on Software Eng., SE-6, pp. 162-169, (March 1980)

[King76] J. C. King, "Symbolic Execution and Program Testing," CACM 19 pp. 385-394 (July 1976).

[Lond75] R. L. London, "A View of Program Verification," 1975 International Conf. on Reliable Software, IEEE Cat. #75-CH0940-7CSR pp. 534-545 (1975).

[Mehl81] E. Mehlschau, "Softool 80, A Methodology and Integrated Collection of Tools for Software Management, Development, and Maintenance," Proc. Conf. on the Computing Environment for Math. Software, JPL Publication 81-67, Jet Propulsion Lab., Pasadena, Calif., pp. 20-21, July 15, 1981.

[Mill74] E. F. Miller, Jr., "RXVP, Fortran Automated Verification System," Program Validation Project, General Research Corp., Santa Barbara, Calif. (Oct. 1974).

[Oste76] L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation, Error Detection, and Documentation System for FORTRAN Programs," Software - Practice and Experience 6 pp. 473-486 (Sept. 1976).

[Oste81] L. J. Osterweil, "Software Environment Research Directions for the Next Five Years," Computer 14 pp. 35-43. (April 1981).

[Oste82] L. J. Osterweil, "Toolpack--An Experimental Software Development Environment Research Project," Proceedings Sixth International Conference on Software Engineering, IEEE Conference Proceedings, Tokyo, September, 1982.

[Oste82a] L. J. Osterweil, "The Toolpack Mathematical Software Development Environment," Department of Computer Science, University of Colorado at Boulder Technical Report #CU-CS-226-82, July 21, 1982.

[Rama75] C. V. Ramamoorthy and S.B. F. Ho, "Testing Large Software With Automated Software Evaluation Systems," IEEE Transactions on Software Engineering SE-1 pp. 46-58 (March 1975).

[Rich81] D. J. Richardson and L. A. Clarke, "A Partition Analysis Method to Increase Program Reliability," Fifth International Conf. on Software Eng., pp. 244-253, March 1981.

[Stuc74] L. G. Stucki and G. L. Foshee, "New Assertion Concepts in Self-Metric Software," Proceedings 1975 International Conference on Reliable Software, IEEE Cat. #75-CH0940-7CSR pp. 59-71.

[Tayl80] R. N. Taylor and L. J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis," IEEE Trans. on Software Eng. SE-6, pp. 265-278, (May 1980).

[TeRe81] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." Communications of the ACM 24 (Sept. 1981) 563-573.

[Teit81] W. Teitelman and L. Masinter, "The Interlisp Programming Environment," Computer 14, pp. 25-33, IEEE Computer Society, Los Alamitos, Calif. (April 1981).

[Weyu80] E. J. Weyuker and T. J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Trans. on Software Eng., SE-6, pp. 236-246, (May 1980).

[Whit80] L. J. White and E. I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Trans. on Software Eng. SE6, pp. 247-257, (May 1980).

ABOUT THE AUTHOR

Prof. Leon Osterweil is currently Chairman of the Department of Computer Science, University of Colorado at Boulder, where he holds the rank of Professor. He is currently also serving as Technical Co-Chairman of the Toolpack project, a cooperative research activity whose goal is the distribution of an advanced software development environment, and which currently involves researchers at seven institutions. Prof. Osterweil received a Bachelor's degree in Mathematics from Princeton University, and Masters and Doctoral degrees in Mathematics from the University of Maryland. He also spent a year on leave of absence working as an engineer and manager at Boeing Computer Services Company. His research interests center on the design, development, and application of software tools and environments. Prof. Osterweil has lectured and taught on these and related topics extensively in this country and abroad.

ASSESSING THE SOFTWARE PRODUCT QUALITIES OF CORRECTNESS AND RELIABILITY

Victor R. Basili
Department of Computer Science
University of Maryland

ABSTRACT

Quality assessment with regard to reliability and correctness needs to be performed across the entire software life cycle. In this paper we discuss what can be done at various phases in the software development life cycle to provide the developer and customer with greater confidence in the quality of the product. Several technologies currently available for use in assessing reliability and correctness are discussed; these include error analysis, testing strategies, reading and review techniques, reliability models, and product metrics. Finally a software development methodology which includes quality assessment is proposed and some aspects of the methodology are described with respect to its use of measurement.

INTRODUCTION

In attempting to assess the quality of a software product, we are looking for some objective or subjective statement that will permit us to know whether the product satisfies certain conditions imposed upon the product. These qualities may include the product's reliability, correctness, modifiability, maintainability, readability, its adherence to the requirements laid out for it, its ease of use, etc. In this paper we will deal mostly with the qualities associated with reliability and correctness , i.e. our ability to assess how well the product works and adheres to the requirements.

We can examine these qualities from the point of view of the user or the developer. In this paper we will try to do both. Ideally the user would like to evaluate the system as a 'black box', knowing nothing about the internals of the process. However the developer should view the system as a 'white box' gathering whatever information is needed to aid in assessing these qualities. Aside from the assessment of the overall reliability and correctness of the system, the developer would like to learn of problems in real time to make improvements during the development of the system. The developer is interested in the quality of each piece of the system at each stage of development. He would also like to understand what approaches have led to quality so that the organization can improve its quality in future developments.

To put things in perspective, we should consider a working definition of the life cycle. Rather than starting with a process model of the life cycle, we will offer a product oriented approach. The typical life cycle in software development consists of several documents. These include the requirements, specification, design, code and test document. The process model by which one generates these documents for a complete system varies. The classic life cycle model is a sequential process where each document is created before going on to the next. In this case the above documents correspond to phases in the life cycle. The requirements phase is where the requirements document is generated and analyzed. It represents the user's view of the system and is meant to define what is needed, without describing how it should be achieved. The specification phase is where the requirements are formalized from the developer's point of view. Again this document represents a statement of the problem or subproblems, rather than a solution. The design phase is where the specification is turned into an abstract solution. The coding phase is where the design is implemented in a programming language executable on a computer. This document consists of the commented source code.

Following each of these phases is typically a testing phase in which the product is executed relative to some specific input data. The test document should consist of a test plan, i.e. an approach to how the system should be tested and how test cases should be generated, and a recording of all test cases and results. Testing is divided into unit testing, integration testing, system testing and acceptance testing. Unit testing involves the execution of individual modules or subprograms of the system. Integration testing checks the ability of the individual pieces to fit together. System testing checks the effectiveness of the system to execute as a whole over some range of inputs. Acceptance testing is the final formal test that the product satisfies the requirements and with the input tests ideally generated by the user.

There are many variations of the phases of the life cycle stated above as well as many different process models that can be used for development. One such model which merits special attention is the iterative enhancement approach where the product is developed incrementally by passing through each of the various phases of the life cycle as versions of the product are developed. Each version contains more and more functional capability. In this approach the documents are also developed incrementally.

The ability to assess reliability from the user's point of view is dependent on the user's ability to generate a 'good' set of test cases for acceptance testing and ability to make use of a procedure for predicting the overall reliability and correctness of the product based upon that testing. This problem sounds simpler than it is. Developing a good representative set of test cases requires a great deal of insight into the requirements: what tests are representative of the requirements and future system use? The procedure for prediction is highly dependent upon those criteria. If the testing process is biased in any way, not only is the quality assessment biased, but so is the prediction of behavior.

Besides the problems of assessing the final reliability and correctness of the system, the developer has other concerns. The whole testing process is typically too late in the life cycle to help with the generation of a quality product. The developer needs to measure reliability and correctness incrementally in real time at the earliest stages of development to learn if the development is going well, to improve and refine the development process and to record the experience for future developments.

There are several technologies currently available for both the user and the developer to aid in the assessment of reliability and correctness: error and fault analysis, testing strategies, reading and review techniques, reliability models and product metrics. In the next section we will present an overview of some of these techniques and in further sections we will show how some of the various approaches can aid in quality assessment. Finally, we will propose some ideas for a comprehensive development methodology which will provide for quality assessment throughout the software development life cycle that makes use of all of these techniques.

TECHNOLOGIES

Clearly the ideas of quality assessment are closely tied to our understanding of the number and types of errors we make in developing software. The more we understand about causes of errors, when the errors enter the system, the costs of isolating and fixing errors, as well as any number of other error classification schemes, the more we can improve the methods for developing software, evaluate the quality of the product and generate effective approaches for discovering errors. Classifying errors helps the developer better understand what pieces of the system are of lower quality and where to focus the effort. This information can be used to focus integration and system tests. It also provides the user with some insights into what areas require more extensive testing at acceptance.

The approaches to evaluating software typically fall into two classes: reading and testing. Reading may involve review of any of the various documents associated with the software product (requirements, specification, design, code and test documents) in isolation for correctness or in pairs for consistency. Obviously reading requires that some form of document exist, but this process can take place at the earliest phase of the software development. Included in this category are individual readings of some life cycle document, any of a number of correctness proof techniques, and formal presentations of the information in walk-throughs or design reviews. The monitoring of these activities can generate valuable information for error analysis and reliability estimation at very early stages in the life cycle. The data collected can also be used by management and quality assurance people to modify practices if they have been ineffective and focus and refine the activities.

Approaches to testing include functional and structural testing. In each case tests must be generated in which the tester develops input for the executing program and has some knowledge of the expected output. Functional testing assumes the test input is made up to check the functionality of the product. Typically these test's are made from the requirements or specification document. The goal of structural testing is to execute as many statements or branches or paths of the program as possible. The testing approach can be guided by knowledge of the error history for a project. The testing process requires that code has been written and can be run on a machine so this process cannot be performed until after the coding phase has begun.

Reliability models are used to evaluate the product during testing. Rather than looking for errors or faults, reliability models record failures of the product. The most common models are stochastic models that record either times between failures or failure counts. These models attempt to generate a function which will allow the user to estimate the time of the next failure as well as measure the amount of testing time needed to reach some specified goal of mean time to failure or reliability. Reliability models are based upon the assumptions that the testing of the system is random and representative and that each error has equal probability of occurring during the testing. It is not probable that the same model can be used across all the testing phases. Clearly the more we know about error analysis and the effect of different testing schemes, the more effective use we can make of reliability models and the better we can interpret their results.

Product metrics can provide secondary level information about the reliability and correctness of the product from a quality assurance point of view. If we understand the relationship between errors and a variety of product metrics such as size, software science metrics [Halstead], numoer cisions [McCabe], number of data references within a program unit (span) [Elshoff], number of interconnections between program units [Henry & Kafura], etc., we can predict with some degree of confidence the error-proneness of the system. However, the relationship between many of these metrics and error-proneness has not yet been fully established.

ERROR ANALYSIS

The assessment of the reliability or correctness of a system is based upon the number and types of errors committed in its development. It is therefore crucial that error data be collected during the entire development of any system. Actually, we are not analyzing errors but faults. It is worth defining some of the terms for the purpose of clarity. According to [IEEE], an error is a discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. It is a human action which results in software containing a fault. A fault is an accidental condition that causes a functional unit to fail to perform its required function; a manifestation of an error in the software. In each of the above definitions we can assume that software includes all the documents, e.g. code, design, requirements.

Software developments vary with respect to many variables,e.g. application, organization, techniques, experience. Fault distributions can provide a signature of the project with regard to each of these factors. Knowing the fault signature for a class of problems can provide input into correctness and reliability evaluation of the product as well as information about the effectiveness of various methods and tools for software development. There are various types of fault distributions that can be studied: total number of faults, faults by component of the system, clerical vs. non-clerical faults, faults per week, faults by source of fault (requirements, design, etc.), faults per detection and correction technique, faults of omission vs. commission, faults per line of code, faults per difficulty of detection, etc.

These distributions can provide information about the effectiveness of various phases or documents in the life cycle, e.g. what document was the source of the most errors. This provides information on the correctness and reliability of software documents. For example, in an error analysis study [Basili & Weiss] on the A7 flight software, a categorization of errors in the requirements documents was defined as: clerical, ambiguity, omission, inconsistency, and incorrect fact. This scheme provided the developers with feedback on the types of problems involved in the requirements document which they used to improve their method for analyzing the document and provided them feedback on the effectiveness of their requirements methodology. This type of analysis also provides the user information on what types of failures to check for during acceptance testing as well as an assessment of the correctness and reliability of the requirements document.

Error distributions can also provide information on the types of errors being made. For example, using the omission vs. commission categorization scheme of errors , an error analysis [Basili & Perricone] in the Software Engineering Laboratory at NASA Goddard Space Flight Center discovered that a large percentage of the errors in a particular piece of software were errors of omission (35%). The results of this study for the user emphasize the importance of using a good representative functional testing scheme for acceptance testing since structural testing would not expose 35% percent of the errors. The developer learned that it was important for the readers to have a copy of the specification available while reviewing the design or code since some functionality might be missing from the design or code.

EVALUATION TECHNIQUES

Testing is currently the major source

of quality assurance with regard to reliability and correctness. There are several approaches to testing, top-down vs. bottom-up and structural vs. functional. Top-down testing assumes the top level components of the system are tested first with dummy routines or stubs used to simulate the bottom level functions. This approach allows the test data to be the actual data for the full system. Bottom-up testing involves the checking out of each of the lower level functions before they are integrated together. This approach assumes the existence of test drivers to simulate the input domain.

In structural testing, tests are based upon the structure of the program using some set of conditions, such as (1) test each statement at least once and (2) test each binary decision both ways. There are other criteria that can be used but these are the most common and practical ones. The approach allows for measurement of the level of success, one can check the percent coverage of statements and binary decisions. One can also check coverage over a specific class of statements, e.g. I/O statements, assignment statements.

There are many different types of functional or requirements testing approaches. We will define one approach for the purpose of later discussion. Equivalence partitioning [Hetzel], [Howden] involves identifying equivalence classes of input conditions. Each condition is then partitioned into two or more groups, valid equivalence classes representing valid inputs and invalid equivalence classes representing erroneous input values. Once the conditions are identified and tabulated, test cases are identified to cover conditions. The goal is to write a new test case covering as many of the uncovered valid equivalence classes as possible and one new test case for each invalid equivalence class.

During unit testing, structural testing can be an effective approach for making sure all the code is accessible. However some form of functional testing is still essential to improve the chances that the subfunction of the individual unit is correct. Clearly functional testing becomes all the more important during integration, system and acceptance testing. As pointed out earlier, structural testing has the drawback that it does not catch errors of omission. The major problem with functional testing is that creating the conditions that need to be checked is a difficult heuristic process and checking for all possible conditions can be almost impossible on a very large system.

In most organizations, some type of problem report is generated to keep track of faults and fixes. These forms could be modified to provide the developer with valuable information for error analysis.

Reading is a completely human based activity. It includes reading of any of the life cycle documents alone or in combination. Documents can be read privately by the individual who wrote them or by another person. Walk-throughs can be performed in which the individual talks a group through the document, or they can be done independent of the developer, by an external group for independent quality control [Fagan]. Typically forms are filled out at the walk-through or design inspection which capture the problems found in the document. This data is useful for determining the error distributions discussed above.

It is worth a minor discussion of the differences in activities performed when doing reading and testing. For reading, test cases do not need to be generated. In testing the error detection and isolation problems are separate, i.e. if for a specific input, a failure is generated, the place where the error took place must still be isolated. In reading detection and isolation are one activity. In testing the error can only be found after the code exists, in reading it may be found any time after entry into a document. In testing, program coverage is test based, if a test has not been devised to cover a condition, the condition will not have been checked out. In reading, assuming the entire document has been reviewed, coverage can be total. Reading does require intermediate functions or requirements against which the design and code are checked.

It has not been adequately demonstrated which of the approaches discussed above have the greatest effect in practice. One study [Myers], compared testing and code reading for effectiveness in discovering bugs. This study is important in that it begins to address the issue of effectiveness of various techniques, whether they should be used together or one as a check of the other. Some work done at the University of Maryland [Hwang] has exposed another problem: even if the proper set of tests have been developed, the tester may not be able to recognize that an error has occurred. A study is currently underway which will partly duplicate the study of Myers as well as shed some further light on the effectiveness of finding various classes of errors using structural and functional testing and code reading.

A quantifiable measure of quality that has become popular in software engineering practices is software reliability. It can be defined as follows: Let E be a class of errors, defined arbitrarily and T be a measure of relevant time, the units of which are dictated by the application at hand. Then the reliability of the software package with respect to the class of errors E and with respect to the metric T, is the probability that no error of the class occurs during the execution of the program for a specified period of relevant time [Goel].

A number of analytical approaches have been developed to address the problems of software reliability assessment [Goel],[Musa]. These approaches are mostly based upon the error history of the software. They may be divided into time-dependent and time-independent approaches. The time-dependent approaches is based on either times between failures or on failure counts in specified intervals. The time independent approach uses either error seeding methods or input domain analysis.

In the time-dependent approach, the times between exposure of errors or the number of errors observed in a sequence of test time intervals are used to estimate the shape of the hypothesized failure (hazard) rate function. From the estimated failure rate functions, one can estimate the number of errors remaining in the software, mean-time-to-failure (MTTF) and software reliability.

In the error seeding approach, a known number of errors is seeded (planted) in the program. After testing, the numbers of exposed seeded errors and indigenous errors are counted. Using combinatorics and maximum likelihood estimation, one can then estimate the number of indigenous errors in the program and also the reliability of the program.

In the input domain based models, the procedure is to generate a set of test cases from an input (operational) distribution. The difficulty of estimating the input distribution is overcome by partitioning the input domain into a set of equivalence classes. An equivalence class is usually associated with a program or logic path. The reliability measure is then calculated from the observed failures after symbolically or physically executing the generated test cases.

A common problem with the use of these models seems to be the lack of a clear understanding of the inherent strengths and weeknesses of such models. Furthermore in many cases, the underlying assumptions and outputs of the models are not fully appreciated by the users. , It is also true that not all models are applicable to all testing environments [Goel, Basili & Valdes]. There has been lack of controlled study and validation of these models.

One idea that implicitly combines testing and reliability is the clean room [Dyer & Mills]. Here the programmer must use a variety of reading techniques since the testing is only done by an independent organization. When used with top down development, this approach allows portions of the system to be tested, using random, functional testing, and the test results to be analyzed by a reliability model. The developer then gets the test results and some quality metric on the reliability of the system at various points in the development. An experiment being run at the University of Maryland shows that the developers become much more conscious of quality than might otherwise be true.

Using each of the techniques discussed, data can be collected that involves all of the error distributions listed. Coverage metrics can be obtained. Although structural testing coverage metrics may not capture the quality of the testing because they do not record errors of omission, they might be useful when used in conjunction with a functional testing plan. In this way, knowing the number of errors of omission we may be able to extrapolate and gain some information on how well the system is tested.

The results of testing can be used as inputs to an appropriate reliability model to obtain estimates of reliability mean time to failure.

LIFE CYCLE METHODOLOGY

Quality assessment and reliability evaluation need to be done across the entire life cycle. In this section we will outline a proposal for a quality assessment approach to software development which involves the application of several of the technologies discussed above. In what follows we will discuss a few of the phases in the quality assurance measurement context, emphasizing reliability and correctness.

Requirements

The earliest time available to assess quality and reliability is during the requirements phase. We can examine the requirements document in an effort to assess its quality and to set up the needed predictors for quality of the final developed product.

Fault distributions can be used to evaluate the reliability and correctness of the requirements document itself. The developer can keep track of (1) the types of errors, (2) types of changes, and (3) the errors by section of the document. Distribution (1) can be used to determine if there are problems with ambiguity,inconsistency, omission or incorrect facts in the requirements document. This information is useful in focusing testing and review procedures for the document itself. Distribution (2) provides information about where changes are occurring and might indicate the need for special testing. Errors can be localized with respect to functionality using distribution (3). There are several other fault distributions which specifically relate to errors associated with a requirements document listed in [Basili & Weiss].

There are metrics than can be used in assessing the existing document. Several of these are defined in [McCall, Richards, & Walters] and include (1) a completeness checklist, e.g. unambiguous references?, all functions defined?, (2) an accuracy checklist, e.g. error analysis performed and budgeted to module?, a definitive statement of requirement for accuracy of inputs, outputs, processing and constraints?,. These metrics can be used to evaluate requirements document and point out potential areas for error.

In order to set up the necessary framework for evaluating the goodness of the final tests and the resulting quality evaluation based upon those tests we can design a test plan based upon the requirements document that can be used to quantify later results. One criteria is to know how well the requirements have been tested. Let us start with a simple predictor. Assume we can develop a metric that indicates the percent of testing performed relative to some minimum set of tests. In the ideal, this would involve knowing the total number of tests (T) based upon some good strategy for functional testing and then using this number to normalize the set of tests actually used (t). Then t/T gives a measure of the thoroughness of testing. Assuming that the t tests are representative of the T tests let ts represent the number that were successful and tf represent the numbgr that failed, then ts/t is a measure of the reliability of the product, relative to the test set t and (tf*T)/t is a simple predictor of the total number of errors in the product.

In defining the metrics above two assumptions were made: (1) we can find a T that indicates the total number of tests needed to expose all the errors in the program (2) we can find a t representative of T. Let us first discuss assumption (1). This implies we can come up with a good functional test plan, e.g. equivalence partitioning, that could be used for the entire requirements document. We must come up with the set of all input conditions and create for each category the set of valid and invalid subclasses. The set of tests needed to satisfy the equivalence partitioning would then be an estimate of T. A simple alternative estimate of T without making up all the possible test cases might be 2*n where n represents the number of input conditions, assuming an average of 2 tests per condition. An even simpler estimate might be achieved, if we can assume that n could be approximated by the number of sentences in the requirements document.

Generating a representative t may be more complicated unless we can categorize the input/output conditions into large equivalence classes. A straightforward approach would then be to choose t randomly from the set T.

We can refine these metrics in two ways, (1) by assuming there are categories of errors, e1, e2, etc. based upon the severity of the errors or some other categorization scheme for errors, (2) by assuming that the requirements document has been partitioned into input domains, possibly based upon the importance of the product meeting certain requirements and then classifying the test cases according to these input domains.

Design

There are many things that can be measured during the design phase. These include fault distributions and counts, complexity metrics and traceability to the requirements or specification document.

Assuming design reading and design inspections, records can be kept of the faults during design and several fault distributions can be derived. Design faults can be categorized with respect to (1) total number of faults, (2) total number of faults/ line of design, (3) number of faults/system component, (4) faults of omission vs. commission, (5) faults caused by a misunderstanding of the interface vs. the design of a single component, (6) faults by when they entered the system. Distribution (1) and (2) can be used to evaluate the reliability of the system and the design approach. The data can be used in a reliability model to try to predict future errors. Distribution (3) can be used to assess the reliability of each component. Components with a higher percentage than the average might be reviewed for further errors, higher complexity and are candidates for redesign. Distribution (4) can be used with data distributions from previous projects to check if the project is consistent. If the distribution is substantially different from previous studies, is it because the development team is doing a better job or because there are problems in the review process, e.g. design is not being checked against the specification so errors of omission are not being found. Distribution (5) tends to pinpoint problems with the methodology. If there are too many interface errors then more work needs to be done in specifying and clarifying the interfaces. These last two distributions also help focus the testing activity and might generate modifications to the test document, e.g. a large number of interface errors might focus more attention on integration testing. Distribution (6) provides information on the weak documents in the life cycle. If many of the errors are requirements errors then, it may be worth going back and reanalyzing the requirements document.

There are many other distributions that can be collected during the design process that provide the developer with insights into how to improve the process, focus the techniques being used and what might need redoing. The user, if any of this information is available to him, can use the information to modify the acceptance test plan.

Complexity metrics can be used to track potential problem areas. To a large extent, what can be measured during design depends upon the design notation used. If a process design language is used, most of the metrics defined for code can be applied. These can be applied at each stage of the design to gather information at the current design point as well as examine progress (size changes) and analyze the effect of updates in the refinement of a design (complexity metrics).

The idea of a metric vector is being studied at the University of Maryland [Hutchens], in which a large number of metrics are collected on the existing product, e.g. the design document. These metrics are then tracked throughout the development, i.e. through design, code. test, maintenance. The metric vector can be used to characterize the current version of any document, indicate changes in the system, and act as a mechanism for checking the relationship between various metrics. Assuming the vector contains metrics that indicate such characteristics as the size, control complexity, data complexity within a component, data complexity across components, number of errors in a component, and number of changes to a component, the developer can at any point in time determine the "quality" of that component, and of the whole system. For quality assessment, bounds could be set for each of the metrics and when the bounds are exceeded, review or redesign of the component can take place. Examining the changes to a particular component can provide information about whether the implementation of a lower level function exceeded the predicted "complexity". This again could signal a flag to the quality assessment team permitting any number of actions. The third benefit of the metric vector is to allow the quality assurance group to derive the relationships for that particular project between the various metrics, e.g. when the control complexity is high there is a larger than average number of errors in the algorithm. This information can then be used to predict potential errors and error types.

To deal with traceability and aid in the testing process, a requirements/ design component vector can be developed [Valdes]. In this vector requirements (rows) are compared with the various system components (columns) and an entry is made in the matrix if the component implements any part of the requirement. Blank rows in the matrix indicate a missing requirement in the design. The greater the level of detail in the requirements ,the more information is available. The rows in the matrix should correspond to the input conditions derived from the equivalence partitioning process in the requirements phase. The matrix provides a view of the traceability of the requirements. A small number of items in the matrix indicates a functional design. The easier it is to draw the matrix, the simpler the design and the easier it is to trace the requirements to the design. The matrix also provides a relationship between functional and structural testing. Using component coverage in structural testing, test cases can be checked to see if all components that implement a particular function have been visited.

The approaches for the specification and coding phases are similar to the requirements and design phases respectively. The coding phase permits more measurement than the design phase since execution metrics can also be applied. During the various test phases, coverage metrics and reliability models can be used. Early study indicates the clean room approach may be an effective means of providing quality assessment during the coding phases.

SUMMARY

Quality assessment of software must be done across the entire life cycle. The assessment program requires measurement and data collection so the user and developer can gain the proper confidence and assurance that the system behaves the way it is expected to behave. This quality assessment program must be built into the software development methodology from the beginning.

Techniques available for assessment are error analysis, reading and testing, reliability models, and product metrics. These technologies need to mature, but they will only mature through use and experimentation in a variety of software development environments. There are still many open questions concerning the available technologies. What are the relationships between the various testing techniques and reading? Can coverage metrics be used in assessing functional testing? How and when can reliability models be used with confidence? What is the relationship between the various complexity metrics and software quality? It is important that effort be expended to answer these questions to assure the future quality assessment of software.

ACKNOWLEDGEMENTS

I would like to thank Professor Amrit Goel for his contributions to this paper. Work on this project was supported in part by NASA Goddard Space Flight Center grant NSG -5123 to the University of Maryland.

REFERENCES

[Basili & Perricone]
V. R. Basili and B. Perricone, Software Errors and Complexity: An Empirical Investigation, University of Maryland Technical Report TR-1195, August 1982.

[Basili & Weiss]
V. R. Basili and D. Weiss, Evaluation of a Software Requirements Document by Analysis of Change Data, Proceedings of the Fifth International Conference on Software Engineering, March 1981, pp.314-323.

[Dyer & Mills]
M. Dyer and H. Mills, Developing electronic Systems with Certifiable Reliability, Proceedings of the Conference on Electronic Systems Effectiveness and Life Cycle Costing, 1982, NATO Advanced Study Series, Springer-Verlag.

[Elshoff]
J. Elshoff, An Analysis of Some Commercial PL/1 Programs, IEEE Transactions on Software Engineering , June 1976.

[Fagan]
M.E. Fagan, Design an Code Inspections to Reduce Errors in Program Development, IBM Systems Journal , Vol.15, No 3, 1976.

[Goel]
A. L. Goel, Software Reliability Modeling and Estimation Techniques, RADC-TRxxx, February 1983.

[Goel, Basili & Valdes]
A. L. Goel, V. R. Basili, and P. Valdes, How and When to Use Software REliability Models, Sixth Annual Software Engineering Workshop, NASA Goddard Space Flight Center, 1982.

[Halstead]
M. Halstead, Elements of Software Science , Elsevier North-Holland, New York, 1977.

[Henry & Kafura]
Sallie Henry & D. Kafura, Software Structure Metrics Based on Information Flow, Transactions on Software Engineering , Vol. SE-7, September 1981, pp 510-518.

[Hetzel]
William C. Hetzel, An Experimental Analysis of Program Verification Methods, University of North Carolina at Chapel Hill, Ph.D. Thesis, 1976.

[Howden]
William E. Howden, Functional Program Testing, IEEE Transactions on Software Engineering , Vol. SE-6, No 2, March 1980, pp 162-169.

[Hutchens]
D. Hutchens, The Value of Objective Measurements in the Characterization of Software, University of Maryland, Ph. D. Thesis, 1983 (to appear).

[Hwang]
S. Hwang, An Empirical Study in Functional Testing, Structural Testing, and Code Reading/Inspections, University of Maryland, Scholarly Paper, December 1981.

[IEEE]
IEEE Standard Glossary of Software Engineering Terminology, IEEE STD-729-1982.

[McCabe]
T. J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering , Vol. SE-2, No. 4, Dec. 1976, pp 308-320.

[McCall, Richards & Walters]
J. McCall, P. Richards, G. Walters, ractors in Software Quality, Rome Air Development Center, RADC-TR-369, November 1977.

[Musa]
John Musa, A Theory of Software Reliability and Its Application IEEE Transactions on Software Engineering , Vol. SE-1, No.3, pp. 312-327.

[Myers]
G. J. Myers, A Controlled Experiment in Program Testing and Code Walkthrough/ Inspections, CACM , Vol.21, 1978, pp 760-768.

[Valdes]
P. Valdes, An Approach to Software Testing and Reliability Assessment, Syracuse University, Ph. D. Thesis, 1983 (to appear).

ABOUT THE AUTHOR

Victor R. Basili is Professor and Chairman of the Computer Science Department at the University of Maryland, College Park, Maryland. He has been involved in the design and development of several software projects, including the SIMPL family of programming languages. He has been measuring and evaluating software development in several places, including the Software Engineering Laboratory at NASA/Goddard Space Flight Center. Dr. Basili is the author of over 50 published papers on software development methodology and the quantitative analysis and evaluation of the software development process and product. He is a recipient of the Outstanding Paper Award from the IEEE Transactions on Software Engineering for the paper entitled "A Controlled Experiment Quantitatively Comparing Software Development Approaches" published in May 1981. He has consulted with several government agencies and industrial organizations, including IBM, GE, CSC, Naval Research Laboratory, Naval Surface Weapons Center, and NASA. He has been Program Chairman for several conferences, including the 6th International Conference on Software Engineering, and the First ACM SIGSOFT Sponsored Engineering Tymposium on Tools and Methodology Evaluation. He has served on several editorial boards, including the Journal of Systems and Software and the IEEE Transactions on Software Engineering. He is a member of the ACM and a member of the Executive Committee of the Technical Committee on Software Engineering, IEEE Computer Society.

IMPACT OF SOFTWARE TESTING ISSUES ON FUTURE SOFTWARE ENGINEERING ENVIRONMENTS

Leon G. Stucki, Ph.D.
Boeing Computer Services Company
Seattle, Washington

ABSTRACT

Software testing tools have been discussed for some time in the literature. Typically, two basic approaches have been followed:

Formal constructive - methodologies and tool systems to specify and build programs with certain "assured" Properties

Code analysis - tools designed to examine specific properties of existing programs

The formal approach while offering theoretically pleasing results is still viewed with skepticism by most involved with large systems. The code analysis tools also suffer serious inadequacies. One major problem is that they work on "existing code" which is harder to modify and often too late in the project life cycle to allow significant changes.

There are several efforts currently underway attempting to partially bridge this gap. The author will discuss one such project - Argus. Argus is an advanced software engineering environment being built on a micro-based workstation.

The Argus environment contains management, design, programming, and analysis tools. The presentation will include examples and observations describing the early use of this system. The notion of software testing is distributed throughout Argus and has heavily influenced the architectrue of this system.

Emphasis has been placed within the design component of Argus on capturing information which will be of considerable value in better understanding the system being specified, and in planning for subsequent testing activities. Capturing this information in the design phase is quite straight forward and considerably less expensive than attempting to create it after the code has been produced.

APPLICATION OF SOFTWARE METRICS DURING EARLY PROGRAM PHASES

Ralph C. San Antonio, Jr.
Dynamics Research Corporation
Wilmington, Massachusetts

Major Kenneth L. Jackson
U. S. Air Force Ballistic Missile Office
San Bernardino, California

ABSTRACT

This paper demonstrates the use of quantitative measures of software quality during early requirements and design phases. The approach is part of an overall methodology for improving software quality that includes systematic procedures for collecting and assessing data, and automated tools for measuring and analyzing software quality characteristics. Software metrics are combined with a requirements specification tool to compare and evaluate software requirements. Results based on a limited application of the approach are presented.

INTRODUCTION

Software quality is a matter of perspective. Developers want software that meets the functional and performance requirements, and is delivered on time and within cost. Users need software that is reliable, easy to use, and supportive of their mission objectives. Maintainers require software that is easy to fix, modify, and test. Software quality, therefore, comprises all software attributes and characteristics that are important to developers, users, and maintainers.

For major defense systems, users and maintainers rely on the developers (the program office and development contractors) for specifying software quality requirements, for monitoring and evaluating software quality, and for ensuring that the software complies with specifications.[1] Unfortunately, definitions of software quality that allow users and maintainers to express their requirements to developers are not well established. Similarly, developers lack quantitative techniques for specifying, measuring, and assessing software quality. This is especially true of early program phases where little research has gone into developing quantitative methods for recording and evaluating software requirements.[2] Yet, the early requirements phase offers the greatest opportunity to influence software quality and life-cycle cost.[2,3,4] Problems that go unnoticed until program testing are always expensive and difficult to fix. Moreover, adhering to poor quality specifications assures delivery of poor quality software.[4] To avoid this result, definitions of software quality that make it possible to discuss, specify, measure, and evaluate software quality during early program phases are required.

This paper discusses a comprehensive approach for improving software quality. The approach supports quantitative comparisons and evaluations of software during early requirements and design phases. It is part of an overall methodology for specifying, measuring, and assessing software quality throughout the software life-cycle. The paper identifies software quality concepts and related metrics that are fundamental to the approach. The paper further describes how the integration of software metrics with an automated specification development tool provides a means to assess software quality in early development phases, and to improve software quality during successive iterations of the requirements specifications. Results based on a limited application of this approach to Peacekeeper Missile Programs are presented to illustrate the benefits. Conclusions and recommendations for future application of the technology are presented.

OVERVIEW OF THE METHODOLOGY

The methodology for improving software quality is comprised of three elements: software quality characteristics stored as data in a project data base; systematic procedures for collecting and evaluating these data; and automated tools for measuring and analyzing software quality. The following paragraphs describe each of these elements, beginning with the project data base.

The Project Data Base

The project data base contains the information required to analyze software quality, including characteristics of the software, the software development process, and the system in which the software operates. Software characteristics are derived from various representations of the software, such as requirements specifications, design documents, and source code. Details of the software development process are obtained from documents such as software development plans, and programming standards and procedures manuals. System features are taken from system specifications and related design documentation. (Table 1 identifies a representative list of software documents from which the data base is constructed.)(5,6)

CATEGORY	DOCUMENT
ENGINEERING	SYSTEM SPECIFICATION FUNCTIONAL REQUIREMENTS SPECIFICATION INTERFACE REQUIREMENTS SPECIFICATION DETAIL DESIGN SPECIFICATION INTERFACE DESIGN SPECIFICATION DATA BASE DESIGN SPECIFICATION STANDARDS AND PROCEDURES MANUAL VERSION DESCRIPTION DOCUMENT
TEST	TEST PLAN TEST SPECIFICATION TEST PROCEDURE TEST REPORTS
MANAGEMENT	CONFIGURATION MANAGEMENT PLAN QUALITY ASSURANCE PLAN SOFTWARE DEVELOPMENT PLAN MAINTENANCE PLAN
OPERATION/SUPPORT	OPERATOR'S MANUAL USER'S MANUAL DIAGNOSTICS MANUAL PROGRAMMER'S MANUAL

TABLE 1. REPRESENTATIVE SOFTWARE DOCUMENT TYPES

The software quality model described by McCall and others (references 7 and 8) was the starting point for defining the data base parameters. This model was selected because it resulted from a thorough analysis of software quality concepts and terminology. In addition, the model identifies software characteristics important to developers, users, and maintainers, and establishes a framework for analyzing software quality. This framework was modified and extended, particularly at the lower levels. The resulting model consists of factors used to describe software quality; criteria, with associated metrics, used to quantify the factors; metric-elements, based on observable software characteristics, used to calculate the metrics; and primitive software measures, called data items, that support metric-element calculations. (The framework is depicted in Figure 1.)

Factors are management-oriented terms such as reliability, maintainability, flexibility, and useability that represent software qualities important to users, maintainers, and developers. Eleven factors are described in the model. (These factors are listed in Table 2.) Each factor is composed of one or more software-oriented terms called criteria.

RELIABILITY
MAINTAINABILITY
FLEXIBILITY
PORTABILITY
USEABILITY
REUSEABILITY

INTEROPERABILITY
TESTABILITY
EFFICIENCY
CORRECTNESS
INTEGRITY

TABLE 2. SOFTWARE QUALITY FACTORS

Criteria are software-oriented terms such as consistency, completeness, accuracy, and self-descriptiveness that relate software characteristics to the eleven factors. Twenty three criteria are identified in the model. (Criteria are listed in Table 3.) Some criteria support more than one factor. For example, consistency supports the factors reliability, maintainability, and correctness. Also, some of the criteria are further partitioned into subcriteria to refine the definitions. Consistency, for example, is subdivided into procedure consistency and data consistency. Each criterion and subcriterion is associated with a metric.

Metrics quantify software characteristics related to the criteria and subcriteria. They are comprised of lower-level measures called metric-elements.

TRACEABILITY
COMPLETENESS
CONSISTENCY
ACCURACY
ERROR TOLERANCE
SIMPLICITY
MODULARITY
GENERALITY
EXPANDABILITY
INSTRUMENTATION
SELF-DESCRIPTIVENESS
EXECUTION EFFICIENCY

STORAGE EFFICIENCY
ACCESS CONTROL
OPERABILITY
TRAINING
COMMUNICATIVENESS
SOFTWARE SYSTEM INDEPENDENCE
MACHINE INDEPENDENCE
COMMUNICATIONS COMMONALITY
DATA COMMONALITY
CONCISENESS
ACCESS AUDIT

TABLE 3. CRITERIA

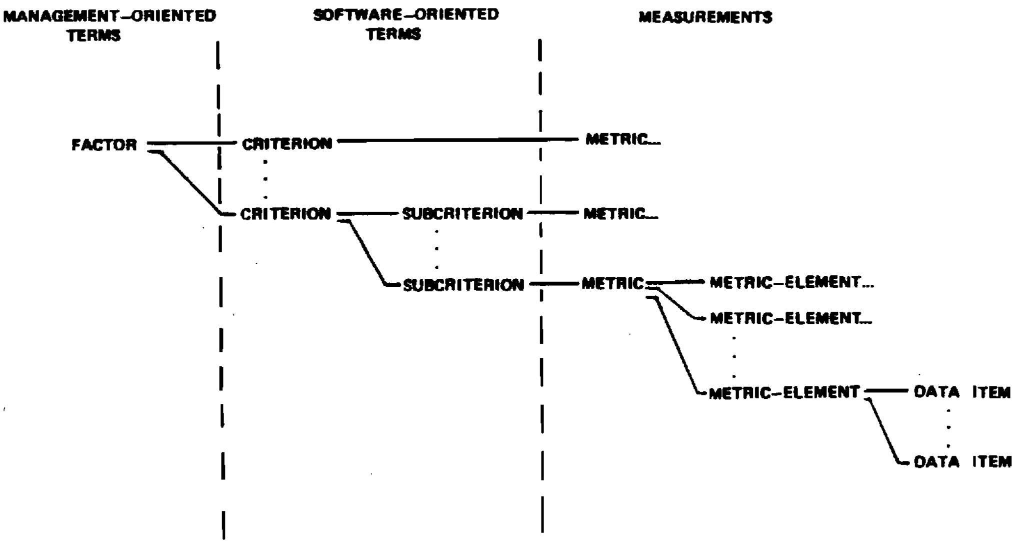


FIGURE 1. SOFTWARE QUALITY FRAMEWORK

Metric-elements are objective measures of software characteristics used to calculate metric values. (Table 4 lists metric-elements for the criterion completeness.) There are 153 metric-elements defined in the referenced model to support the metric calculations. Implied in the model, but not specified, is a lower-level unit of measurement required to support metric-element calculations and to complete the definition of the data base. These are called data items.

FUNCTIONS SATISFACTORILY DEFINED
DEFINED FUNCTIONS USED
REFERENCED FUNCTIONS DEFINED
DATA REFERENCES HAVING ORIGIN
DATA REFERENCES HAVING DESTINATION
CONDITIONS AT DECISION POINTS
ALTERNATIVE PROCESSING OF CONDITIONS
ALTERNATIVE CONDITION OPTIONS SET
PROCESSING FOLLOWS SET CONDITION OPTIONS
CALLING SEQUENCE PARAMETERS AGREE
PROBLEM REPORTS RESOLVED

TABLE 4. METRIC-ELEMENTS FOR COMPLETENESS[9]

Data items are primitive software measures used to calculate metric-elements. Examples include the number of functions, the number of branch paths, and the number of data references in a program. Data items are defined for all life-cycle phases. (Table 5 identifies requirements-phase data items for two of the metric-elements related to the criterion completeness.) To date, 374 data items have been identified to support the metric-elements described in the original model.[9]

METRIC ELEMENT	DATA ITEMS
DATA REFERENCES HAVING ORIGIN	DATA REFERENCE NOT HAVING ORIGIN
	NUMBER OF DATA REFERENCES NOT HAVING ORIGIN
	NUMBER OF DATA REFERENCES
ALTERNATIVE PROCESSING OF CONDITIONS	CONDITION OPTION WITHOUT PROCESSING
	NUMBER OF CONDITION OPTIONS WITHOUT PROCESSING
	NUMBER OF CONDITION OPTIONS

TABLE 5. DATA ITEM EXAMPLES[9]

The framework shown in Figure 1 establishes a basis for evaluating software quality throughout the software life-cycle. It identifies the data items that must be measured and included in the data base during each life-cycle phase. The steps required to collect, measure, and analyze these data are described in the procedures.

Systematic Procedures

The procedures define a systematic approach for collecting and analyzing data, and for evaluating software quality during each life-cycle phase. The procedures identify when measurements are made, how the information is collected, what analyses are accomplished, and how the results are reported.

While responsibilities may vary from project to project, the procedure remains the same. Typically, data collection coincides with the software development milestones and the delivery of software specifications and related documentation. Software quality characteristics are measured using automated tools for requirements, design, and coding information available in machine-readable form. When this information is not available in a machine-readable form the characteristics are measured manually using worksheets and checklists. Metric-element values are calculated using the resulting data. Statistical analyses of the metric-element values are used to compare modules against one another and to establish trends. Once a data base sufficient to assure statistical confidence has been established, metric-elements are combined into higher-level metrics. Thresholds are established to identify where additional resources should be applied to improve software quality. Throughout this procedure, the analyses are supported by the automated tools integral to the methodology.

Automated Tools

Wherever possible, automated tools are used to aid in the measurement and assessment process. The primary goal is to minimize the cost required to collect the data and calculate metric and metric-element values and statistics. Also, the tools assure consistent results from one application to the next. Presently, the effective development and use of measurement tools is hampered by the wide variation in host-computer environments and languages used to develop software for major defense systems; however, standard analysis tools can be developed and used to effectively support the assessments since the data base is defined by the software quality model.

APPLICATION TO PEACEKEEPER MISSILE PROGRAMS

The approach described above was applied to Peacekeeper Research and Development Flight Programs (R&D FP) to demonstrate the benefits of using software quality metrics. The principal goal was to identify techniques and procedures for integrating quantitative methods of measuring and assessing software quality into the Peacekeeper development community. The initial demonstration was restricted to the R&D Flight Program Computer Program Development Specification (CPDS) that resulted from the requirements phase. To simulate automating this phase of the process the specification was translated into a machine-readable form. For this application the Problem Statement Language (PSL)/Problem Statement Analyzer (PSA) developed at the University of Michigan was chosen.

PSL/PSA was developed to improve the process of analyzing and preparing software specifications.[10] It allows the user to represent a software specification in machine-readable form using PSL objects, relationships between objects, and object properties. PSA checks statements for consistency during creation of the PSL data base. PSA also provides reports that can be used by the analyst to evaluate the system description.[11] The objects and relationships used by PSL/PSA provide information in the following areas; system input/output flow, system structure, data structure, data derivation, system size and volume, system dynamics, system properties, and project management.[10]

PSL was also used to remove ambiguities at lower levels in the software quality framework. For example, without further guidance the determination of whether or not a function is "used" could vary between analysts. Table 6 provides a PSL definition of the data item defined function not used that eliminates this uncertainty.

DEFINED FUNCTION NOT USED

A PSL PROCESS IS USED IF IT IS RELATED TO AT LEAST ONE OTHER PSL OBJECT BY

INCEPTION CAUSES
TERMINATION CAUSES
TRIGGERS
TERMINATES
INTERRUPTS
UTILIZES

THOSE PSL OBJECTS THAT CAN INTERACT WITH PROCESSES WITH THE ABOVE RELATIONSHIPS ARE INPUTS. EVENTS, CONDITIONS AND PROCESSES.

TABLE 6. PSL DEFINITION OF DEFINED FUNCTION NOT USED

Defining data items in this manner also allowed the use of PSA to measure the PSL representation of the specification. Worksheets were used to record the measurements thus obtained; the results were stored in the project data base for later analysis and assessment.

Analysis of Data

The analysis was restricted to the metric-element level since the historical data base necessary to combine these elements into higher-level metrics is lacking. In addition, the analyses were based on a partial translation of the specification into PSL. This translation covered all aspects of the three major CPDS functions, Steering, Sequencing, and Control, except for the dynamics of the Steering and Control functions. While this affected the outcome, the results are typical of those obtained during the evolution of a specification. The following paragraphs illustrate the insight gained from this approach using metric-elements for the criterion completeness as examples.

Metric-elements are calculated by simply dividing the number of occurences that do not satisfy a condition or characteristic by the total number of occurences for the set considered. For example, the metric-element data references having origin is calculated at the system level by dividing the number of data references not having an origin within the system by the total number of data references within the system. Hence, a value of zero is highest quality and one is lowest quality.

	DTA-RFRNCS-HVNG-ORGN	DTA-RFRNCS-HVNG-DSTNTN	ALTRNTV-CNDTN-OPTNS-ST	ALTRNTV-PRCSNG-AT-CNDTNS	PRCSNG-FLWS-ST-CNDTN-OPTNS	FNCTNS-STSFCTRLY-DFND	DFND-FNCTNS-USD	RFRNCD-FNCTNS-DFND
SYSTEM	0.54	0.06	0.53	0.58	0.17	0.01	0.48	0.00
MAJOR FUNCTION								
SEQUENCING	0.55	0.00	0.52	0.54	0.09	0.00	0.00	0.00
STEERING	0.55	0.01	0.50	1.00	1.00	0.00	0.00	0.00
CONTROL	0.54	0.07	0.60	0.90	1.00	0.01	0.77	0.00
FLIGHT PHASE								
TERMINAL COUNTDOWN	0.50	0.00	0.50	0.50	0.00	0.00	0.00	0.00
PHASE ZERO	0.68	0.00	0.50	0.50	0.00	0.00	0.25	0.00
PHASE ONE	0.56	0.06	0.50	0.71	0.43	0.00	0.47	0.00
PHASE TWO	0.58	0.08	0.50	0.50	0.18	0.00	0.50	0.00
PHASE THREE	0.53	0.08	0.55	0.55	0.00	0.00	0.58	0.00
PHASE FOUR	0.49	0.04	0.50	0.58	0.27	0.04	0.41	0.00

NOTES:

0.0 · HIGHEST QUALITY

1.0 · LOWEST QUALITY

(RESULTS BASED ON PARTIAL TRANSLATION OF SPECIFICATION)

TABLE 7. METRIC-ELEMENT VALUES FOR COMPLETENESS

Calculations for the metric-elements contained in the criterion completeness are shown in Table 7. Values are shown for the system, for each major function, and for each flight phase. Metric-elements were grouped in this manner to gain added visibility into the specification. In addition, metric values were calculated for subfunctions within Sequencing, Steering, and Control. Figure 2 shows values of the metric-element data references having origin for subfunctions within Sequencing by flight phase. Also shown in the figure is the mean value of 0.54 for the metric-element.

The mean value was used to compute an acceptance threshold for the group. The threshold was calculated to reject a percentage of the metric-element values. Three functions in Figure 2 exceed the threshold and warrant further evaluation. This technique allows the developer to identify areas where quality improvements are most needed and to apply resources accordingly. Since the threshold can be arbitrarily set, the procedure is used to lower the mean value as much as possible, consistent with available resources.

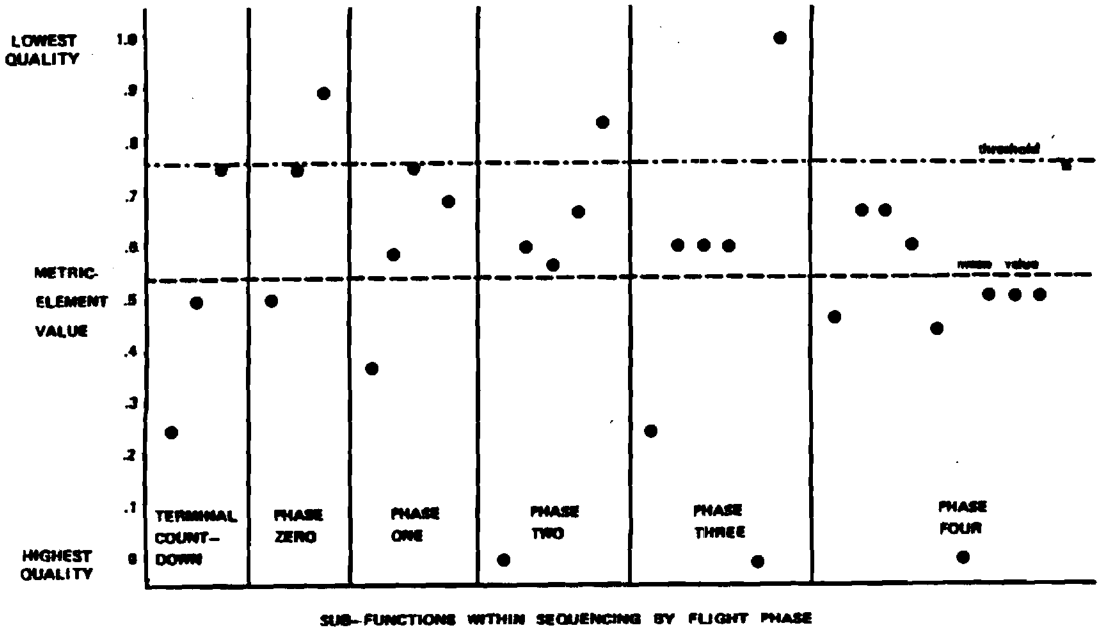


FIGURE 2. VALUES OF METRIC ELEMENT DATA REFERENCES HAVING ORIGIN FOR SEQUENCING SUBFUNCTIONS BY FLIGHT PHASE

Software Quality Assessment

The entries in Table 7 that are enclosed by circles were not affected by the incomplete translation of the specification; these values accurately reflect the specification that resulted from the requirements-phase activity.

One observation drawn from Table 7 relates to the metric-elements data references having origin and data references having destination. Data references having origin measures the completeness of references to data sources in the specification. Quality ratings for this element were low for both Sequencing and Steering because these functions referenced data that had no obvious point of origin. References having destination measures data usage for data produced by a function. In contrast to the earlier results, this metric element was satisfactory for Sequencing and Steering. One explanation for this difference follows. Functions are identified because data is required for other functions; thus the destination is known. Less is known about sources for data used by these functions; often this information is left unspecified. This is particularly true at the lowest functional levels.

Another observation relates to the low ratings for the metric-elements alternative condition options set, and alternative processing of conditions for Sequencing. Condition options control the functional flow within a system. Processing is identified for each option set. Typically, specification writers are success-oriented and deal explicitly with only one of two condition options; the alternative processing for the failure case is frequently overlooked. The ratings for alternative condition options set and alternative processing of conditions flagged the omissions.

CONCLUSIONS AND RECOMMENDATIONS

While the results are both limited and preliminary, they clearly demonstrate a quantitative technique for improving software quality that can be used during early requirements phases. The feedback to the developer isolates problems and pinpoints areas where additional resources should be applied to improve the quality of the specifications. The result is a more complete and less ambiguous specification for program design and test.

Integration of the software quality model with PSL/PSA had several advantages. First, PSL was used to remove ambiguities in the model by giving precise meaning to the data items. Second, defining the data items in PSL enabled the use of PSA to collect data automatically, thus reducing the cost for the actual measurement process. The third advantage is related to the nature of PSL/PSA. To a large extent, the benefits derived from PSL/PSA are determined by the sophisitication of the user; however, the software quality model provides a comprehensive set of review criteria for use with PSL/PSA, making it a much more powerful tool for an unsophisticated user.

Recommendations for Future Application

Despite these advantages, the requirements data base should be established early in the software development process to enhance the specifications and to avoid the costly translation process. In the future, measurements should be made for several programs to provide a sufficient basis for setting metric thresholds. Studies of cost versus quality levels are required to enable program managers, users, and maintainers to establish realistic goals early in the program. To reduce cost, as much of the technology as possible should be incorporated into the software development environment. Downstream, implementation of Ada® and the Ada Programming Support Environment will provide the opportunity to instrument the development environment and standardize on automated measurement and analysis tools. The goal will be to influence the day-to-day engineering decisions that have the greatest impact on software quality.

REFERENCES

1. Wood, D. L., "Department of Defense Software Quality Requirements", in Software Quality Management, edited by Cooper, J. D. and Fisher, M. J., New York: Petrocelli Books, Inc., 1979.

2. Curtis, B., "Experimental Evaluation of Software Characteristics", in Software Metrics, Perlis, A. J., Sayward, F. G., and Shaw, M., editors, Cambridge, Massachusetts: MIT Press, 1981.

3. Brown, J. R., "Programming Practices for Increased Software Quality", in Software Quality Management, edited by Cooper, J. D. and Fisher, M. J., New York: Petrocelli Books, Inc., 1979.

4. Arblaster, A. T., Worden, R. P., Knight, C. D., Software Quality Assurance Study, (Final Report, European Space Research and Technology Centre, Contract No. 4183/79/NL/PP (SC)), Logica Ltd., October 1980.

5. MIL-STD-1679A (Navy), Weapon System Software Development, May 1982.

6. MIL-STD-SDS (Proposed), Defense System Software Development, April 1982.

7. McCall, J., Richards, P., Walters, G., "Factors in Software Quality", 3 Vols., RADC-TR-77-369, Rome Air Development Center, 1977.

8. McCall, J. A., and Masumoto, M., Software Quality Metrics Enhancements, RADC-TR-80-109, Rome Air Development Center, 1980.

9. Francis, J., Town, D., Miller, J., San Antonio, R., The Application of Software Quality Metrics to MX Operational G&C Software, (Final Report, Contract No. F04704-81-C-0006), Dynamics Research Corporation, September 1982.

10. Teichrow, D., and Hershey, E., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processes Systems", IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 41-48.

11. Howden, W., "A Survey of Static Analysis Methods", in Tutorial: Software Testing and Validation Techniques, Miller, E., and Howden, W., 2nd Edition, New York: IEEE Computer Society Press, 1981, pp. 102-115.

ACKNOWLEDGEMENT

The work described in this report was funded, in part, under AF Contract No. F04704-81-C-0006 and DRC Independent Research and Development (IR&D) Project No. 3018. The Air Force Project Officers were Major Kenneth Jackson and Captain James Haugen. The principal investigator for this effort was Mr. James Francis. Key contributions were made by Dr. Donald Town (IR&D Program Manager), Mr. Jeffrey Miller, Ms. Mary Ann DiBurro and Ms. Sandra Bellone. Valuable comments and suggestions were provided throughout the effort by Mr. Al Peschel, Mr. Larry Hinckle, and Mr. Charles Hamilton of TRW, Inc.

Ada® is a registered trademark of the U. S. Department of Defense

ABOUT THE AUTHORS

MR. RALPH C. SAN ANTONIO, JR., manages the Software Engineering and Management Group at Dynamics Research Corporation. He holds a Bachelor of Science degree in Engineering from the U. S. Air Force Academy and a Master of Science degree in Engineering from UCLA. From 1968 to 1978, Mr. San Antonio served in the U. S. Air Force where he was involved in acquisition programs for satellite and communications systems. During this period, Mr. San Antonio contributed to Air Force policies and regulations for acquiring defense system software. He is a member of the IEEE and the National Security Industrial Association Software Group.

MAJOR KENNETH L. JACKSON is the U. S. Air Force program manager for the Peacekeeper Software Development and Advanced Technology in the Guidance and Control Division of the Ballistic Missile Office. He has responsibility for development and evaluation of the ground, flight, and missile targeting programs. Major Jackson received a Bachelor of Science degree in Electrical Engineering from Ohio University and a Master of Science degree in Electrical Engineering from the Air Force Institute of Technology. His prior responsibilities have included development and flight testing a Minuteman missile Global Positioning System (GPS) receiver used to evaluate guidance and control system performance. As a former Minuteman maintenance field engineer he has experience solving user problems during weapon system life-cycle support. He has co-authored several technical papers and has taught college level courses in programming.

MANAGING SOFTWARE TESTING USING RELIABILITY ESTIMATES *

Martin L. Shooman
Professor of Electrical Engineering and Computer Science
Polytechnic Institute of New York
Brooklyn, New York

ABSTRACT

One of the major problems in software testing is the lack of qualitative measures of progress. Software reliability models are now available which predict within reasonable accuracy the reliability, failure rate, mean time between failures, and number of expected errors during field operation. These models answer the question: "If I release the software now without further test and debugging, how well will it perform in use?" If the operational requirements contain quantitative software reliability goals for the system, the predicted reliability measures can be compared with the specified goals; then a cost benefit analysis can be performed to decide when to terminate testing.

The paper discusses the various reliability models which have been used in practice and developes a simple exponential reliability model. Several case studies are summarized where these models have been used successfully in practice. The paper concludes with suggestions as to how the use of software reliability models can be broadened, and what additional research is needed to further refine these models.

1.0 INTRODUCTION

1.1 Need for a Quantitative Measure

Software design involves an inherent paradox. Coding involves the synthesis of thousands of individual actions and decisions to form a program. Any mistakes result in a program with errors which will fail under certain conditions. Thus, the design of a program involves precise and exact ideas. Once we have created a program and ask the question how good the program is, we encounter just the antithesis of preciseness. People resort to all sorts of qualitative means to measure the quality of a program. Of course we can adopt the philosophy of Abraham Lincoln: "you may fool all of the people some of the time; you can even fool some of the people all of the time; but you can't fool all of the people all of the time." (Bartlett 1968, p. 641). Thus, we could simply release software to the field and see how well it performs. Of course this is a risky and often costly plan, due to the high cost of fixing software errors in the field compared with removing them during development testing. Also some software which is poorly designed will never perform satisfactorily in the field.

What is needed is a quantitative index of testing progress during development. This implies that a relative metric of quality would be saitsfactory as long as the ranges corresponding to "good" , "fair", and "poor" were known from past usage and experimentation. The reliability metrics which have been developed over the past decade and are proposed in this paper can not only serve as a relative criterion, but as and absolute measure as well. This requires that the specifications for the system be carefully drawn, and an adequate and realistic value be chosen for the reliability goal. For example, suppose we focus on MTBF as our software metric. If we know based upon analysis of collected data that predecessor software similar to the one under development exhibit a 200 hr. MTBF in use, and are considered good systems, then we can base our goals on this information. Based on the above example, typical specification philosophies and their probable results are given below:

1. The new system is much more complex then the old one. Thus, we specify MTBF=200 hr. as a difficult and realistic goal. The contractor will probably be able to improve design and development techniques sufficiently to meet this goal even with the new more complex system.
2. The new system is more complex then the old one, however, we have decided that reliability improvement is an important goal. Thus, we specify a MTBF goal of 500 hr. and ask the contractor in the proposal phase to describe in a quantitative plan (including prediction and reliability tests) how he intends to achieve the higher reliability goal.
3. The new system is more complex than the old one, and to insure that the reliability is improved we specify a MTBF of 2000 hr. No reliability plan is required, and the contract is to be awarded to the lowest bidder who meets the minimal technical requirements. The probable result is that the contractor will see that the goal is unrealistic, that there are no checks until the software is delivered, and will proceed on faith that his designers will meet the goal through good programming. The likely result is software with a MTBF < 200 hr., legal negotiations for a contract waver based upon a variety of issues, and a compromise which results in a poorer system than is required.

Result 1 and 2 are acceptable if we realize the compromises which have been made, however, result 3 is all too common and is generally a disaster.

* This work was supported in part by Office of Naval Research under Contract No. N00014-75-C-0858.

1.2 Focus on the Integration Test Phase

For the purposes of our discussion, we may divide the software production cycle into five phases: (1) Specification (including requirements and preliminary design); (2) Design (including detailed specifications, design representations such as pseudocode, HIPO, etc., and design reviews); Module (unit) Test of the control structure and individual modules; (3) Integration Test of the interfaces between the control structure and each module; (4) Simulation Testing and early field testing including alpha (in house) test sites and beta (early field) test sites; and (5) Operational Use in the field.

Our purpose is to formulate a software reliability model as early as is feasible to help guide the development of the project. One can formulate a software reliability model during the design phase but one can not measure the parameters of the model. If however, there exists a software reliability handbook with data recorded for previous projects, we can select a similar project and use the recorded parameters for a very rough initial estimate. This can help us predict the number of errors which will be found and give an estimate of the amount of testing required to bring the system to the specified reliability goal. A key requirement for most of the models which will be described is that run time and number of errors uncovered during testing be carefuly recorded. In most projects, the earliest one can begin to obtain accurate software reliability data is after the start of integration test. At this time, the software is placed under configuration control and the configuration manager collects written accounts of all errors and produces accurate records and counts. Thus, our focus will be on the integration test phase.

1.3 Types of Reliability Models

There are several types of reliability models, however, this paper focus on the class of macro reliability models which are discussed in detail in Sec. 3. In a macro model, one obviates the problem of how to model the program structure by focusing on only the number of errors in the software and not how severe they are or where they are located.

More detail can be modeled if one can represent the software structure by a program graph and if the program errors are associated with the various paths through the program, and the path execution frequencies. Such models are not discussed again until Sec. 5, since they are still in the research stages. (Shooman 1976).

Several individuals have suggested that one should be able to test a program with n test cases. If r of these tests succeed, then hopefully one could make a point and interval estimate of the software reliability based on n, r, and the underlying error distribution. Preliminary models of this type are discused in Chap. 5 of Shooman 1983; however, more research is needed in this area.

If we focus on the operational phase of a project, then not only reliability but availability is an important measure of system performance. Such models utilize markov probability models and are discussed in the literature.(Goel 1978, Trivedi 1975).

1.4 Experience with Software Reliability Models

Software reliability models have existed in the research literature for a little over 10 years. Over the last five years, demonstration studies have been made showing how the models can be employed in practice, and many of these are discussed in Sec. 4.0. Some practicioners have begun to use these models, while others await proof beyond the shadow of all possible doubt that the models work. Section 5.0 discusses additional research and demonstration work which is urgently needed to continue the momentum in this field.

1.5 Acceptence of Hardware Reliability Models

We may obtain further insight into some of the aversion toward software reliability models if we briefly study the broader reliability field. Some practitioners use the terms hardware reliability and software reliability, and we will adopt these terms even though precise distinctions are difficult with some systems. (Also human controlled computer complexes must also consider operator reliability.) Thus, we ask how successful is hardware reliability prediction and how well accepted are the results?

Hardware reliability (reliability theory) began in the late 1940's and early 1950's. A great amount of the early effort and a sizeable portion of the continuing effort in the field is directed toward collection, analysis, and documentation of field failures and a calculation of the associated failure rates. This data is published in failure rate handbooks and maintained in failure rate data bases, eg. MIL-HDBK 217, 217A, 217B, 217C; the Government Industry Data Exchange Program. A probabilistic reliability theory has been evolved which allows and analyst (or a computer program) to decompose a complex structure into its elementary parts. The handbooks are then consulted for part failure rates, and the reliability as a function of time is calculated. (Shooman 1968). If one wishes to express the result as a single metric rather then a function of time, the mean time between failures, MTBF (also called mean time to failure) is generally used.

The agreement which can be realized between reliability predictions made during design and subsequent field observations varies between very good and poor depending on the skill of the analyst, the completeness of the system design, and the quality and relevance of the failure rate data at hand. If the system structure is well described, and good failure rate data is available for the parts under environment and use conditions which are expected, then the MTBF can generally be predicted within 25%. Under poorer conditions, the estimates are within a factor of two; which is accurate enough since a conservative design allows ample margin for error.

Experienced reliability practitioners are generally satisfied with the accuracy of the results they can obtain, and understand the limitations when the system description is incomplete or when appropriate failure rate data is unavailable. Unfortunately, hardware reliability analysis is sometimes questioned for a variety of reasons (Shooman 1981) which are either invalid or related to a lack of familiarity with the field:

1. Some of the early practitioners were not always well trained.
2. Few university personnel were trained in reliability in the 1950's and 1960's.
3. Reliability is generally taught as a specialized graduate subject.
4. Reliability analyses are generally performed by a staff group in industry, rather than by the designer.
5. Management in the US often treats reliability as an imposed requirement (annoyance??) and not as a component of good design. (We all know only too well the success which the Japanese auto industry has had by adopting the opposite attitude)
6. The quality assurance (reliability) department is often viewed as playing an adversary role in the design process.

2.0 FOUNDATIONS OF A SOFTWARE RELIABILITY MODEL

2.1 Basis of the Probabilistic Model

To define a probabilistic model for program errors, we begin by considering all the paths in a program. For each path there are many combinations of initial conditions and input values comprising mutually exclusive execution sequences. Once software is placed in use, a certain number of residual errors exist. Thus, some of the execution sequences result in system failures. The choice of inputs and initial conditions selects which of the execution sequences will be processed. A software failure occurs when an execution sequence containing an error is processed. Although each execution sequence is deterministic, the combinations of input data and initial conditions in most real-time systems is unknown beforehand, thus these uncertainties provide the probabilistic element in the model.

2.2 Definition of Software Errors

Hardware errors occur due to poor-quality fabrication, design error, overload of the component, and wear-out.
Software errors occur due to reasons analogous to those of hardware, however, there is no software failure mode directly analogous to that of hardware wear-out. Another difference is that design errors represent perhaps 10% of all hardware failures, while 90% or more of all software errors are design errors.

At times it is important for clarity to be precise in our definition of software errors. We start by defining a system failure as a significant deviation from specified operation; a wrong answer, missing output, extraneous output, too slow response, garbeling of a data base, complete system crash, etc.. If the failure is due to a hardware problem, it is a hardware failure; if due to an operator error, it is a human failure; and if due to a software error, it is a software failure. Many modern computer systems contain multiple processors, redundant computations, and other features such that not all errors in the code will actually result in a system failure. Thus, for preciseness, we call internal code problems code faults. If the fault alters the system operation, then it is a software error. If the software error causes a significant change in system performance, then it is a software failure. In imprecise colloquail speech we call either errors or faults "bugs."

2.3 Definition of Software Reliability

The following definition of software reliability is commonly used:

Software reliability is the probability that a given software system operates for some time period without software error, on the machine for which it was designed given that it is used within design limits.

The above definition is simple, yet included within it are several subsidiary definitions which must be made with care. First of all, reliability is defined as a mathematical probability, which implies that we must make a probabilistic model as outlined in Sec. 2.1. This requires that we model the imput data to the software. The software system must also be accuratly described. For example are we going to count errors in the operating system or the FORTRAN compiler as system failures? The probability is a function of time, thus we must describe the inputs over time. (Often lacking more precise information a random or pseudorandom sequence is assumed.) One of the hardest tasks is to define what we mean by a (significant) software error. We are implying that the software error must be significant enough to cause a system failure. Note that use on a different computer, even if the two different models are preported to be interchangeable, may change things because of different computational speeds, timing of operations, minor differences in language translators (compilers, interperters, assemblers, etc.) We must also carefully state the design limits. As an example, it is well known that a time-sharing system designed for 50 users seldome operates as reliably with 45 users as with 10.

3.0 SOFTWARE RELIABILITY MODELS

3.1 Error Removal Model

As discussed in Sec. 1.3 we will concentrate on Macro Software Reliability models of exponential form. Several authors have proposed similar models (Jelinski 1972, Musa 1975), however, this discussion will center on the one described by this author in number of references (Shooman 1972, 1979, 1983). The model assumes that the program enters the integration test phase with E_T total errors, and as errors are found during integration they are promptly corrected. Thus, after τ months of integrat on testing, $E_c(\tau)$ errors have been corrected and the remaining number of errors is

$$E_r(\tau) = E_T - E_c(\tau) \qquad (1)$$

In a more advanced model (Shooman and Natarajan 1976), it is assumed that new errors are generated during development.

3.2 Failure Rate Model

If we assume that the failure rate, $z(t)$. is proportional to the number of remaining errors then

$$z(t) = K'E_r(\tau) \qquad (2)$$

where K′ is the constant of proportionality. In previous literature the author has often normalized Eq. (1) by dividing by the number of object code instructions, I_T. In such a case the right hand side of Eq. 2 is also divided by I_T.

3.3 Software Reliability Model

Using the principles of reliability theory (Shooman 1968), we obtain expressions for the reliability function, R(t), and the mean time between failures (MTBF):

$$R(t) = \exp(-K'E_r(\tau)t) = \exp(-K'(E_T - E_c(\tau))t) \qquad (3)$$

$$MTBF = 1/K'E_r(\tau) = 1/K'(E_T - E_c(\tau)) \qquad (4)$$

Study of Eq. 4 reveals a familiar reliability function where R is unity at t=0, and decays exponentially as the operating time, t, increases. The failure rate, $K'E_r(\tau)$, determines the rate of decay. If the test and debugging (redesign to correct detected errors) performed during the integration phase of software development is efficient, then at the planned release time, $\tau 1$, $E_c(1)$ will nearly equal E_T. This will result in a small failure rate, a slowly decaying reliability function, and a high MTBF (see Eq. 4). If at the planned end of the integration phase many errors are still left, the failure rate will be higher then desirable, and the MTBF will be below the prestated goal. The choices are to release a substandard software system to the field or to spend additional test time to reduce the number of residual e ors. Thus, modeling and sketching the R(t) and MTBF functions during integration testing provides a valuable means of judging the progress being made toward the specified reliability goal as the release time approaches. The MTBF function generally rises rapidly toward the end of testing, thus a model is very useful in making an accurate prediction. It is interesting to examine Eq. 4 for the case where there is only one error left. At this point the statistical assumptions of the model break down; however, this value, MTBF = 1/K′, can be viewed as sort of a limiting value.

3.4 Model Parameter Determination

In order to use the above model, we must estimate the parameters, K′ and E_T, for the model. We do this by placing the software under simulated use testing at two more points during the integration test phase, and record the number of failures and operating hours. This allows us to calculate the measured failure rate λ as the number of failures per hour, and the measured MTBF as the reciprocal of the failure rate. In Shooman 1972, 1979, 1983, a simulation program is proposed as the simulated use test. In Shooman 1979, early field tests serve as the simulated use test. Musa 1975, discusses the use of the integration test data along with an adjustment factor to serve as the simulated us test.

To use the reliability model described by Eqs. 1-4, we must have an accurate record of E_c, and must have a measurement of MTBF at a minimum of two different points in the development cycle. This leads to two simultaneous equations which are solved for the unknowns K′ and E_T. Using the development in Chap. 5 of Shooman 1983, we begin by writing the reciprocal of Eq. 4 at two different points.

$$\lambda 1 = K'(E_T - E_c(\tau 1)) \qquad (5)$$

$$\lambda 2 = K'(E_T - E_c(\tau 2)) \qquad (6)$$

Assuming that the number of removed errors $E_c(\tau)$ is carefully recorded by the configuration control manager, Eqs. 5 and 6 can be solved for K′ and E . (In statistical terms such a solution is called the modified method of moments).

Another way of estimating the model parameters is obtained by rewriting Eq. 5 for any time * τ and rearranging terms, so that

$$E_c(\tau_i) = E_T - 1/K'\lambda_i \qquad (7)$$

If we plot the number of errors corrected at time τ, $E_c(\tau_i)$, on the y-axis, and the failure rate, λ_i, on the x-axis, we obtain a straight line. The y-axis intercept is E_T, and the slope is $-1/K'$. A set of software error and failure rate data taken from Shooman 1979 is given in Fig. 1.

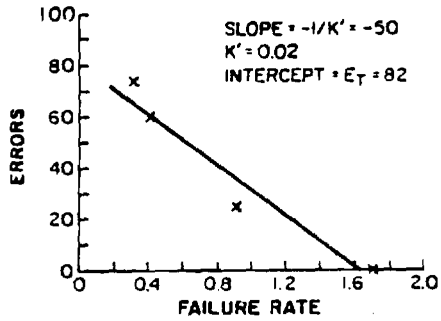


Fig. 1 Least Squares Fits of Error Data from Shooman 1979b. (Using TRS-80 Computer and Radio Shack Statistics Package No.26-1703.)

The parameters can be determined by fitting a straight line to the data by eye, or by using the well known technique of least squares estimation, readily available on most computers.

A powerful statistical method known as maximum likelihood estimation can also be used to estimate these parameters. (Shooman 1973, 1983). In the paper by Shooman and Schmidt 1982, a comparison is made of moment, least squares, and maximum likelihood as methods for estimating the model parameters. Since, no one method seemed to dominate over the others, the author recommends that least squares be used since it is the simplest and also provides a graph which helps to judge goodness of fit.

4.0 FIELD USE OF SOFTWARE RELIABILITY MODELS

Over the last decade reliability researchers have continued to develop the theory and apply it to field data. Although there are many models in existence, there are three models which have been called execution time models (Jelinski 1972, Shooman 1972, Musa 1975) which are quite similar, relatively easy to apply, and have been shown to yield good results. The key to using these models in practice is the collection of appropriate data during the integration test phase of software development. In fact, some of the comparisons which have been made in the past among models have yielded inconclusive results because of incomplete data.

The author believes that the best way to prove the utility of software reliability models is to cite the evidence which shows that they work. The remainder of this section will summarize several cases in which software reliability models have been shown to yield good results. For a further discussion of these cases the reader is referred to the cited references and to Shooman 1983, Chapter 5.

4.1 Error removal data

Dickson, Hesse, Kientz, and Shooman (1972) studied the error histories for three mainframe supervisory programs (operating systems) written by different manufacturers as well as the ground based software for four successive NASA Apollo missions. Data for a Bell Labs switching computer program (TSPS) was later added to the study. The conclusions were as follows:

1. The error removal data was quite similar for these 8 projects. The total number of errors removed per 1,000 lines of machine language code was 10.2 and the standard deviation of the data was 4.4.
2. The program sizes varied between 100K and 240K (machine language instructions).
3. The error removal rates per month were similar. A similar study was repeated with data on smaller programs (from 0.7K to 5.5K instructions) reported by Akiyama (1971) and yielded similar results.

4.2 Miyamoto's Data

Miyamoto (1975) applied the model described in Sec. 3 to a real-time message switching system developed between 1968 and 1970. The result of the study was a MTBF (Miyamoto called it mean time between software errors, MTBSE) curve which rose steeply near the end of debugging (see Fig. 2). This agrees with the predicted behavior, since the MTBF is proportional to the reciprocal of the number of remaining errors in Eq. 4.
Other interesting details of the study are given in the paper.

4.3 Musa's Data

Musa (1975) applied his execution time model to 16 different systems which were under development at Bell Labs. In four cases he compared the predicted MTBF with the measured MTBF when the system was placed in use. The results agreed quite well. The MTBF curve curve given in Fig. 3 exhibited a growth shape similar to that of Fig. 2. The first two rows of Table 1 show how closely Musa's predictions and measurements agreed.

In another study Musa (1979) investigated the validity of the assumption that the software failure rate is proportional to the number of remaining errors. This assumption is key to Musa's model and to the one developed in Sec. 3. Musa's data supported the assumption.

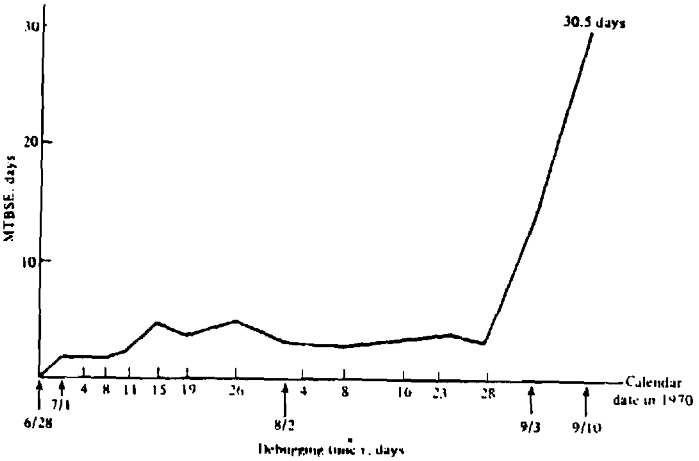


Fig.2 Growth curve of software reliability (mean time between software errors). (From Mivamoto, 1975, Fig.6)

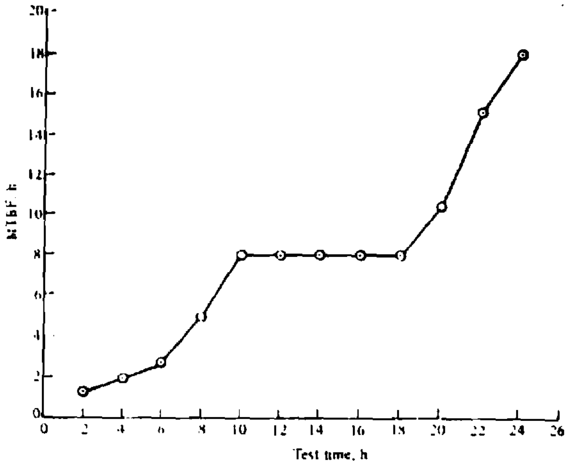


Fig.3 MTBF versus test time for project 1. (Replotted from the data of Fig.3,Musa,1975)

4.4 Shooman's Data

A software reliability model was used to study the quality of a software system undergoing early field/final development testing. (see Shooman 1979b). The purpose of the study was to determine whether the present software which had known problems could be perfected before a new and better alternate version of the software was ready in perhaps 6 months. The model predicted that it would take several hundred work days to perfect the present software. Based on their intuition and the analysis, management stopped further development of the present software and placed all their efforts behind the alternate system. Thus, although the data was imperfect, accurate enough predictions could be made to make management decisions. A detailed discussion of the project, the analysis of the data, and the conclusions appears in the reference.

4.5 Shooman-Schmidt Analysis

The data collected by Musa (1975) was further analyzed by Shooman and Schmidt (1982). The model developed in Sec. 3 was fitted to the same four projects described in Table 1. Several conclusions were drawn from this study:

1. Both the Musa model and the model of Sec. 3 worked well for the four projects.
2. The % error for the Musa model ranged between 0.3% and 58%, with an average of 34%.
3. The % error for the model of Sec. 3 ranged between 2% and 60% with an average of 37%.
4. The % error for the model of Sec. 3 varied from project to project depending on which parameter estimation technique was used (Maximum likelihood, least squares, moments). No method seemed consistently more accurate.

Details of the study are given in the reference.

	Project 1	Project 2	Project 3	Project 4
Measured (during use period)	14.6	31.4	30.3	9.2
Predicted (at end of test period) using maximum likelihood point estimate	19.1	35.2	24.4	12.3
50% confidence range	13.5–28.8	> 19.8	> 12.9	6.4–23.6
New or modified instructions*	19,500	6600	11,600	9000
Total program size	21,700	27,700	23,400	33,500
Number of programmers	9	5	6	7
Project length, months	12	11	12	10

Source: Musa (1975).

*Size given is the number of assembly or machine language instructions.

Table 1 Comparison of measured and predicted MTBF,h

4.6 Richeson's Space Shuttle Data

The model described in Sec. 3 was used to analyze software data taken during a simulated NASA mission. The programs, approximately 1.2 million lines of source code, represented the ground based software for the Space Shuttle Mission Control Center Data Processing Complex. (Richeson 1981a, 1981b, Shooman and Richeson 1983). The model predicted the software failure rate. Multiplying the failure rate by the mission duration gives the expected number of mission software failures. If it is assumed that the number of failure has a Poisson distribution, (this is consistent with the exponential model for time between failures which we have been considering), then an interval estimate can be computed.

The conclusions given in Shooman and Richeson 1983 were:

1. It is feasible to gather error and run time data during the simulation of large real-time software systems.
2. The model of Sec. 3 can be used along with this data to predict the number or errors, failure rate, and MTBF during a specified operational period.
3. The number of software DR's (discrepancy reports) recorded during the first Space Shuttle mission (critical + major + minor) was 17. The expected number from the model was 11, and for a 95% confidence band, 17 or fewer were expected.
4. If only critical DR's are considered, the model predicts that the expected number is 2 and that for a 95% confidence band, 4.4 or fewer critical DR's were expected. During the mission none were reported.
5. If critical + major DR's are considered, the model predicts that the expected number is 5, and for a 95% confidence band, 9.8 or fewer critical + major DR's were expected. During the mission 7 were reported.

Further details are discussed in the three references cited.

5.0 FUTURE PLANS

The evidence cited in the previous section shows that software reliability models do work if the proper data exists. Unfortunately, many practitioners are not familiar with these studies, while others still wish more evidence. During the mid 1970's there was substantial DOD funding for reliability research, however, just as the research was beginning to bear fruit, this funding was curtailed. This author suggests that this situation be reversed. Specific suggestions are given below:

1. Several demonstration projects be funded to show how reliability prediction, data collection, measurement during development, and field measurements can be performed as software is developed. This could best be accomplished via a University company team.
2. A group of established analysts be funded to study the major reliability models and the existing software reliability data bases and compute the resulting model constants. This set of examples along with a clear discussion of how to apply the various models would constitute a software reliability application manual. In 1978 a group of the leading software reliability modelers offered to work together on such a project, however, only a small amount of progress has been made to date.
3. A group of portable, user friendly, well documented programs should be developed to support the application manual described above. Some aspects of artificial intelligence could be incorporated to help the inexperienced

analyst explore the data, look for data problems, choose among models, etc.

4. More research is needed to develop micro reliability models. Only early research results have appeared, and a sustained effort is needed if further progress is to be made in this difficult, but important area.
5. Research is needed to define quantitative acceptance tests, the associated consumer and producer risks, methods for constructing test data, and demonstration projects, culminating in a future acceptance test standard.
6. Research should be done to construct cost/benefit models to help determine more scientifically when to stop furrther testing and release a system.

6.0 REFERENCES

Akiyama, Fumio: "An Example of Software System Debugging," Proc. IFIP Congress '71, Ljubljana, Yugoslavia, American Federation of Information Processing Societies, Montvale, N.J., 1971.

Bartlett, John: "Bartlett's Familiar Quotations," Little. Brown and Company, Boston, Fourteenth Edition, 1968.

Beizer, Boris: "Software Testing Techniques," Van Nostrand Reinhold Co., New York, 1983.

Dickson, J., J. Hesse, A. Kientz, and M. Shooman: "Quantitative Analysis of Software Reliability," Proc. Ann. Reliability and Maintainability Symp., IEEE, January 1972.

Freiberger, Walter, ed.: "Statistical Computer Performance Evaluation," Academic Press, New York, 1972.

Goel, A. L. and K. Okumoto: "Bayesian Software Prediction Models," RADC TR-78-155, July 1978.

Hetzel, William C.:"Program Test Methods," Prentice-Hall, Inc., Englewood Cliffs, N. J. , 1973.

Jelinski, Z., and P. Moranda: "Software Reliability Research," pp. 465-484 in Freiberger (1972).

Miyamoto, Isao: "Software Reliability in Online Real Time Environment," Proc. Inter. Conf. on Reliable Software, IEEE catalog no. 75CH0940-7CSR, April 1975, p. 194

Musa, John D.: "A Theory of Software Reliability and Its Application," IEEE Trans. Software Eng., vol. SE-1, no. 3, September 1975, pp. 312-327.

Myers, Glenford J.: "Software Reliability Principles and Practice," John Wiley & Sons, New York, 1976.

Richeson, George: "Software Reliability Data Analysis and Model Fitting for the Shuttle Data Processing Complex Real-Time Applications", Internal Memorandum, Lyndon B. Johnson Space Center, NASA Houston, Jan. 1981.

Richeson, George: "DR Analysis for STS-1 A Comparison of Software Reliability Model Prediction and Actual Error Occurrence", Internal Memorandum, Lyndon B. Johnson Space Center, NASA Houston, April 1981.

Shooman, Martin L.: "Probabilistic Reliability: An Engineering Approach," McGraw-Hill Book Co., New York, 1968.

Shooman, Martin L.: "Probabilistic Models for Software Reliability Prediction," pp. 485-502 in Freiberger (1972).

Shooman, Martin L.: "Operational Testing and Software Reliability Estimation during Program Development," Rec. 1973 IEEE Symp. Comp. Software Reliability, catalog no. 73 CH0741-9CSR, New York, Apr. 30, 1973, pp. 51-57.

Shooman, Martin L.: "Structural Models for Software Reliability Prediction,"Proc. 2d Int. Conf. Software Eng., IEEE Computer Society, October 1976, pp. 268-280.

Shooman, Martin L.: "Software Reliability," Chap. 9 in Computing Systems Reliability, T. Anderson and B. Rendell, eds., Cambridge University Press, New York, 1979a.

Shooman, Martin L.: "Software Reliability Data Analysis and Model Fitting", Workshop on Quantitative Software Models for Reliability, Complexity, and Cost: An Assessment of the State of the Art, Kiamesha Lake , N.Y., Oct. 9-11, 1979b, IEEE Cat. No. TH0067-9, New York.

Shooman, Martin L.: "The Transfer of Reliability Research Results to Engineering Practice," presented at the ONR/ARO Reliability Workshop, Department of Commerce Auditorium, Washington, D.C., May 1, 1981.

Shooman, Martin L.: "Software Engineering: Design, Reliability, Management," McGraw-Hill, New York, 1983.

Shooman, Martin L. and Srinivasan Natarajan: "Effect of Manpower Deployment and Bug Generation on Software Error Models," Proc. Symp. Software Eng., Polytechnic Press, New York, 1976, pp. 155-170.

Shooman, Martin L., and Henry Ruston, 1979 Final Report Software Modeling Studies," Report SRS 119/POLY EE 80-006, Vol. I, Polytechnic Institute of New York, Dec. 31, 1979.

Shooman, Martin L. and R. W. Schmidt: "Fitting of Software Error and Reliability Models to Field Failure Data," in "Applied Probability - Computer Science the Interface," eds., Disney and Ott, Birkhauser Boston Inc., 1982, vol. I, pp. 299-327.

Shooman, Martin L. and George Richeson: "Reliability of Shuttle Mission Control Center Software," Proceedings Annual Reliability and Maintainability Symposium, IEEE Jan. 1983.

"Software Engineering Research Review-Quantitative Software Models," Data and Analysis Center for Software, Rome Air Development Center, Griffiss Air Force Base, New York, 13441, March 1979.

Thayer, Thomas A., Myron Lipow, and Eldred C. Nelson: "Software Reliability: A Study of a Large Project Reality," North-Holland Publishing Co., New York 1978.

Trivedi, Ashok K., and Martin L. Shooman: "A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters," Proc. Inter.

ABOUT THE AUTHOR

Dr. Martin Shooman is Professor and Director of the Computer Science Division at the Polytechnic Institute of New York. He holds BS and MS degrees in Electrical Engineering from MIT and a Doctorate in Electrical Engineering from the Polytechnic Institute of Brooklyn. He has published over 100 papers, articles, and research reports on reliability theory, software engineering, and control systems, and has contributed chapters to 8 books. He is author of "Probabilistic Reliability," 1968 and the recent text "Software Engineering, " 1983. Professor Shooman is a fellow of the IEEE and has received four best paper awards from the Reliability and Computer Societies.

SOFTWARE ERROR STUDIES

Carolyn Gannon
General Research Corporation
Santa Barbara, California

ABSTRACT

The collection and analysis of data on errors in software are discussed, to illustrate the payoff of software error studies and to generate interest in making such data collection and analysis an integral part of software development projects.

Major goals in studying software errors are reviewed, both near-term and long-term. Near-term payoffs for collecting and analyzing error data include:

- Pinpointing problem areas
- Demonstrating reliability (through the absence of errors)
- Tracking project status (according to types of error and the development phases in which they are detected)
- Guiding the testing process

Long-term payoffs include

- Developing models for software cost and reliability estimation
- Focusing the development of software analysis tools
- Refining computer languages

Well-known studies of software errors are reviewed. These studies, most of them sponsored by the US Government, provide insight into the classification of software errors and how they are sometimes used in estimating software quality and costs.

Methods for collecting and categorizing software error data are described. In particular, the planning, collecting, analyzing, archiving, and information diseminating processes are investigated. In conclusion, mechanisms are recommended to aid in the recording and analysis of software error data.

INTRODUCTION

The primary motivation for collecting data on software errors is to improve the quality of delivered software. These data are valuable not only to assess the quality of the software from which they are gathered but also to provide "lessons learned" data for the next similar project. Often errors are repeated, even with the same programming staff, on several similar projects merely because of the complexity of the software and inability to remember the details of past experience.

Another important use for databases of software error history is to assist in Independent Verification and Validation (IV&V) and to provide actual data on types of errors to be addressed by future software tools. Software tools are often developed by companies that do not themselves generate large quantities of application software. Therefore, the tool developers must rely on intuition or on published studies to decide what types of errors their tools should prevent or detect.

Computer languages are continually undergoing refinement (such as the many dialects of JOVIAL) and new definition (as in the case of Ada). While the main purpose of these developments is to facilitate programming for specific application areas, there is no reason why the prevention of common errors cannot also be a motivating factor for defining certain language constructs.

Each of these potential payoffs for learning more about software errors assumes that such data have been collected and classified. Further, the data should be accessible in an easy-to-digest form. For example, a programmer about to undertake the development of a specific application might find it very helpful to know what kinds of errors have been encountered by colleagues who used the same computer language on similar computers for applications of similar characteristics (program algorithms, size, interfaces, design, etc.).

In spite of these payoffs, there are many reasons why good databases of software error information are not commonly maintained:

- There is no standard classification of software errors.
- Tools for automatically collecting error data are not widely used.
- Most projects have no budget of funds and time for error data collection.
- Companies do not wish to publicize their errors (and programmers do not wish to admit their errors).

Assuming that software error data are collected and preserved, they must be easily accessible. For use by a program manager in scheduling and budgeting an upcoming project, the most accessible form might be a printed report, in which the data is already categorized. Tool or language researchers might prefer the data in raw, machine-readable form, to be stored in their own database formats for a variety of analyses, depending upon the objective. Programmers might be assisted by an "expert system" with an underlying database of software errors and a rule base of symptoms for determing the cause of a problem. Therefore, once collected, the type of storage and retrieval of software error data is also an important issue.

A number of software error studies have been performed in the past ten years. Some of these are described later in this paper. Several Government agencies maintain machine-readable repositories of data from software projects. One system provides on-line abstracts of problem reports over the ARPANET [1]. With the recent advances in database querying and in expert systems, the time seems ripe for consolidating the collection, storage, analysis, and selection of error data into some prototype error-data systems.

COLLECTION OF DATA ON SOFTWARE ERRORS

In 1978 Robert Thibodeau studied the state of the art in software error data collection and analysis [2]. He found that one of the leaders in collecting and analyzing project and software data was Rome Air Development Center (RADC) at Griffiss AFB, New York. Since that time RADC has continued sponsorship in these areas. To set the stage, RADC sponsored a conference on software data collection in 1976 [3]. In 1981 Jane Radatz based an analysis of IV&V data on project data available through RADC [4], and the Data & Analysis Center for Software (DACS) offers magnetic tapes and reports containing software data [5]. Thibodeau's work, sponsored by the Army Institute for Research in Management Information and Computer Science (AIRMICS), provides an in-depth survey of software error data collection and is summarized here.

Table 1 lists representative sponsors and organizations in the field of software error data collection and analysis. Most of the research in this field has been directed at measuring or improving software reliability. Collected project and error data have formed the basis for the results of these efforts.

In addition to the software error data described in reports generated by the projects noted in Table 1, data collections are available from

- USAF RADC DACS
- USA Ballistic Missile Defense Systems Command (BMDSCOM)
- USAF Data Systems Design Center

The DACS set of software data include a magnetic tape, hardcopy listing, reports, and data collection forms. These products are now offerred on a fee basis. They are the result of a study by System Development Corporation that produced an eight-volume report [6]. SDC studied a number of aspects of the data collection problem including cost, resistance of software developers, interference in the software development process, and methods of storage and access.

According to Thibodeau, software error data collected in the past has suffered in quality for four reasons:

1. The classification criteria were not clearly described.
2. The errors were not necessarily classified by the same persons who reported the problems.
3. The intensity or extent of testing was not known.
4. Data subject to change (such as program size) were not recorded at the same time that the problems were experienced.

Therefore, some of Thibodeau's recommendations for software error data collection are:

- Explicit requirements for the type of data should be established before the data are collected, by setting up a system of objectives and priorities.
- Methods of automated data collection should be investigated.

At General Research Corporation, we use PRIS (Problem Report Information System) [18] for recording software errors. PRIS is an interactive menu-driven program that keeps track of open and closed Problem Reports (PRs). It is the sort of small tool that can be developed for a project at little cost. For each error, PRIS records a severity level, a PR number, problem description, reporting person's identification, software location or version. The date is embedded in the PR number. PRIS has menus for inputting new problems, modifying selected data fields, displaying error status, and printing the open or closed set of errors. The primary output is a concise report of the error status; however, the PRIS working file or output file could easily be saved for later analysis of error types or for archival. Associated with PRIS are two forms: the PR form, used for recording symptoms too lengthy or ambiguous to input directly into PRIS, and a PR "Fix" form. This form, filled out by the error-correcting programmer, contains information about what statements and modules were changed to correct the error and what versions (or "deltas" if a configuration management tool is being used) reflect the changed code.

Software error data collection begins with a classification of errors into types. Whether the classification is for a particular project or for generalization, the task is not easy. An appendix to Radatz's IV&V study report [4] provides a concise list of major error categorizations from the literature. The next section of this paper deals with software error categories.

SOFTWARE ERROR CATEGORIES

Classifying software errors is a prerequisite for measuring the quality of software and determining the effect of applying manual or automated tools to the software. There are many error classification schemes in the literature. Since software errors can be defined as deficiencies in design, code, or documentation that cause the resulting program to perform differently than intended, the range of error categories is large.

One major difficulty in determining a standard classification of errors is that some errors exist only in certain applications. Another difficulty lies in the distinction between "symptom" and "error." Sometimes only symptoms are recognized, and it is months before the real error is determined.

TABLE 1. SUMMARY OF RESEARCHERS IN THE COLLECTION AND ANALYSIS OF SOFTWARE ERROR DATA

Research Sponsor	Company	Principal Author	Topic
USAF/RADC	SDC	Finfer	SW Data Collection
	Logicon	Radatz	Impact of IV&V
	RADC	Sukert	Reliability modeling
	Raytheon	Wiliman	System reliability history
	SDC	Willmorth	SW data collection
	IBM	Baker	Data collection, project history
	IBM	Motley	Prediction of programming errors
	TRW	Brown	Impact of programming practices
	TRW	Thayer	SW reliability
	PINY	Shooman	Reliability measurement models
	PINY	Trivedi	Prediction of SW performance models
	MITRE	Amory	Error classification
USAF/SAMSO	Aerospace	Callender	Industrial practices
USAF/ESD	GRC	Graver	SW development costs
USAF/AFOSR	GRC	Gannon	SW test techniques
USA/BMDATC	Logicon Logicon	 Lambert	Error classification Reliable SW study
USA/Frankford	SofTech	Goodenough	Test data selection
USA/AIRMICS	GRC	Thibodeau	Error data collection
EPRI	GRC	Saib	SW validation
NSF	Sperry Univac NYU	Ostrand Weyuker	Error collection and categorization
NASA	Aerospace	Hecht	SW reliability measurement
NADC	NPGS	Bradley	Structure and errors

TABLE 1 (continued)

Research Sponsor	Company	Principal Author	Topic
Unknown Sponsors	Aerospace	Hecht	SW/HW reliability
	Aerospace	Reifer	Test tool overview
	Fujitsu	Akiyama	Prediction of SW bugs
	IBM	Belady	Large program development
	IBM	Endres	Errors and causes
		Gilb	Unreliability
		Walsh	Structured testing
		Yourdon	Reliability measurements
	Inst. for Adv. Tech.	Odin	Reliability
	Logicon	Dana	Classify/detect errors
	Logicon	Rubey	SW validation
	McDon. Doug.	Moranda	Predicting SW reliability
	Naval Post.	Schneidewind	Error processes
	PINY	Shooman	Programming errors
	RAND	Boehm	SW development
	Swedish Nat. Defense Inst.	Palme	Languages for reliable SW
	SEL	Hausen	Measuring reliability
	Tracor	Sontz	Quality assurance
	USA/SAFEGUARD	Dickson	Reliability analysis
	USAF/Ogden	Shelley	SW reliability
	USN/NSRDC	Culpepper	SW reliability
	Voest-Alpine	Kopetz	Error detection
	UCSB	Pyster	Error classification
	Hughes	Bowen	Error classification

In 1979 Arthur Pyster ran some experiments at the University of California, Santa Barbara, in which his students tried to classify errors in a set of programs according to several well-known schemata [7]. The results of his experiments showed the difficulties of identifying errors rather than symptoms, and of classifying errors by schemata with a large number of categories and subcategories.

In a recent report, Ostrand and Weyuker [8] state that there are serious problems with all classification schemata available in the literature. The main problems are that categories are ambiguous, overlapping, and incomplete. Like Pyster, they found that many schemata have too many categories and that there is confusion among error symptoms, error causes, and actual errors.

In spite of the flaws in existing software error classifications, it is useful to review them before deriving a schema to apply to a specific project. Radatz [4] listed the following error categorizations:

• Amory and Clapp	[Ret. 9]
• Rubey	[Ref. 10]
• Dana and Blizzard	[Ref. 11]
• Thayer, et. al.	[Ref. 12]
• Hartwick	[Ref. 13]
• Endres	[Ref. 14]
• Bowen	[Ref. 15]
• AN/SLQ-32(V)	[Ref. 15]
• Baker	[Ref. 16]
• Fries	[Ref. 17]

This paper does not attempt to evaluate software error classification schemata or recommend a particular one. The Bowen paper [15] and the Ostrand and Weyuker report [8], as well as others, assess several existing classification schemata along with proposing their own. Schemata with a relatively small number of major categories seem to be the easiest to use and to adapt to particular projects. Several such classifications are shown in Table 2. Besides these major categories, some schemata also include subclassifications such as:

- A severity classification [Bowen, ref. 15]
- Dimensions of where, what, how, when, and why (Amory and Clapp, [9])

TABLE 2. COMMON MAJOR CATEGORIES OF SOFTWARE ERRORS

Thayer, et al.	—	Computation
		Logic
		Data input
		Data handling
		Data output
		Interface
		Data definition
		Database
		Operation
		Other
		Documentation
Amory, Clapp	—	Input data
		Internal data
		Computation procedures
		Control procedures
		Interface procedures
Bowen	—	Design
		Interface
		Data definition
		Logic
		Data handling
		Computational
		Other
Ostrand, Weyuker	—	Data definition
		Data handling
		Test (i.e., evaluate a condition)
		Test plus process (i.e., evaluate a condition and perform a specific computation)
		Documentation
		System
		Not an error (problems that are resolved without changing the product)

- Attribute categorization (Ostrand and Weyuker, [8])

In all of the error classification schemata, the phase of the software development during which an error occurs is an important datum. It is commonly known that early detection of errors reduces software cost. Besides the usefulness of this information in estimating costs, it is helpful to software tool developers to know whether an error occurred during the requirements, design, coding, or post-coding phases (or some iterative cycle of these phases, in a "rapid prototyping with successive refinement" approach to development). Radatz used "anomaly" categories geared to the life-cycle phases, in which some of the subcategories are intentionally redundant. This scheme, reported by Hartwick [13], is shown in Table 3.

TABLE 3. ANOMALY CATEGORIES KEYED TO PHASES

Requirements Specification Anomalies	
R1.	Incorrect Requirements
R2.	Inconsistent Requirements
R3.	Incomplete Requirements
R4.	Other Requirement Problems
R5.	Presentation, Standards Compliance
Before-Code Design Specification Anomalies	
D1.	Requirement Compliance
D2.	Choice of Algorithm, Mathematics
D3.	Sequence of Operations
D4.	Data Definition
D5.	Data Handling
D6.	Timing, Interruptibility
D7.	Interfaces, I/O
D8.	Other Design Problems
D9.	Presentation, Standards Compliance
Code Anomalies	
C1.	Requirement, Design Compliance
C2.	Choice of Algorithm, Mathematics
C3.	Sequence of Operations
C4.	Data Definition
C5.	Data Handling
C6.	Timing, Interruptibility
C7.	Interfaces, I/O
C8.	Other Code Problems
C9.	Presentation, Standards Compliance
After-Code Design Specification Anomalies	
P1.	Incorrect Documentation
P2.	Inconsistent Documentation
P3.	Incomplete Documentation
P4.	Other Documentation Problems
P5.	Presentation, Standards Compliance

Since the purpose of Radatz's study was to evaluate the effectiveness of IV&V on several very large projects, it is clear why she selected these subcategories. Many of them address the issues that IV&V contractors are expected to look for, such as incompleteness and inconsistencies. It is obvious, however, that in such a classification some errors are more serious than others. Indeed, depending upon the goals of the developer or IV&V contractor, both the subcategories and the weightings can vary. For example, programs developed for a high degree of portability may weight "interfaces, I/O" more heavily than "sequence of operations."

While there has been considerable activity in the classification of software errors, efforts should continue to specify classes of errors according to project type, development approach, and severity. If certain standard error categories can be effectively determined (even if they continue to evolve as languages and development approaches change), the ability of IV&V tools to detect and even prevent errors can be greatly improved.

COSTS AND PAYOFFS

The long-term benefits of collecting and analyzing software errors cannot be easily quantified. To do so, one would have to consider the combined benefits of better software cost estimating models; tools for software requirements specification, design, and testing that make use of knowledge of what kinds of errors are prevalent; and computer languages that have been developed to be error-resistant. Even in the near term, within the life of a project, one can only guess at the value to managers and programmers of having an analyzable record of outstanding and corrected errors.

Although IV&V is a different activity from error data collection and analysis, I believe that the ratios of costs to payoffs for the two activities are similar. Radatz found that IV&V costs an average of 25% of the software development cost of a project (approximately 20% of the total software acquisition cost, which includes administrative costs). Her report provides numerous charts that justify her statement that IV&V pays for itself through the early detection of anomalies. She also showed that IV&V affected few of the factors known to influence programmer productivity, indicating that the process did not add much overhead to the programmers' activities. The factors that did add overhead are similar to those that would occur in an error data collection and analysis activity. These include:

- Documentation requirements
- Percentage of support staff
- Error reporting and correcting procedures
- Need to share computing facilities
- Classified security environment
- Meetings and interfaces
- Secondary resources (computer time, documentation reproduction, etc.)

The keys to minimizing the overhead associated with software error collection and analysis as part of a software development project are to (1) facilitate the error reporting mechanism and (2) provide efficient feedback to managers on the status of the software and to programmers on the resolution of the errors. To provide a quick and accurate mechanism for error data recording, the system should be an on-line program (assuming that most programmers are working at terminals) and "first-person" (that is, the person discovering the error reports it).

RECOMMENDATIONS

The use of a simple recording and reporting system (such as PRIS, described earlier) can greatly reduce the resistance of both programmers and managers to tracking software errors. It also provides part of the database of information that can be used for research in software reliability, test tool development, and software cost estimation. What additional information is necessary depends on the anticipated use of the database; most such information can be easily supplied by a project manager before the data collection activity begins. As an aid similar to an on-line error recording system, a project description system should be available in a set of project management tools to input such characteristics as:

- Program description
- Implementation language
- Computer hardware
- Estimated program size
- Programming staff proficiency levels
- Project schedule
- Software development and test tools to be used

In order to achieve the near- and long-term goals for software error data collection and analysis described earlier, the following recommendations are made:

1. Determine the objectives of the data collection effort before it begins.
2. Get management support (funding and schedule) for data collection and analysis.
3. Identify a reasonable set of data to collect (a minimal set of pertinent data is much more valuable than a large set of vague or superfluous data).
4. Build or acquire easy-to-use data recording tools.
5. Study the literature for a candidate error classification scheme (or customize one for your own needs).
6. Store the error data in machine-readable form.
7. Make the error data and/or analysis available to other projects within the company, and to DACS or other institutions for use in research.

Encouragement should also be given to building additional tools for performing analyses using the compiled error data. In the future, a knowledge-based tool (expert system) with access to an error database could aid in making judgements on error classification, or select (given the symptoms of a problem) a test tool for tracking down an error.

ACKNOWLEDGEMENTS

I wish to thank Robert Thibodeau for providing some of the referenced background material and for his review of this paper. He made many helpful suggestions. I also wish to thank Sabina Saib and Dana Nance, both from GRC, for their review and edit.

REFERENCES

1. The AFDSDC ADS Project Approval/Development Process, AFDSDCM 300-8 (TEST), AF Data Systems Design Center, Gunter AFS, June 1976.

2. R. Thibodeau, The State-of-the-Art in Software Error Data Collection and Analysis, General Research Corporation, NTIS ADA075228, January 1978.

3. N. E. Willmorth, Proceedings of Data Collection Problem Conference, RADC-TR-76-329, Vol. VI, December 1976.

4. J. W. Radatz, Analysis of IV&V Data, Logicon, Inc., 31 March 1981.

5. DACS Newsletter, RADC/ISISI, Griffiss AFB, NY, Vol. III No. 5, June 1982.

6. N. E. Willmorth, et al., Software Data Collection Study (8 vols.), System Development Corporation, TM-5524/001/01, December 1976.

7. A. B. Pyster, The Need for Better Error Classification, Computer Science Technical Report Series, TRCs79-2, University of California, February 1979.

8. T. J. Ostrand and E. J. Weyuker, Collecting and Categorizing Software Error Data In an Industrial Environment, Tech. Report 47, August 1982.

9. W. Amory and J. A. Clapp, Engineering of Quality Software Systems (A Software Error Classification Methodology), RADC-TR-74-324, Vol. VII, January 1975.

10. R. J. Rubey, "Quantitative Aspects of Software Validation," Proceedings of the International Conference on Reliable Software, April 1975.

11. J. A. Dana and J. D. Blizzard, The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 1974.

12. T. A. Thayer, et al., Software Reliability Study, TRW Defense and Space Systems Group, RADC-TR-76-238, 1976.

13. R. D. Hartwick, Software Acceptance Criteria Panel Report, Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management, Software Workshop, April 1979.

14. A. Endres, "An Analysis of Errors and Their Causes in System Programs," IEEE Tranactions on Software Engineering, Vol. SE-1, No. 2, June 1975.

15. J. B. Bowen, "Standard Error Classification to Support Software Reliability Assessment," National Computer Conference, 1980.

16. W. F. Baker, Software Data Collection and Analysis: Real-Time System Project History, IBM Corporation, RADC-TR-77-192, June 1977.

17. H. J. Fries, Software Data Acquisition, RADC-TR-77-130, April 1977.

18. The Software Workshop, "Problem Report Information System User's Guide, General Research Corporation, RM-2450, 1982.

ABOUT THE AUTHOR

Ms. Carolyn Gannon is the Director of the Software Technology Department at General Research Corporation. She is currently managing projects in the areas of software testing and public safety command and control systems. Efforts in the area of software testing have included software error analysis and development of Automated Verification Systems for JOVIAL and FORTRAN. Ms. Gannon holds a Master's degree in Computer Science and Bachelor's degree in Mathematics, both from the University of California, Santa Barbara.

EXPERIENCE IN TESTING LARGE EMBEDDED SOFTWARE SYSTEMS

John B. Bowen
Systems Engineer

Marion F. Moon
Chief Scientist

Hughes-Fullerton
Fullerton, CA

ABSTRACT

Hughes-Fullerton's approach to test and evaluation (T&E) of large embedded software systems has emphasized the test-bed environment; however, current research thrusts are directed toward formalizing individual test phases such as software integration. This paper presents an overview of the Hughes-Fullerton software development cycle, discusses T&E results of a current large-scale air defense project, relates experiences in using simulators, and presents a position on the effectiveness of independent verification and validation as well as endurance testing.

INTRODUCTION AND BACKGROUND

The test and evaluation of software in large embedded systems developed by Hughes-Fullerton started over twenty years ago with an air defense ground environment system for a foreign government. At that time testing was performed by a team of software engineers who had previously written the requirements specifications. These engineers, by first writing the functional specifications and then testing the integrated software for adherence to those specifications, performed a closed-loop validation -- but the developers and evaluators were not organizationally independent. The programmers performed their own parameter and assembly tests, and their methodologies and code were rarely evaluated for quality or standards. Today Hughes employs new methodologies and facilities such as structured design and interactive development systems. Although we also employ some advanced testing techniques and tools, we still recognize a need for improvement. For over ten years we have been using independent test teams with success, and are pursuing research in topics such as error persistency and design for testability.

Hughes-Fullerton business is primarily large-scale systems that include embedded software with more than 3,000 modules.* We are currently developing software for some twenty distinct projects. To assist in managing the quality of so many activities, the Software Engineering Division (SED), which provides programming services for these projects, has promulgated a set of software engineering procedures. SED also has a Software Engineering and Technology Department whose charter is to keep the work focused on the state of the art. Their efforts include tradeoff studies, generation and maintenance of development tools, and technology transfer in areas such as T&E.

OVERVIEW OF STANDARD SOFTWARE DEVELOPMENT PROCEDURES

Hughes-Fullerton follows the software development phases generally accepted by the industry and the Government. Those phases are: requirements analysis, design, coding, parameter and assembly testing (also called unit testing), software integration, independent testing, system testing, and installation testing.

Requirements Analysis

The software requirement specifications are often written by the lead systems engineering group. Generally SED is consulted or participates in the generation of these specifications. However, the first stage that SED is officially responsible for is analysis of the specification. This phase consists of assessing the feasibility of implementing the specifications in software, determining if there is existing software responsive to similar requirements, and generating independent test plans based on specification requirements.

Design

Hughes employs a programmer team concept organized by Computer Program Component (CPC) or in some cases by Computer Program Configuration Item (CPCI). Examples of CPCs are weapons, surveillance, data recording, and diagnostics. The team leader is completely responsible for the detailed design, coding, and checkout of the software in the particular CPC. As a rule, modules undergo code reviews by the team leader, and upon successful completion of the review are usually placed under configuration control. (Some projects place their modules under configuration control after completion of parameter and assembly testing.)

Hughes-Fullerton employs Constantine and Yourdon's structured design methodology supported by an automated interactive graphics and metric tool for decomposition of the software design to the module level. Intramodule design is controlled by SED training courses, individual project standards, and detailed design reviews.

Coding

Hughes coding standards restrict programming control structure to the five basic structures: Sequence, If-Then-Else, Do While, Do Until, and Case. The standards also contain module and data naming conventions, as well as statement labeling conventions. Each module must have a single entry and single exit, and no self-modification

* We define a module as the smallest unit of program code that can be compiled, loaded, and invoked by other units. Examples are procedures and subroutines. Hughes-Fullerton modules do not exceed a median of 25 executable HOL statements.

of statements during execution is allowed. Most recent air defense applications have been coded in the Jovial high order language, and the direct code option has been limited to special timing situations such as the online performance monitoring function. (This function periodically checks the system status, and cannot interfere with the application operation cycle.)

Parameter and Assembly Testing

Parameter and assembly tests provide for the testing of specific modules or groups of modules in preparation for integrating them into the system master version. These tests emphasize the internal processing of modules and are performed by the programmer who coded the modules. The main objective of parameter and assembly testing is to ensure that the modules under consideration are reasonably complete before further testing on a broader scale, and that each module or group of modules functions properly in isolation. Informal test procedures and reports are generated by the programmer and approved by the team leader.

Software Integration

The software integration activity is an orderly sequence of putting modules together to perform system functions in accordance with an integration or build plan. This activity emphasizes interfaces between modules, and ensures that modules will function properly in the latest system configuration. Some degree of testing must be performed in integration to provide confidence that a complete string of software operates properly, but not necessarily that the entire system operates correctly.

Independent Testing

The independent tests validate that the performance specifications are implemented properly. The testing is performed by a test group that is organizationally independent from the personnel involved in the software design and coding activities. Test plans and detailed test procedures are written to validate each requirement (i.e., "shall" statements of the functional (Part I) specification).

System Testing

System tests are formal acceptance demonstrations, which execute the portions of the software functions that are mutually agreed upon by the contractor and customer. The instrument for contractor/customer agreement is the test procedures, which are written against the system specification. These tests are also written to the specification "shall" level, are configured by CPCI, conducted by the lead systems engineering group, and formally witnessed and approved by the customer.

Installation Testing

Installation testing or field testing is essentially a replay of the system tests, with the addition of live tests that cannot be accommodated in plant. They are performed at the site of operation. The final system configuration is installed at this time, and live or actual inputs are used. Further operational testing is the responsibility of the customer with the support of the developer.

LESSONS LEARNED ON A RECENT PROJECT

Project Profile

Hughes-Fullerton has recently completed the development of software for a complex air defense system. Seven regional control centers supported by 86 sensor sites provide the command, control, communications, and surveillance functions for this system. The system provides for the transfer of sensor data from the sites to the regional control centers, the lateral-tell of track and status information between centers, and the forward-tell of all information from the centers to a central operations center. The system is capable of operating in standard and degraded modes, and can provide backup capability for interfacing systems. Nearly 30 positional consoles and 10 remote access terminals support the operation of each regional control center.

The embedded software is configured in seven CPCIs and totals nearly 6,000 modules which are coded in Jovial (J3). There were approximately 1,000 software changes during development that were the result of changes to the requirements. The changes in requirements included both clarifications and enhancements. Since the software has been placed under configuration control, 6,561 actual errors have been detected.

The development was performed in two major phases: design verification (DVP) and implementation (IP). At the peak of IP over 100 persons worked on software development. Approximately 40 percent of the software was "lifted" from previous air defense projects. Micro-phases completed within the IP were requirements analysis, design, coding, parameter and assembly test, integration, independent test, and system test. The project is now in the installation phase which includes on-site verification (OSV) testing for each of the regional control centers in the surveillance network. One center has successfully completed OSV testing and is operational.

Overview of Testing Activity

In general this air defense system development followed the Hughes-Fullerton standard procedures for software T&E. One exception was the omission of parameter and assembly testing for those clusters of modules which were lifted from existing Hughes systems. Some statistics about the independent testing activity exemplify the size of the effort involved in the T&E activity. There were nine software test engineers, including a team leader, assigned to the independent test team. A total of 143 test procedures with 14,277 test steps were generated and conducted for the seven CPCIs. The team expended 30,332 manhours over 32 calendar months in performing the independent test activity. The distribution of effort for detailed activities was: test plan generation (15%), test procedure generation (35%), and test conduct and analysis (50%).

Problems Encountered During T&E

Several lessons have been learned in this T&E effort which can contribute to more effective efforts in future software projects: One stemmed from the decision to start the next test phase based on schedule milestones rather than on current test phase status. These decisions placed too much of an extra burden on independent testing. Eventually the test effort was stabilized by the group performing independent testing, but much too late in the development cycle. Another lesson originated from the lack of a consensus philosophy on what constitutes a good test approach. The verification of the requirement specifications for consistency and testability was not emphasized sufficiently, nor was the planning for the software integration activity. A more systematic approach to these two phases would certainly have uncovered many errors earlier in the development.

Requirements Analysis. Of the 11,152 "shall" statements and 2,445 pages in the project's software requirements (Part I) specifications, there were a number of changes -- many coming late in the development cycle.

An up-front problem was the requisite quality of the software requirements specifications. Many requirements were not consistent, clear, or complete; therefore, at times it was difficult to map an individual requirement to a test procedure. Some software engineers viewed the requirements specification as a sacrosanct baseline (i.e., they would not or could not change it) and many problems only surfaced as obvious difficulties arose during design, code, and test.

Parameter and Assembly Testing. Insufficient records were kept during parameter and assembly testing, therefore the effectiveness of such tests on the project are unknown. Under the rush to adhere to schedules SED practices for parameter and assembly test were not followed. For example, modules and assemblies were frequently "promoted" to integration and even to independent testing with very little or no unit testing. When parameter and assembly tests did occur, sometimes there was no evidence of formal written objectives or certification.

Integration Activity. It is noteworthy that approximately 40 percent of the total errors detected after the software was placed under configuration control were encountered during software integration. Software integration was performed by a software integration coordinator with the aid of an assistant and the services of the project configuration control librarian. It is interesting that 48 percent of the errors encountered during integration were caused by the Logic category (e.g., omitted or out-of-sequence logic) and only 15 percent were caused by the Interface category (e.g., inconsistent call parameters). Intuitively, one would expect that Interface errors would be prevalent during software integration. A possible explanation for these results is that some of the software modules were not ready for integration testing.

Independent Testing. Twenty-five percent of the errors detected during testing were found in independent testing. The preparation of independent tests was time consuming. In addition, the many changes to the requirements and in turn to the design and code, as well as expected corrections to the code, required extra effort in rewriting independent test procedures and in retesting.

System Testing. Twenty percent of the errors were found during system testing. A common cause of errors encountered during this phase was the regression error caused by an incorrect modification.

On-Site Testing. Twelve percent of the errors were detected during on-site verification tests during the installation phase. Although this distribution is acceptable, we would prefer that the distribution reflect the earlier detection of errors. The multi-installation at the various sites contributed somewhat to the number of errors - primarily owing to differences in adaptation data definitions and to intersite functions, such as lateral-tell, which were not completely simulated at the Fullerton test bed. Analysis of causes of errors during onsite testing reveals a predominance of the omitted or incomplete logic category. This finding is in agreement with the study by Glass(12) on persistent errors found during software development on the AWACS project.

Recommended Solutions

On future projects, management procedures need to be devised to ensure that the following minimum test and evaluation standards are followed:

- Use of checklists to verify all specifications
- Transition between all development phases based on accountable completion criteria
- Audits for all source code updates
- Identification and certification of test tools for early use in the development cycle
- Documentation, approval, and certification of parameter and assembly tests

OTHER RELATED EXPERIENCE

Use of Simulation Drivers

Types of Simulation. Several kinds of simulation have been and are being used in the evaluation and testing of large-scale systems. Analytical simulations are used to study mathematical properties, interface simulations are used to simulate the actions of another system, operator, etc., and environment simulations are used to mimic the external or real-world in which the system is to operate.

Analytical simulations are used to investigate everything from satellite orbit controls to digital filter design for signal processors. In addition, discrete event simulations have been used to study computer loads, input-output queuing, device utilization, communication network congestion, etc. Such simulations usually support requirements definitions and top-level software design but rarely are used for testing of completed products.

Interface simulators have also been used in the development and low-level testing of software applications. Many of these can be thought of as "drivers". These are usually used as a substitute for unavailable devices, operators, etc. An interface simulator can be used to simulate the actions of a display console and its operator. The success of these simulators depends on their fidelity, responsiveness, and completeness.

Interface simulators have been used for limited acceptance testing and certification tools when interfacing with the real interface is unavailable, or too costly. When interface simulators are used for this purpose, they should be developed by an independent organization but most often they are not. They should also undergo the same formal specification and documentation as the software under test.

Several examples of each of these types of simulation can be shown; however, a detailed examination of one environment simulator will provide more insight into the capabilities, implementation, usage, and deficiencies of simulation.

Application of Environment Simulator. Environment simulators are used to represent the external or real-world situation. These are used with the equipment and software in as close to an operational configuration as possible. In many cases, these simulators provide the kind of outputs that would be generated by a sensor. In other cases, these simulators generate physical inputs into the sensor. The latter are much more expensive but provide for somewhat more realism, particularly if the sensor itself contains software which needs to be tested. In either case, an off-line program is generally used to calculate non real-time parameters and other scenario data. This data is then used by the real-time simulation program to drive the system under test.

Environment simulators are not only useful for large-scale systems, but also for medium-sized systems which may be too expensive to systematically and rigorously test in a real environment. Environment simulators can be designed for repeatability, which allows for testing to find anomalies and for retesting after changes are made. This repeatability also provides a convenient tool for systematic regression testing following sizable revisions and modifications to the software system. Repeatability is a virtual impossibility in the real-world.

Environment simulators also provide a vehicle for certification or acceptance testing of a software system before it undergoes operational testing. This implies that the environment simulator should possess a high degree of fidelity, operate in real-time, and be very complete in all details. All of these requirements mean that environment simulators can be quite expensive. These simulators should also undergo the same formality of specification as the software under test.

Example of Environment Simulation. A recent project illustrates what can be achieved with an environment simulator. The system in question is a navigation and location system that consists of a master station and several hundred user units. An alternate master station provides for back-up and continuity of operation. Operations can be conducted with adjacent master stations, which allows user units to range over a wide area.

Each master station consists of a communications unit, a display console, a page printer, a cartridge magnetic tape unit, and three computers -- one providing network control and communication, one providing unit position location and tracking, and one providing display and data management services.

Each user unit may be programmed by the network control computer to participate in one or two relay assignments. Each assignment provides for cooperative time-of-arrival measurements and the relaying of messages to and from the master station. The time-of-arrival measurements are used by the position location and tracking computer to determine range between user units, and by triangulation, the location of those user units. In addition, each user unit may be given assignments to make passive time-of-arrival measurements from other active relaying units. These measurements provide additional range measurements for position location and tracking and also provide "who can hear whom" data for the automatic network assignment algorithms in the network control computer.

The contract for this engineering development model called for a limited number of user units to be built -- not enough to test the full capability of the system. To test the full capability, the sponsors required and funded the development of a real-time environment simulator. This simulator was to provide for the direct simulation of a network of several hundred user units. The simulator was to accept all user unit assignments and messages from the network control computer, simulate all relay processing, calculate the appropriate time-of-arrival values and simulate nearly all of the passive time-of-arrival measurements. The results of all of this activity were to be returned to the network control computer in a form completely compatible with communications relay unit. All of this activity was to be done for several user units each 250 milliseconds of real-time, the same interval the real network operates in.

Implementation. The simulation system was divided into two parts, an off-line function and a real-time on-line function. The initial goal was to have all functions on-line. The computational load, however, was too great for the available computer resource. The computations included simulating user unit motion, probability of communications based on transmit power and propagation loss over terrain, time-of-arrival measurements, and terrain elevation determination.

The off-line scenario generation process consists of five steps or parts. The first step does an error detection process of the set-up data. A digitized terrain map is read and re-digitized to reduce computational loads. The second step preprocesses motion data and generates a file of unit positions as function of time. The third step generates control information such as units turning on or off. The fourth step calculates propagation loss data between units. The fifth and last step calculates the probability of communication matrix using results of the previous steps onto a magnetic tape for use by the on-line simulation program.

Some of these steps could be combined. The long running time for generating large scenarios (20 hours for several hundred units operating for 30 minutes) suggested that checkpoints would be needed. So, rather than having classical checkpoints, the job was partitioned at logical breakpoints.

Usage. The off-line system has been used to generate several scenarios. The first of these was a specialized idealized scenario to form the basis for a certification buy-off test performed for the systems engineering organization. This test stressed the ability of the system to locate and track a network of user units operating in a sensitive geometric arrangement.

A second scenario was designed to test a full load of several hundred user units consisting of man-pack units, surface vehicles, helicopters, and jet aircraft all operating over an extended range in rough terrain. This scenario was designed to run over 30 minutes. This test was also used to perform stress testing, in that more units were introduced into the system than the system had internal capacity for. Stress testing of this type is very difficult to control and achieve in real-world situations short of full-scale exercises or combat.

Other scenarios were generated to fine tune system parameters for tracking accuracy improvements. Special scenarios were used to confirm software correctness by comparing analytical simulations with environment simulations.

Most of these scenarios were re-run at major update points in the development cycle as regression tests. In more than one instance, significant errors were detected and removed before the software was released. Both the developers and system engineers feel today that this system simply could not have been developed to its present high state without the real-time environment simulator.

Critique. The results have been quite useful. Much of this success is due to the inherent discrete nature of the measurement process. The discontinuous nature of the probability of communication matrix is perhaps the weakest feature. The repeatability of a scenario however has proved to be a very useful capability. This is not to imply that the scenarios are repeatable in microscopic detail; they are not. The "dither" that results because of variations in timing has been useful in exposing certain sequencing problems.

Much of the success of this simulation can be traced to the basic nature of the system. The results of a position location system can be displayed on the display console. Any deviations can be quickly recognized by eye; very little post-run data reduction is needed. This characteristic is not true for many systems.

For all of the success of this environment simulator, it is not without deficiencies. The set-up procedures are tedious and voluminous, especially for a large number of units operating over an extended period. The limited off-line computational resource forced a coarse terrain digitization. This means the probability of communication matrix is quite crude. The inability to modify motion data on-line limits testing of certain geometries. The on-line memory capacity limits the kinds and amounts of digital messages which can be generated.

Regression Testing

Myers[19] defines regression testing as "...that testing that is performed after making a functional improvement or repair to the program." Changes can cause previously correct programs to regress. He goes on to say that regression testing is important because changes and repairs are error-prone operations. It is important for other reasons as well. Building software through incremental development may cause failures due to unusual coupling in external, real-world processes. The software is correct according to the specifications, but the specifications are flawed.

In any case, re-running test cases which were previously correct is good insurance. Generally no one test will be adequate and a variety of tests should be run. Some authors[9] suggest that a complex test should be run when changes are made to a complete program and then smaller specialized tests run to localize any detected errors. This is in reverse order to the kinds of tests normally used during development. Experience on the navigation system described above showed that such complex tests could be and were run during development and not only as a culmination test.

Regression tests should be easy to run, should not involve extensive manpower or set-up time, and results should be easy to analyze. If these tests violate any of these criteria they probably won't be run with enough frequency to be of any value. The argument that regression testing is unnecessary in an incremental development process, because of the extensive amount of redundant testing done at each step, is a faulty argument. Most of the "redundant" testing at each increment generally focuses on newly added capability and does not re-test existing or previously tested programs. Our experience shows that regression testing can be useful.

Experience in Independent Verification and Validation (IV&V)

Hughes-Fullerton has had very little experience with independent verification and validation either as an IV&V contractor or as a contractor with projects being subjected to an IV&V effort. This situation is changing, however. The sponsor of one current project is employing a full-time IV&V contractor. Other anticipated contracts will also employ an IV&V contractor. One previous sponsor used an IV&V contractor for one special assignment.[13]

Despite this limited experience, some observations can be made. It appears the IV&V is expensive. This is admitted even by those who do much of this work.[22] Whether or not it is cost-effective is not clear. The developer can be expected to find about 10 times as many errors as the IV&V contractor (1:1000[22] versus 1:100[3]). A "shotgun" approach is probably not cost-effective nor particularly productive. The IV&V tasks should focus on validation of user needs and verification of the design and implementation. These efforts should address products and not processes. To minimize biases, the IV&V contractor should be precluded from any follow-on contracts.

Endurance Testing (MIL-STD-1679)

Requirement. Software developed in accordance with MIL-STD-1679[5] is required to pass a continuous duration test. The specific length is not quantitatively defined except for systems which are intended to operate for more than one day, in which case, the minimum length of time is 25 hours. Folklore says that one author of this requirement had a program fail to keep the time-of-day correct after midnight. By extrapolation then, a system which is intended to operate continuously for more than a week should also be tested continuously for the intended period (month, year, etc.) just to be certain that time and dates are handled correctly. While the intent of this requirement is clear, it doesn't follow that a test has to be of that duration to meet the intent. Special purpose tests can be devised to show the correctness of handling special cases. A calendar routine for handling leap years is a good example: a four year test would be folly, not prove very much, and cost a good deal of money. The limit of such tests is easy to see.

MIL-STD-1679 states further that if the complexity of the system is such that the intent of the specification cannot be met, the duration of the test shall be extended. The authors of MIL-STD-1679 are very generous with others' money.

What then constitutes "endurance" testing and how is it best achieved? Endurance testing appears to take two forms. The first is looking for "reliability" or "robustness" though neither can be adequately defined. Historically some large-scale systems had failures due to some internal accretions which resulted in mysterious operations -- spurious bits being turned on, buffer overflows with light loads, etc. The failures appeared only after long running times, 40 to 60 hours typically.

The second form appears to look for latent errors in the software. Note carefully that this is different than reliability or robustness. This search for errors, however, is better done by specific tests which by their very nature will be of short duration. The ability to find latent errors with long running tests would require complex test sequences, variations, and people to perform them.

Experience. A look at some of our experience may be of interest.

A radar system was required to perform 100 hours of error-free software test. This requirement was imposed because the software was assumed to contribute to system MTBF, and a 100-hour error-free run would "prove" that the software met its reliability contribution. The system, however, operates for 23.5 hours with one-half hour down for maintenance each day. Finally, the basic cycle of software operation is only a few seconds, that is, the software repeats a basic cycle every few seconds with almost no variation, particularly during the 100-hour test.

Another system is intended to operate for many days at a time. The endurance test requires 25 hours of error-free operation using a fixed "full" load with little variation. Pure MIL-STD-1679. The basic cycle or period of this system is about 12 seconds with a major cycle of 3 hours. A separate test of a few hours duration is used to test modifications.

Another radar/weapon system intended to operate continuously is required to pass a 50-hour test consisting of 2 hours maintenance, 22 hours operational software, 2 hours maintenance, 22 hours operational, followed by another 2-hour maintenance run. The basic cycle of this system is 2 seconds with a major cycle of 6 seconds. In practice, this system could be expected to operate for more than 24 hours upon occasion.

Another radar system intended to operate continuously is required to pass a 60-hour error-free test run. This test changes operating modes and loads at various points in the run. The basic cycle of this system is a few seconds.

Another radar system intended to operate continuously does not have a separate endurance test. The system has a reliability test in which software failures count as system failures. This test is run continually over several months and includes both real-time and maintenance software operations. The basic cycle of the real-time software is a few seconds.

Summary. As can be seen from these examples, several different systems approach the endurance testing in completely different ways and for apparently different reasons. A consensus across many of these projects is that very few if any errors are found during these endurance tests. It does appear that testing a program at intervals greatly beyond the basic operating cycle is probably a waste of money and time. It would appear that running many shorter tests of greater variety would be more productive and useful.

RECOMMENDATIONS FOR RESEARCH AND DEVELOPMENT

We suggest that the test and evaluation of large embedded software systems be thought of as an ongoing experiment that can be reiterated until the current system is fully mature. We recommend the development of candidate strategies for planning and monitoring the software integration process. The aims of these strategies should be to ensure that the overall process is sensitive to the structure of the particular software system being developed and that it allows for continual feedback of the status of the activity. Incremental builds, threads, and iterative enhancement of a skeletal function are examples of such strategies. The use of separate design verification and implementation phases, as they were used on our example air defense system project, qualifies as a high-level build approach. Prototyping is similar, except that the product is normally a throwaway.

An internal study recommended that the most fruitful areas of research within the SED environment are associated with the ability to automatically verify requirements, the implementation of requirement-tracing aids, and automatic generation of test cases. The study recommends the following three areas for improvement of the SED test environment:

- Development of better original requirements and design specifications
- Placement of those specifications in a machine-manipulatable form to allow automatic verification procedures
- Implementation of an effective requirement-tracing tool

Since independent testing is a very stable area in the SED software development cycle, it would be possible to evaluate an applicable R&D effort to enhance the current SED methodology. A great deal of effort is now expended in preparing and conducting independent tests. These tests are the last significant effort performed on software prior to performing customer-viewed system testing, and constitute a major product IV&V test. Consequently, it can be envisioned that a technology upgrade in independent test methodology could replace existing, largely manual, efforts with an automated and improved product evaluation as well as provide a substantial cost reduction.

ABOUT THE AUTHORS

MR. JOHN BOWEN is a systems engineer in the Research and Analysis group of the Software Engineering Division at Hughes-Fullerton, where he is performing studies in software quality metrics. He is currently collecting error data from a large-scale air defense system as part of an RADC software reliability model study. He recently completed a software reliability requirements study task for the MILSTAR terminal for USAF/ESD and participated in the design of semi-automatic error seeding technique for the evaluation of V&V tool sets for NASA-Ames. He has also investigated proposed software complexity metrics such as cyclomatic number, module size, and software science measures as reliability predictors. Mr. Bowen conducted software integration and system tests for USAF command and control systems 407L, 473L, and SAGE as well as for the DoD Damage Assessment Center (DODAC) system. He is the author of the Computer magazine article, "A Survey of Standards and Proposed Metrics for Measuring Software Quality Testing" which has been selected twice for reprinting by the IEEE.

MR. MARION MOON is the Chief Scientist for the Software Programs Laboratory at Hughes-Fullerton, which develops embedded software systems for Hughes-Fullerton's radar, sonar, and communication product lines. These include shipboard radar, sonar, and torpedo control systems, artillery and mortar firefinder radars, field-deployed position location systems, and airborne-controlled communication net management systems. As Chief Scientist Mr. Moon is the principal technical consultant for the some 15 projects currently being developed in the Laboratory. Previously he held software technical director positions on the US Navy Advanced Capability MK 48 Torpedo (ADCAP) program and the US Army Position Locating and Reporting System (PLRS) program. He also headed the Advanced Techniques Section where new software design methodologies were perfected and several computer system simulation models were developed for use on Hughes software projects. While in the US Army, Mr. Moon tested missile fire control and air defense systems. He holds a BSEE from the University of Kansas and has done graduate work in computer science and engineering at UCLA.

REFERENCES AND SELECTED BIBLIOGRAPHY

1. Adrion, W.R., et al, Validation, Verification, and Testing of Computer Software, Computing Surveys, vol 14, no 2, pp 159-192 (Jun 1982)
2. Basili, V.R. and Turner, A.J., Iterative Enhancement: A Practical Technique for Software Development, IEEE Transactions on SE, vol SE-1, no 1 (Dec 1975)
3. Boehm, B.W., Software Engineering Economics, Prentice-Hall (1981)
4. Bowen, J.B., A Survey of Standards and Proposed Metrics for Software Quality Testing, Computer, pp 37-42 (Aug 1979)
5. Chief of Naval Material, Military Standard for Weapon System Software Development, MIL-STD-1679 (Navy), (Dec 1978)
6. Davis, C.G., The Testing of Large, Real Time Software Systems, Proceedings of the 7th Texas Conference on Computing Systems, 4/25-35 (1978) (Apr 1982)
7. DeMillo, R.A., Software Test and Evaluation Project (STEP), draft report (Sep 1982)
8. Deutsch, M.S., Software Verification and Validation -- Realistic Project Approaches, Prentice-Hall (1982)
9. Deutsch, M.S., Software Project Verification and Validation, Computer, pp 54-70 (1981)
10. DoD/USDR&E, Department of Defense Directive 5000.3, Test and Evaluation (Dec 1979)
11. Fujii, M.S., A Comparison of Software Assurance Methods, ACM Software Quality Assurance Workshop, San Diego (Nov 1978)
12. Glass, R.L., Persistent Software Errors, IEEE Transactions on S.E., vol SE-7, no. 2 (Mar 1981)
13. Hamilton, M. and Zeldin, S., The Application of HOS to PLRS, final report HOS-TR-12 (Nov 1977)
14. Kacik, P.J., An Example of Software Quality Assurance Techniques Used in a Successful Large Scale Development, ACM Software Quality Assurance Workshop, San Diego, pp 181-186 (Nov 1978)

15. Kopetz, H., Software Reliability, Chapter 7 -- Verification of Software, Springer-Verlag (1979)

16. Lewis, R.O., Software Verification and Validation (V&V), Software Quality Management, Petrocelli, pp 235-253 (1979)

17. Nelson, J.G., Software Testing in Computer-Driven Systems, Software Quality Management, Petrocelli, pp 255-274 (1979)

18. Macina, A.J., Independent Verification and Validation Testing of the Space Shuttle Primary Flight Software System, NSIA/AID/SD/NASA Mission Assurance Conference (Apr 1980)

19. Myers, G.J., The Art of Software Testing, Wiley (1979)

20. Page, J., Methodology Evaluation: Effects of Independent Verification on One Class of Application, Proceedings of the Sixth Annual Software Engineering Workshop (Dec 1981)

21. Powell, P.B., Software Validation, Verification, and Testing Technique and Tool Reference Guide, NBS Special Publication 500-93 (Sep 1982)

22. Radatz, J.W., Analysis of IV&V Data, RADC-TR-81-145 (Jun 1981)

23. Rustin, R., Debugging Techniques in Large Systems, Courant Computer Science Symposium 1, Prentice-Hall (1971)

24. Smith, M.K. and Hudson, D.R., A Survey of Software Validation, Verification, and Testing Standards and Practices at Selective Sites, NBSIR 82-2482 (Apr 1982)

25. Watkins, M.L., A Technique for Testing Command and Control Software, Communication of the ACM, vol 24(4), (Apr 1982)

THE ECONOMICS OF SOFTWARE TESTING - AN INTRODUCTION

Raymond J. Rubey
SofTech, Inc.
Dayton, Ohio

ABSTRACT

Software testing, which has the discovery and elimination of errors as its objective, has a cost that depends on the number of tests. Errors that go undiscovered by testing cause costs to be incurred, both to correct the errors and because the software fails to perform its intended functions. More complete knowledge regarding the cost of tests and the cost incurred due to errors would enable determination of the economically efficient amount of testing that should be done. This amount would be achieved when the errors eliminated by further testing have a total cost less than the cost of additional tests needed to discover those errors. This paper discusses the factors that determine test cost and error-associated cost. These test cost factors are related to the likelihood that errors will be discovered when particular classes of tests are executed.

INTRODUCTION

An often quoted statement asserts that "testing can only show the presence of errors but never their absence." This statement is based on the fact that it is impossible to test all alternative input values, input combinations, permutations of program decisions, etc. for any non-trivial program. In actual practice, our confidence that a computer program is free from error is achieved by testing, in spite of the fact that this testing is imperfect. Fortunately most applications do not require absolute assurance that the computer program is free from any error; rather it is sufficient that the program is very nearly correct. If we are satisfied with less than perfection, we can both afford and construct the desired computer programs. In actual practice, we must decide when sufficient testing has been done. One of the best criterion to use is the fulfillment of a comprehensive test plan. Unfortunately, the budget often determines the test duration; when the software development money has been spent, the testing is stopped. This is the worst economic criterion; we must have better ones. The following sections provide some guidelines and concepts for the development of such criteria.

First we will begin with a global look at the economic aspects of testing. We can define an error cost for a computer program containing undetected errors (i.e., residual errors) as:

$$\text{Total Error Cost} = \sum_{r=1}^{R} (\text{cost of error } r) \times (\text{probability of error } r \text{ occurring})$$

where R = number of residual errors.

We can also define the cost of testing as follows:

$$\text{Testing Cost} = \sum_{t=1}^{T} (\text{cost of executing test } t) + (\text{cost of defining test } t)$$

where t = number of tests that are run.

When the cost of testing to find the residual errors is greater than the cost that would be incurred if these errors cause the program to fail, it is no longer economic to continue testing. Unfortunately we often know very little about such costs. We can and should make some effort to improve our knowledge and estimation abilities however. Let us begin by concentrating on individual errors rather than on the errors in total.

INDIVIDUAL ERROR COST

Each individual error in a program has a cost and it is too much of an approximation to consider these costs as equal. Simply stated some errors cost more than others. We can divide the error cost into three components as follows:

$$\text{Cost of Error R} = \text{"Lost Function" Cost} + \text{Correction Cost} + \text{Correction Distribution Cost}$$

"Lost Function" Cost

A computer program is written to accomplish functions for a user; these functions can be as diverse as that of controlling a tuning of an automobile radio to that of guiding a space booster. When in a particular situation the program fails to do a needed function because of an error, then a cost is incurred. This cost might be trivial (e.g., because the radio has been tuned to the wrong station) or may be enormous (e.g., because the space booster has failed to achieve orbit). This "lost function" cost is clearly a function of the type of application. If there is only a single user of the program, the "lost function" costs might be limited. However if there are many users (e.g., we have sold a lot of digitally tuned radios) the sum of all the individually small costs may be substantial. It would seem there are an increasing number of applications with substantial "lost function" costs. In these applications the total error cost due to residual errors may be large even though the probability of these errors is low; thus considerable testing cost is justified. Software development managers and especially users should be aware of the range of "lost function" costs for their application. Of course in most situations, the "lost function" cost is only incurred when a program is put to use and need not be considered as being incurred during development.

It might be argued that software developers need not pay attention to "lost function" costs since, unless warranties or other liability measures are invoked, the "lost function" costs are incurred by the user. Such an attitude would quickly reduce the number of users willing to hazard further ventures with the short sighted developer.

Correction Cost

When a software error occurs, we usually attempt to correct or modify the program so that the error will not occur again. A cost is incurred in making the correction. The size of this cost is, in large part, dependent on the time between when the error is made and when it is detected. If the error is made on one day and detected the very next day, it will usually be cheap to correct because very little work has been expended during the short interval. This is a reason for the advantages claimed by various top-down and structured development approaches; testing of parts can be done before the whole has been coded. If there is a long period between when an error is made (e.g., in software requirements definition) and when it is detected (e.g., in the very last stages of testing before use) then the correction cost is usually high (1). Many past studies and reports indicate that the following relationship holds:

$$\text{Relative Correction Cost} = 10^{2D}$$

where D = fraction of the software development that has elapsed between error creation and error detection.

Any activity, such as design reviews, walkthroughs, code reading and programming standards, that reduces N will reduce correction cost. Correction cost, unlike "lost function" cost is fairly independent of application. Better software development activities help reduce error costs for everyone.

Distribution Cost

When an error is detected and corrected in a program, all users of that program must be given the corrected version of the program if they are not to continue to be subject to the "lost function" cost of the error. If we are still developing the program, the distribution cost is usually negligible. For our space booster software application the distribution cost is very small since the program has a single user. However, for our digitally tuned radio and for similar applications, the distribution cost may be substantial. Indeed, as developers we may be willing to incur considerable user discontent to avoid high distribution costs.

TESTING COST

Given this brief look at the cost incurred when an error occurs, let us examine what it costs to do the testing. As indicated in the introduction, there is a cost associated with actually executing a test and a cost associated with defining a test. The cost of executing a test is largely a function of the resources (e.g., execution time, memory, input/output devices, etc.) required by the program being tested. These resources are partly a function of the software application and its environment and partly a function of the software's structure.

Before we can execute a test however we must define the test itself. Test definition requires that we specify the inputs for the test, the expected output of the test, and the method by which the test outputs will be obtained and analyzed. Usually the cost of defining a test is much greater than the cost to actually execute the test. The cost of defining a test is a function of the number of inputs, number of outputs, number of unique input/output combinations, the number of equivalence classes, the data flow complexity and control flow complexity of the software under test. These first two factors may be determined by simple counting. The input/output combination factor may be determined by counting the number of rules in a decision table obtained from a cause/effect graph mapping of the input/output relationships. (2) The number of equivalence classes may be determined by a detailed examination of the software's requirements. (2) The data flow and control flow complexity may be determined by metrics, such as the McCabe number, for the corresponding flow graphs. (3)

It is usually more difficult to define the first test to be run than it is to define subsequent tests because these subsequent tests are only variations of the first test. Thus if we know what it costs to define the first test, as a function of the factors outlined in the preceding paragraph, we can estimate the cost of a subsequent test by calculating the Difference Factor for that subsequent test. The Difference Factor for a test would be 1 if the inputs, outputs, input/output combination, equivalence class, data flow paths and control flow paths were all changed for the new test as compared to all previous tests. The Difference Factor will be 0 if the inputs, outputs, input/ouput combinations, equivalence class, data flow and control flow of a test is identical to any previous tests. Usually a test will have a Difference Factor between 0 and 1 and the Difference Factor is calculated by the following formula:

$$\text{Difference Factor} = \frac{\text{Number of actual differences}}{\text{Number of potential differences}}$$

Obviously it costs nothing to define a test with a Difference Factor of 0; also it is obvious that such a test is very unlikely to reveal a previously unknown error. The best test is one that reveals a previously unknown error. We could define the "goodness" or utility of a test by the following formula:

$$\text{Test Utility} = \sum_{r=1}^{R} (\text{Probability that error } r \text{ is detected}) \times (\text{Cost of error } r)$$

where R = total number of residual errors.

The probability that a previously unknown error is revealed by a new test is proportional to the Difference Factor of that test. If the Difference Factor is large, significantly different control and logic flows will be exercised, previously untested input/output combinations will be demonstrated, and different numeric output values will be obtained. The error is thus more likely to be revealed than if the data and control flows and the inputs and outputs were almost the same as a previous test. Briefly, good tests cost more than poor tests.

The Difference Factor can be used to measure the completeness of a collection of tests. If it is not possible to define a test with a Difference Factor greater than 0, then obviously all combinations and alternatives have been tried. However, if tests with Difference Factors of .5 or more can easily be defined, then more testing is required.

Given the above understanding, two testing strategies are possible. One strategy would be to execute tests with large Difference Factors early during the test effort. This would make the testing more expensive but would tend to uncover errors earlier. This early detection is desirable because of the previously discussed "Correction Cost" effect. However an alternate strategy would be to conduct an orderly sequence of tests, in which each test differed only slightly from the preceding test. Test costs would be less although the cost to correct mav be increased because errors might be found later than was the case for the first strategy. The most effective strategy is not apparent at this time.

SOFTWARE STRUCTURE IMPLICATIONS

Up to this stage we have considered the software test as a monolithic entity. The software can and should have a structure. The testing effort in turn can and should take advantage of this structure to reduce the cost of the complete test activity. An effective software structure is hierarchical and this hierarchy is composed of modules that each:

1) Have a single entry.
2) Have a single exit.
3) Have explicitly defined inputs and outputs.

The higher level modules, that invoke or call lower level modules, contain alternative control paths. Thus, although the structure itself is fixed, the sequence of module invocations for any particular execution of the program will be different.

For a large, complex program, it will be expensive to define tests with Difference Factors that satisfy all of the criteria defined in the previous section. A better approach would be to test the modules individually, combine them into an integrated program and test the interfaces between modules.

If a one-step integration approach is taken the test cost is:

$$\text{System Test Cost} = \sum_{n=1}^{J} \text{Cost of testing module } n + \sum_{m=1}^{K} \text{Cost of testing interface } m$$

where J = number of modules
and K = number of intermodule interfaces.

The cost of testing the individual modules was discussed in the preceding section. The cost of testing interfaces is a function of the number of inputs and outputs shared between each pair of modules, the number of unique modules invocations and the number of structural control paths. An interface testing Difference Factor can be defined in a manner analogous to the factor defined for an individual module in the preceding section. In any but the smallest programs (e.g., <10 modules) testing by modules first and then integrated testing is more economical than monolithic testing of the total program.

FUTURE DIRECTIONS

The above discussion has outlined only a few of the economic considerations that are important in software testing. Important directions for future work include the effectiveness and cost of test tools and facilities, the need for and cost of regression testing, and the incorporation of probabalistic software reliability models. One problem requiring continual attention is the definition of terms used in and associated with testing. It does little good to worry about the cost of testing or of errors when we do not have common agreement on the meaning of the words testing and error. Is debugging different from testing? If it is, when does debugging stop and testing begin? To what extent is the programs failure to detect a user's erroneous input an error? Answers to questions such as these are an important step in understanding the economics of software testing. The simple and informal discussion of this paper has outlined only a few of many possibilities in this important area.

REFERENCES

1. Boehm, Barry W., Software Engineering Economics, Prentice-Hall, 1981.

2. Myers, Glenford J., The Art of Software Testing John Wiley and Sons, 1979.

3. Mcabe, T.J., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2(4) 1976.

ABOUT THE AUTHOR

Raymond J. Rubey is Technical Director of SofTech's Dayton office. He recently was Visiting Professor of Engineering at the Air Force Institute of Technology where he prepared and taught graduate classes in software engineering, reliability and management. Mr. Rubey has managed SofTech's verification and validation contracts for both the Precision Location and Strike System (PLSS) and the Electronically Agile Radar (EAR) software. Prior to joing SofTech, he was with Logicon where he managed a wide range of projects, including the verification and validation of the B-1 avionics software, the development of test tools and simulators, the definition of quantitative measures of software quality and the development of avionics programming languages and compilers. Mr. Rubey has prepared and presented seminars on software quality assurance and testing throughout the United States, Europe and Japan. He has a Bachelors and Masters degree in Engineering from U.C.L.A.

PROGRAMMING LANGUAGES, TESTING, AND REUSABILITY.*

Peter Wegner, Brown University
Providence, RI, 02912

Abstract: We present a variety of ideas and opinions on increasing the productivity and reliability of software. Interface technology and knowledge engineering are suggested as primary themes for research and development in the 1980s and 1990s. The impact of programming environments and powerful personal computers on testing and management technology is considered. The evolution of programming languages and the relation of Ada to its predecessors is reviewed. The notion of "capital" and "capital-intensive" are defined in terms of reusability of resources, and the contribution of Ada to the development of capital-intensive software technology is examined. The relation between maintainability, enhancement, and evolution of systems is discussed. The relation between knowledge engineering and software technology is explored. Coordinated approaches to making software technology more capital-intensive, such as the Japanese fifth-generation computer proposal and the DOD software initiative are examined in the conclusion.

1. *Interface Technology and Knowledge Engineering*

The evolution of computer science in the second half of the 20th century may, as a first approximation, be characterized by the following phases:

1950s: Experimental systems (architectures, languages, applications)
1960s: Mathematical models (automata, formal languages, semantics)
1970s: Software engineering (life cycle, abstraction, methodology)
1980s: Interface technology (software, hardware, user interfaces)
1990s: Knowledge engineering (expert systems, education, visual programming)

This view is simplistic but nevertheless helpful. It suggests that computer science evolved from primarily experimental origins in the 1950s through a mathematical phase concerned with the modelling of the "phenomena" of computer science to an engineering phase concerned with cost-effective and reliable software construction. The engineering approach focussed on life-cycle methodology and abstraction in the 1970s. The focus in the 1980s appears to be on the development of interfaces for software components, hardware components, and users. It is predicted that, in the 1990s, a primary concern will be that of making interfaces more intelligent.

We are in the midst of a computer revolution that parallels the industrial revolution in the magnitude of its social and technological changes. Software and knowledge engineering play a role in the computer revolution similar to that played by traditional engineering in the industrial revolution. Interfaces play a key role in both industrial and information engineering but require more explicit, self-conscious definition because information structures are more abstract and less tangible than physical structures such as bridges and buildings. Information technology should make use of principles and techniques of traditional engineering where this is appropriate, but should adapt its techniques to the fact that there are differences as well as similarities between the products of information technology and traditional technology. These ideas are further discussed in [Wegner, 1982b].

Our view of interface technology and information engineering as primary themes (buzzwords) for the 1980s and 1990s is supported by the fact that fourth-generation graphics-based personal computers of the 1980s emphasize interface technology, while fifth generation computing systems proposed for the 1990s emphasize knowledge engineering. [Fifth, 1981].

Fourth generation computers have high-resolution graphical user interfaces with "windows" to represent a desktop with multiple ongoing activities. Visual representation of inputs, outputs, and intermediate states of a computation increases the bandwidth of the man-computer interface and provides both a better representation and greater control of the computing process.

Interface technology at the hardware and display levels is being matched by modular interface technology at the language-level in new languages like Smalltalk and Ada, and by "object-oriented" software-development methodologies that support interface technology at the level of applications. The 1980s are concerned with integrating interface technology at the level of hardware, software, and applications so that modular applications can be easily designed, easily mapped into programming language and hardware representations, and easily modified and tested.

Fifth-generation computing systems, which may become state of the art in the 1990s, will add intelligence to the high-bandwidth interface provided by fourth-generation computing systems. The Japanese fifth-generation computing proposal is a prototype for such systems. Its proposed hardware includes a database machine and a problem solving and inference machine. Its proposed system programming language is a logic programming language such as Prolog. Its software includes support for natural language and speech understanding and problem solving over a wide set of problem domains. The project includes not only technical goals such as increasing productivity and saving energy, but also social goals such as coping with an aging society. The project is regarded by some US researchers as overambitious. But it has a worthwhile set of goals which, even if they are not achieved in their entirety, can catalyze an integrated research effort that could give Japan a technological lead in developing computing systems for the 1990s.

Fourth- and fifth-generation computing technology emphasizes complementary aspects of information engineering. Fourth-generation computers emphasize engineering of the man-computer interface to increase the bandwidth of man-machine communication and the potential of man for assimilating knowledge. Fifth-generation computers emphasize engineering of the internal representation of knowledge for intelligent problem solving. They emphasize very high-level problem specification that avoids concern with the intermediate stages of problem solution, while fourth-generation systems allow flexible representation of and access to intermediate states of a computation so that "grass-roots" contact with the "operational semantics" of a computation can be maintained while thinking at a high level of abstraction.

The systematic exploration of substantive ideas in interface technology and knowledge engineering is beyond the scope of this paper. However, singling out these notions as major research and development themes in the remaining years of the 20th century has non-trivial implications. It suggests that programming languages like Ada be evaluated in terms of their module interfaces for abstraction and concurrency, and that

* This research was supported in part by ONR contract #N00014-78c-0656

machine architectures, distributed systems, personal computers, and testing methodologies be examined from this point of view.

2. *Impact of New Programming Environments*

The overall objective of software technology is to reduce the cost and increase the reliability of software. Realization of these goals requires a number of different, complementary activities.

(1) The development of better design, development, and maintenance methodologies.

(2) The construction of better testing, verification, and validation tools.

(3) Compile-time and execution-time redundancy for error detection and correction.

The primary goal of programming languages is to facilitate better programming methodology that reduces errors and makes programs easier to correct and to modify. Testing is facilitated by tools in the programming environment for module testing, simulation, and configuration management. In developing Ada it was realized very early that its success depended on the quality of the environment as well as the language. Ada aims to contribute to cheaper and more reliable software both at the language level through improved design methodology and at the environment level through integrated testing, simulation, and management techniques.

Testing in the narrow sense is concerned with determining whether selected inputs in a test data set yield desired outputs. In a broader sense testing should be concerned not only with correctness, but also with reliability, robustness, preformance, and utility [Good, 1979]. Moreover, testing should not be restricted to the "debugging and testing" phase of the software life cycle, but should be viewed as a continuing activity that is performed formally and informally as a part of every life cycle activity. This view is reflected in the following characterization of the software life cycle due to Barry Boehm [Boehm, 1981].

(1) **Feasibility:** Defining a preferred concept for the software product and determining its life-cycle feasibility and superiority to alternative concepts.

(2) **Requirements:** A complete, validated specification of the required functions, interfaces, and performance for the software product.

(3) **Product Design:** A complete, verified specification of the overall hardware-software architecture, control structure, and data structure for the product, along with such other necessary components as draft user manuals and test plans.

(4) **Detailed Design:** A complete, verified specification of the control structure, data structure, interface relations, sizing, key algorithms, and assumptions of each program component (routine with < 100 source instructions).

(5) **Coding:** A complete, verified set of program components.

(6) **Integration:** A properly functioning software product composed of the software components.

(7) **Implementation:** A fully functioning hardware-software system, including such objectives as program and data conversion, installation, and training.

(8) **Maintenance:** A fully functioning update of the hardware-software system. This subgoal is repeated for each update.

(9) **Phaseout:** A clean transition of the functions performed by the product to its successors (if any).

The above view of the life cycle as a sequence of distinct phases with explicit interfaces that serve as checkpoints for transmitting the project from one group of workers to the next was appropriate for batch processing systems and has served as a useful framework for the systematic exploration of mechanisms for improving software productivity and reliability. New programming methodologies will cause a radical shift in our way of doing business and will require a modified life cycle model that admits multiple passes to allow rapid prototyping, iterative feedback and enhancement between life cycle phases, and system controlled management, testing, and evaluation.

New programming environments will make communication among different groups working on a large project much easier by providing a system-wide mail system and access to a common data base containing the current system and its development history. Testing, verification, and validation may be performed more frequently than before using a time-stamped snapshot of a developing system without disrupting the progress of the development group. The system will be able to prompt the validation group on what is to be tested, and to prompt managers on configuration control activities.

Programming environments will not only improve the ease of communication but also its quality. High-resolution graphics will allow new representations of the system status that will provide better insight and understanding for managers, programmers, and testers. Research on "visual programming", which is concerned with the representation of intermediate system states so that the system comes to life as a dynamically evolving entity, will allow new forms of test and evaluation based on "observing" intermediate states of the subsystem being tested. Graphical interfaces will imbue information structures residing in a computer with a degree of reality approaching that of physical structures such as bridges and buildings, and will serve to reduce the conceptual gap between hardware and software engineering.

Syntax directed editing will aid in program development and guide the programmer in using good programming methodology. Graphics will aid in test and evaluation by providing graphical representations of the process of program execution and symbolic evaluation. Tools for functional and structured testing will be provided and will make use of graphical representations of both the program execution process and the space of data inputs. Finer control of both the program development and execution process will suggest an entirely new set of test and evaluation tools that make use of monitoring and prompting techniques and the techniques of visual programming. The new generation of test tools will be firmly integrated with compilers, debuggers, editors, and other tools of the environment.

Programming environments will contain educational aids to instruct programmers in the purposes of testing and the use of testing tools. The boundary between education, documentation, and productive use of software tools is fuzzy, since all three activities are concerned with the management of complexity. Educational display techniques will often find a use in providing additional insights for program development and testing purposes. Educational aids will be particularly important for large systems where testing requires an understanding not only of general testing principles but also of the details of the particular application.

The choice of test data and test cases is an art requiring expert knowledge in the domain of application and it is likely that expert systems for testing will be developed in broad domains of application. If the total cost of a system exceeds one billion dollars it may well be worth investing in expert systems that apply domain-specific knowledge not only to testing but *also* to other phases of the life cycle. The development of expert systems to assist in life cycle management and testing for broad application domains is an area that deserves further research.

Another important research area is the development of good general-purpose tools for the construction of testbeds, simulators, and rapid prototypes. Consideration of these issues is beyond the scope of this paper. However, we shall briefly illustrate the flavor of testing in specialized application domains by mentioning a specialized testing application in the domain of Ada, namely the Ada compiler validation project.

The Ada compiler validation project is perhaps the most comprehensive project for testing of compilers that has ever been undertaken and deserves to be studied as a well-documented example of a specialized testing system. The compiler validation aids produced by Softech as part of its compiler validation project include the following:

a) An implementors guide which "identifies common errors in Ada compilers, describes compiler implementation techniques that will avoid difficulties, provides exemplary programs that illustrate potential trouble spots in conforming to the standard and clarify intended interpretations of the standard". The implementors guide will be continually updated to reflect increasing understanding of Ada implementation issues.

b) Test programs (approximately 1400 programs with an average of 50 lines of text) which will include both class-A tests which demonstrate that legal programs are accepted and class-B tests which demonstrate that illegal programs are rejected.

c) Validation support tools that assist in preparing tests for execution and in analyzing the results of execution.

Although testing can never guarantee the correctness of the compiler, the set of examples has been carefully selected to test a comprehensive a subset of language features, including features that are tricky to implement. Production compilers will be required to pass these tests before they are accredited as Ada compilers. These tests are more comprehensive and systematic than for any previous language and will set objective standards of correctness that will guarantee high quality for the end product.

The availability of both the test examples and the implementors guide before the first production compilers are completed provides an opportunity to resolve both divergent interpretations of implementation issues and potential differences in program behavior before there is a large community of users with a vested interest in the idiosyncracies of a particular compiler. The danger of proliferating dialects due to subtle differences in implementation is therefore reduced.

The compiler validation effort illustrates the high project management standards of the Ada project. Ada is being developed with considerably more attention to project management than previous languages. Careful project mangement has contributed to clarity and a high level of documentation for the requirements and design of Ada and should contribute to achieving well-designed and high-quality validated Ada implementations. It also illustrates the very considerable effort that is needed to design test systems for specific applications, and the fact that the considerations in developing such systems are related more to the application domain than to general testing principles. Compiling is perhaps not typical of embedded computer applications. Analysis of two or three test systems for typical embedded systems is needed to provide bottom-up insights to help design an "application generator" for the generation of testbeds and simulators for embedded systems.

3. *Programming Language Perspective*

In order to provide some linguistic perspective, let's briefly review the history of programming languages. As illustrated in Table 1, we can subdivide the development of programming languages prior to 1970 into first-generation languages, developed between 1954 and 1958, second-generation languages developed between 1959 and 1961, and third-generation languages developed between 1962 and 1970.

Example 1:

1954-58: First Generation Languages	
Fortran I **Algol 58** **Flowmatic** **IPL V**	**Remarkably large number of features of later programming languages already present in embryo form**
1959-61: Second Generation Languages	
Fortran II	**subroutines, separate compilation**
Algol 60	**block structure, data types**
Cobol	**data description, file handling**
LISP	**list processing, pointers**
1962-70: Third Generation Languages	
PL/I	**Fortran + Algol + Cobol**
Algol 68	**rigorous successor to Algol 60**
Pascal	**simple successor to Algol 60**
Simula	**classes, data abstraction**
1970-80: The Generation Gap	
Many Languages	
None Endured	
The Software Crisis	
Ada - started 1975, ready for use 1985	

Table I: The Development of Programming Languages

First-generation languages include the numerical languages Fortran I (developed by IBM) and Algol 58 (developed in Europe), the business data processing language Flowmatic (developed by Grace Murray Hopper), and the list processing language IPLV (developed by Newell, Simon and Shaw at the Rand Corporation). Many of the basic ideas of programming languages were developed during this period. Fortran I introduced arithmetic expressions and statements, arrays, IF and DO control structures, functions and subprograms. Algol 58 introduced the idea of block structure and declarations. Flowmatic introduced the idea of records and file processing. IPLV introduced the basic concepts of list processing and the idea of garbage collection.

The second-generation languages include Fortran II, the most widely used language in numerical programming, Cobol, the most widely used language in business data processing, Algol 60, which is not as widely used as Fortran and Cobol but is the conceptual starting point for later language development, and LISP which has played a major conceptual role in language development and is widely used in artificial intelligence.

The second-generation languages integrated the ideas of first-generation languages into a working system. Each is a refinement of a specific first generation language, suggesting that language designs benefit when the concept formulation and integration phases are separated by a period of reflection. It is surprising that the most widely used and influential languages of the early 1980s were all developed at essentially the same time (1960 give or take a year).

The third-generation languages attempted to refine the ideas of second-generation languages. But none of them succeeded in gaining as widespread use or influence as the second-generation languages.

PL/I attempted to combine the best ideas of Fortran, Algol, and Cobol. It included the arithmetic expression syntax and separate compilation features of Fortran, the block structure and declarations of Algol 60, and the record and file handling of Cobol. But it was not able to integrate these ideas into an elegant, easily implementable language. PL/I proved to be a baroque language which was unable to compete with Fortran in simplicity of use or efficiency of implementation.

Algol 68 was a well-designed language which, if it had been properly documented, could have been a worthy successor to Fortran. Unfortunately, it suffered from a bad reference manual, and was never able to live down its undeserved reputation for complexity. Algol 68 was technically adequate as a successor to Fortran, but failed because of poor public relations.

Pascal was developed by its designer Nicklaus Wirth as a successor to Algol 60 which supports modern software methodology but emphasizes simplicity over rigor in its language design. It has achieved great popularity, has been widely implemented on microprocessors, and is widely used as an introductory language in college level academic computing courses. However, it has some design deficiencies in its parameter passing and variant record mechanism and does not adequately support the development of large programs consisting of many hundreds of modules.

Simula is a well-designed language which, had it been properly marketed, could have become a successor to Fortran. It supports modular programming through a language mechanism called the "class" which is a forerunner of the notion of packages in Ada, and of the notion of data abstraction. But Simula was developed in Norway and never had the backing of a large and influential user organization.

The attempt to develop general-purpose languages to replace Fortran and Cobol continued from the 1960s into the 1970s. By the early 1970s it was realized that the goal of great expressive power, which was the dominant goal of language design in the 1960s, should be tempered with the goal of simplicity of use and of implementation. It was realized the the software crisis in building large computing systems was due to our inability to handle complexity. Attention shifted from concerns of expressive power in language design to concerns of methodology for the management of software complexity. Pascal was the first language to strive for the goals of simplicity and support of software methodology.

Many languages were developed in the 1970s, but none achieved even the status or influence of third generation languages such as PL/I. The 1970s were a fertile period for the development of language concepts in areas such as program verification and data abstraction, but did not result in an integrated language design challenging the entrenched languages of the early 1960s. The 1970s thus constitute a "generation gap" in the development of comprehensive production-oriented programming languages.

4. *The Ancestors of Ada*

Ada is a "Pascal-based" language. Pascal may be regarded as the "father" of Ada, both because the Ada designers asserted that their design was "Pascal-like" in their design proposal, and because the goals of simplicity and implementability of Pascal are closer to those of Ada than are the goals of comprehensiveness and expressive power of PL/I or those of orthogonality and rigor of Algol 68. Programming languages which have influenced the development of Ada include Simula, which pioneered the notion of data abstraction, and Algol 68 through its clean data structure and type facilities. Simula and Algol 68 may be viewed as "uncles" of Ada. Algol 60 was the father of Pascal, Simula, and Algol 68, and is thus the grandfather of Ada.

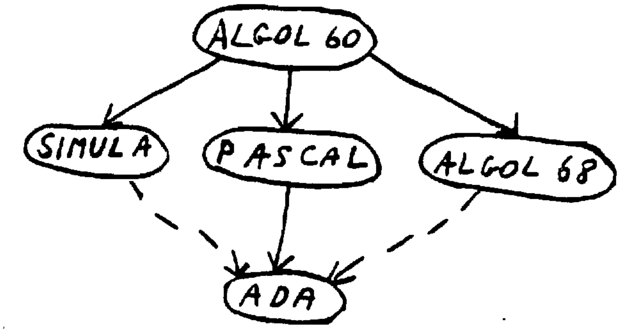


Figure 1: Factors Influencing the Development of Ada

5. *What Makes a Language Successful?*

The failure to develop a viable successor to Fortran and Cobol in the 1960s and 1970s is both surprising and puzzling, particularly in view of the intensive programming language activity during this period. The design and development of a new common language is clearly a very difficult undertaking, in part because the factors necessary to its success transcend the purely technical factors of "quality of design".

The factors necessary to the development of a successful programming language include the following:

a) High-quality design which supports modularity, reliability, efficiency, etc.

b) Efficient, user-friendly, implementation

c) Clout (support by a powerful organization such as the DOD or IBM)

d) Ecological niche (the language must fill a need not met by an incumbent widely used language)

Thus the criteria for success of a language are captured by the following "equation":

SUCCESS = DESIGN + IMPLEMENTATION
+ CLOUT + NEED

The failure of the third-generation languages to replace second-generation languages can be explained by the fact that they did not sufficiently dominate incumbent languages in these four "success factors".

PL/I was strong on clout since it was supported by IBM, but was weak in its design, late in its implementation, and, because of its complexity, did not really dominate Fortran in new application areas.

Algol 68 was stronger than PL/I in its design, but weak on implementation (because of insufficient support) and weak on clout. Pascal was weaker than Algol 68 in its design, strong on implementation (because implementations were relatively straightforward), and strong on ecological niche (for small academic application). But its niche did not include the large embedded computer applications that provided the impetus for the development of Ada. Simula was strong on design and came closer than other third generation languages to filling the embedded computer niche, but was weak on clout and implementation.

Ada appears to dominate third-generation languages on the above criteria for success. It is strong on design (although its design could undoubtedly be improved). It is strong on clout, since it is supported by the DOD and is being adopted by major computer companies and the European Economic Community. It is strong on "need" because of the demonstrated inadequacy of current languages for large embedded computer applications.

Ada is probably closest to Cobol in its political and technical development. Both Cobol and Ada derived their initial clout from the "Department of Defense" and were developed in response to a specific need. A comparison of Ada, Pascal, Cobol, and PL/I on our four criteria is given below:

	ADA	Pascal	Cobol	PL/I
desiyn	B	B	C	C
implementation	?	A	B	B
clout	A	C	A	A
need	A	B	A	B

Figure 2: Comparison of Programming Languages

The major question-mark in the development of Ada is the fact that there are not yet any production-quality implementations. But all indications are that Ada can and will have efficient, user-friendly implementations by 1984. Existing experimental compilers have proved feasibility, production compilers are being undertaken by software houses with a great deal of experience in developing compilers of comparable complexity, and the level of public scrutiny and rigor of validation requirements are greater than ever before. All these factors lead to optimism with regard to both the quality and user-friendliness of the implementation. If the schedules and performance expectations of implementations are met during the next two years, then the probability that Ada will become the common language of the late 1980's and 1990's is very high indeed.

6. *Capital-Intensive Technology and Reusability*

One of the parallels between traditional engineering and software engineering is the role of capital goods in increasing productivity and reliability. The industrial revolution led to an economic system called capitalism and to a process of capital formation which has caused production of consumer goods to become progressively cheaper, more versatile, and more reliable. The "consumer goods" of software engineering are not as tangible as industrial goods. But the notion of capital goods, such as programming languages and software tools, for improving productivity and reliability is just as central to software engineering as it is to traditional engineering.

A production process is capital-intensive if it requires expensive tools or involves expenditures early in its life cycle for the purpose of increasing productivity later in the life cycle. Software technology, just as traditional technology, was labor-intensive in its youth and is becoming increasingly capital-intensive as it matures both because tools and application programs are becoming more ambitious, and because modular program development methodologies require considerably greater expenditures during program design to reduce expenditures in later phases of the software life cycle.

The Encyclopaedia Britannica states that "capital" is a word of many meanings which have in common that capital is a "stock" rather than a "flow". [EB, 1968.] It asserts that "In its broadest sense capital includes the human population; non-material elements such as skills, abilities, and education; land, buildings, equipment of all kinds; and all stocks of goods - finished or unfinished - in the hands of both firms and households." We shall hazard a simple definition of capital that is consistent with the above statements, and show that many activities in software and knowledge engineering are capital-intensive according to this definition.

"Capital" may be defined as a reusable resource. Capital goods in conventional engineering, such as a lathe or an assembly line, are reusable resources for producing consumer goods. Capital goods in software engineering, such as compilers and operating systems, are reusable resources for producing application programs. Application programs and databases are reusable resources which may be repeatedly used in computing useful results. Programmers are reusable resources in the production of programs. Capital goods are produced only once and may then be used repeatedly in the production of economically useful products. The process of developing capital goods for use in the production of consumer goods is called capital formation. Technologies for producing consumer goods which depend heavily on capital goods are called capital-intensive technologies.

Reusability is a general engineering principle whose importance derives from the desire to avoid duplication and capture commonality in undertaking classes of inherently similar tasks. It provides both an intellectual justification for research that simplifies and unifies classes of phenomena, and an economic justification for developing reusable software products that make computers and programmers more productive. The assertion that we should stand on each other's shoulders rather than on each other's feet may be interpreted as a plea for reusability.

The notion of reusability captures the essence of distributing capital cost over the set of actual or potential uses of a capital good. It may be used to justify capital-intensive activities for physical products of conventional engineering, conceptual products of information engineering, social products such as education, and abstract products such as concepts and theorems. For example, the capital cost of education may be justified by its contribution to the reusability of people, while the capital cost of research may be justified by the reusability of research results in improving productivity.

The fundamental economic motivation for the development of general-purpose computers is the reusability of computer hardware. Software provides an essentially cheaper and more flexible mechanism than hardware of realizing a broad spectrum of logical behaviors on a given physical device. General purpose computers are a capital-intensive response to the information processing needs of society that allow critical computing resources such as the central processing unit to be reused one million times per second, and less critical resources such as the computer memory to be reused for a sequence of jobs with very different behavioral characteristics. Reductions in hardware costs have reduced the disparity between costs of software and hardware implementation and have resulted in the proliferation of dedicated computers devoted to single applications. But software will continue to have the advantage of dynamic reusability of resources within a single application even if hardware costs become totally negligible.

There are many synonyms for reusability, including commonality, portability, abstraction, and generalization. The development of common programming languages such as Ada and common families of computers such as the IBM

370 and the military computer family [MCF] are capital-intensive activities that have consumed many thousands of man years of effort and are expected to have large economic benefits. Portability allows software written in one computing system to be reused in other systems. Abstraction is concerned with capturing the common (reusable) features of a class of objects, situations or processes and ignoring their differences. Generalization, which is closely related to abstraction, extends the domain of applicability of a concept, situation, or process, so that it is applicable (reusable) in a broader variety of situations.

Our definition of capital in terms of reusability is so broad that it makes almost any activity in computer science capital-intensive. But this is not necessarily a flaw in the definition. It reflects the fact that information is generally used as a tool to accomplish some larger purpose. Consumer goods of the information revolution include newspapers, videogames, and systems such as airplanes and washing machines in which computers are embedded. However, computing systems used in program development are almost entirely devoted to capital intensive activities. As society becomes more information intensive the proportion of resources spent on capital goods is likely to increase.

Our definition of capital does not distinguish between public capital goods which are freely available to all users and private capital goods which are proprietary or available only under a licensing arrangement. Public capital goods are more freely available in the software industry than in conventional engineering industries because the cost of replicating software products is negligible. This changes the pattern of economic incentives and may create a discrepancy between the public interest, which would benefit from free availability of all capital goods, and the private interest, which can gain a competitive advantage from proprietary capital goods. Thus the programming language Ada, which was developed by the Department of Defense as a public capital good to stimulate increased productivity in a broadly based user community, is viewed with apprehension by some established software companies because its success could wipe out their competitive advantage from current proprietary computer systems. The definition of "capital" in terms of reusability models the increased productivity resulting from public capital goods, but not the economics of private capital goods under "capitalistic" competition.

7. *Economic Impact of Ada*

Ada was developed in response to the software crisis of the 1970s to support the design, development, and enhancement of large, real-time, evolutionary computing systems. It is a prototypical product of capital-intensive software technology, requiring ten man-years of elapsed time and hundreds of man-years of effort to produce and holding out the promise of great improvements in software productivity. It involves reusability at several different levels.

(1) Commonality, which facilitates reusability by a large user community. It distributes development cost over a large number of applications, and avoids duplicate expenditures on special-purpose programming languages and system software.

(2) Portability, which allows system and application software (written in Ada) to be reused on a variety of computers. This further distributes development costs and avoids program duplication.

(3) Modularity, which facilitates the construction of libraries of reusable software components. Modular program design is capital-intensive since it increases expenditures early in the life cycle for the purpose of later savings. Modularity facilitates maintenance and enhancement by localizing the effect of program changes, and allowing existing code to be easily reused when the code is changed.

(4) Maintainability, which facilitates reusability of existing software when it needs to be modified. Both maintenance and enhancement involve relatively small changes to a large system, and should require a small effort commensurate with the magnitude of the changes rather than with the magnitude of the system being modified.

(5) Software tools, which extend commonality (reusability) from the programming language to the program support environment. Program support environments, whether for Ada, Interlisp, or APL, require greater effort to build and result in much greater benefits than just a language and its compiler. The term "software tools" consciously suggests parallels with capital-intensive tools of traditional engineering.

(6) Methodology, which extends commonality from the level of system software to the level of concepts and software practices.

(7) Education, which contributes to the reusability of people. Education transfers the stock of knowledge (conceptual capital) from teachers and textbooks to the minds of students. It requires reusable textbooks, course materials, and knowledge about programming. The stock of (reusable) knowledge in human minds is the primary capital-intensive starting point of any technology.

The potential benefits of Ada arise from a number of different kinds of reusability, including reusability of the language, of programs written in the language, of principles of program development, and of people whose knowledge is reusable. Each of these forms of reusability has generated a large literature both in the context of Ada and in the larger context of software engineering. Our purpose here is not to explore the detailed implications of these forms of reusability but to point out that reusability is a common principle that may occur in different forms and contribute in different ways to making software products capital-intensive. The above qualitative examination of the factors which make Ada capital intensive could provide a framework for a quantitative cost-benefit model for capital-intensive software products that complements Boehm's model of the software life cycle. [Boehm, 1981.]

Ada was developed for the purpose of increasing software productivity and reliability through federal initiatives. It is an experiment not only in the development of a new language and technology but also in the use of incentive schemes for capital formation. The incentive structure of the Ada effort appears to have worked well in the requirements and design phase, and is now being used in implementation and technology transfer activities. It has generated considerable derived capital investment in the private sector. The success of the Ada effort in allowing diverse constituencies to work together towards a common goal suggests that public initiatives can play a major role in making our information environment more capital-intensive.

Ada started out in 1975 as simply a programming language, evolved in the late 1970s to include software tools and program support environments, and has come to serve as a focus not only for technical but also for social and educational activities in modernizing software technology. We shall briefly indicate Ada's technical contribution to interface technology and then review some non-technical issues required to realize large-scale changes in technology associated with introducing a new programming language.

A key feature of Ada is its treatment of module interfaces as independent entities that can be separately specified and separately compiled. Interface specifications are reusable for a variety of different implementations, including software, hardware, or VLSI implementations, and provide a basis for a capital-intensive software components technology. The interface specification facilities of Ada are an improvement over those of earlier languages such as Fortran, Pascal, or PL/I, but still have rough edges and should be improved in successors to Ada. The fact that Ada programs will make heavy use of modular interface technology should make the transition from Ada to a successor easier than that from current languages to Ada.

Ada supports concurrent and real-time programming through its task facilities. Concurrency may be used to speed up inherently sequential computations by parallel execution of independent subcomputations. A more fundamental reason for the importance of concurrency is that banks, ships, and cities are more naturally modelled by concurrent than sequential processes. Concurrent programming languages are intrinsically higher-level than sequential languages because they avoid the need to specify the order of execution in cases where this is an implementation detail. Ada provides logical concurrency that may be physically realized on a uniprocessing or multiprocessing computer. Mapping of concurrent processes onto physical processes is performed by the system rather than the user.

In order to succeed, a language must be not only technically adequate, but also accepted and properly used by the user community. The proper use of Ada requires new techniques for programming and problem solving radically different from those used for Fortran and assembly language. An extensive education and technology transfer program is required to accelerate transition to the new technology. Techniques of knowledge engineering, including computer-based teaching, can facilitate the process of technology transfer.

Ada is the first language developed by the systematic use of life cycle technology. Its requirements were developed in 1975-1978 through a sequence of requirements documents culminating in the Steelman requirements. Its design phase from 1977 to 1980 resulted in the preliminary reference manual [Ada, 1982] which is still being refined with a view to standardization in 1983. The implementation phase started in 1979 and is projected to be completed in 1984, with implementations for a variety of computers, including microcomputers. Implementations will include program support environments for debugging, testing, and software management as well as environments for learning Ada that are integrated with program support environments for using Ada. The operations and maintenance phase will begin in 1985, and should allow periodic evolutionary enhancement of the language, as well as the possibility of revolutionary transition to a new language.

The life cycle of Ada should include an education phase and technology transfer phase that runs concurrently with the traditional requirements, design, implementation, and operations and maintenance phases, as illustrated by the following Ada program.

Example 2: Software Life Cycle with Concurrent Activities

```
procedure LIFE_CYCLE is
  task HUMAN_RESOURCES;  -- task declaration
  task body HUMAN_RESOURCES is
  begin
    EDUCATION;
    TECHNOLOGY_TRANSFER;
  end;
begin                    -- concurrent execution of task
  REQUIREMENTS;          -- with statements of procedure
  DESIGN;
  IMPLEMENTATION;
  OPERATION_AND_MAINTENANCE;
end LIFE_CYCLE;
```

This program illustrates the use of Ada to specify concurrent activities that arise in informal discourse - in this case in the context of the software life cycle. Ada can be used as a tool for organizing not only programs but also projects, lectures, and documents, and we shall probably see the increasing use of Ada by managers to express organizational structure.

In order to make education and technology transfer first-class phases of the software life cycle, tools and management procedures for these life cycle phases must be developed. It is necessary to identify constituencies to be educated, develop curricula tailored to each constituency, and ensure that teaching is properly integrated with subsequent use of the concepts being taught. Incentives should be heavily used in accelerating the process of technology transfer. Examples of such incentives are carrot-and-stick inducements to middle managers to encourage high-risk retooling for modern technology at the expense of comfortable but obsolete current practices. Well-documented case studies tailored to major application areas are needed which supplement toy examples of short courses and textbooks and allow programmers to learn the new language by speaking it. These issues have been addressed in a Softech study for the Army [Softech, 1982] and are discussed in [Wegner, 1982a].

8. Maintenance, Enhancement, and System Evolution

In any technology there is a tradeoff between short-term optimization for a particular product and evolutionary flexibility in adapting to change. In industrial technology the tradeoff is generally in favor of product optimization. However, software systems must be designed for evolution to cope with the greater variety of potential uses, changing environments of use, and a rapidly changing technology. An evolutionary technology is capital-intensive according to our definition, since evolution requires reuse of present resources in building future resources. It is also capital-intensive in the intuitive sense, since greater expenditures are required up-front to allow later flexibility.

Complex structures, both natural and social, are generally the result of evolutionary development rather than of a single creative act. This is also true of large software systems constructed by people, whose ability to manage complexity is very limited. A new programming language such as Ada evolves from experience in the design and implementation of previous languages such as Pascal, and in turn forms a basis for the design of future languages. Successful programming systems such as UNIX evolved from small beginnings and have achieved their success by having a simple core to which facilities can easily be added by a wide variety of system programs. Maintenance and enhancement requires large software systems to be constructed so that modification and evolution can be accomplished in a time proportional to the magnitude of the changes rather than the size of the system.

Systems which have the property of being easily maintained and enhanced when they are large have the property of being locally modular and of being constructible in an evolutionary manner from primitive components. They can be constructed by incremental "builds" of subsystems which serve as prototypes in the construction of larger systems. An adequate solution of the maintenance and enhancement problem implies evolutionary system structure not only for a large system as a whole but also for its component subsystems. Maintainable systems require not only static modularity but also "dynamic" modularity that facilitates incremental development, testing, and rapid prototyping. Solution of the maintenance problem requires an evolutionary life cycle methodology which allows complex systems to evolve from a simple core by multiple independent extensions. Failure to find a simple expandable core may result in system rigidity even if the system has a high degree of modularity. The additional requirements on modularity needed to support evolutionary development deserve further study.

Iterative enhancement [Basili, 1982] is an example of an evolutionary life cycle approach. It advocates using a skeletal implementation (rapid prototype) as a starting point for iterative redesign of what has already been produced and evolutionary addition of new features until the system is completed. When the set of tasks to complete a project can be predicted they can be listed in a project control list and systematically scheduled. The approach is useful even when the set of subtasks and the end result are incompletely defined. For example the present paper was developed by iterative enhancement of an incomplete specification, starting from a brief discussion of the capital-intensive nature of Ada and growing by iterative revision and expansion to its present scope and size. The technique of iterative enhancement was first developed in the context of software engineering, but may turn out to be even more pertinent to the writing of papers and books, where evolution is an inherent part of the process of creation. Text-editing systems and other computerized knowledge engineering aids greatly facilitate evolutionary development of manuscripts and are likely to have a profound impact on the writing habits of both technical and non-technical authors. This paragraph was one of the later additions during the iterative enhancement process that led to the present paper.

Lack of evolutionary flexibility contributed to the failure of technologically advanced countries like Great Britain in coping with competition of countries whose industrial development occurred later in time. It could similarly lead to dissipation of the current US lead in the software field to countries like Japan whose software technology is less dependent on old software systems and management structures. Inability to adjust to changing technology was a cause of great pain and social dislocation in the industrial revolution. Evolutionary flexibility both for individuals and for the technology as a whole should be a primary goal of information technology.

The evolution of software systems may be viewed as a special case of the evolution of organizations with both human and computer components. There is a considerable literature on the structure, social dynamics, and adaptability of organizations. At a very general level, Toynbee's study of the genesis, growth, breakdown, and disintegration of civilizations is about the failure of civilizations to adapt to changing environments. [Toynbee, 1947.] The text "Organizations", which is a source book for Simon's Nobel-prize winning work on formal models of organizational behavior, is a good starting point for readers interested in this area. [March and Simon, 1958.] "Organization Development" presents an analytical framework for the development of organizations in terms of flows of information among their components. [Shein, 1973.] Holland explores the problem of adaptation for both natural and artificial systems, emphasizing the response of such systems to a changing environment. [Holland, 1975.] The similarity of models of large industrial organizations and large computer systems is reflected in the computer literature in anthropomorphic terminology such as "actors" and "messages" in the modelling of "societies" of interacting computer programs. [Hewitt, 1977.] Milner's Calculus of Communicating Systems is an example of a formal (algebraic) model of communicating systems whose components may be people or computers. [Milner, 1980.] The study of evolutionary behavior of mixed man-computer systems, and of interfaces that allow creativity and growth of people in a computer environment, is central to the development of a capital-intensive software technology that combines current efficiency with adaptability to change.

The above discussion suggests a distinction between "evolution in the large" for very large organizations with long time horizons measured in decades or centuries, and "evolution in the small" for smaller (but still large) organizations and time horizons measured in months or years. Adaptation to changing technology is concerned with evolution in the large, while development, maintenance, and enhancement of a particular system is concerned with evolution in the small. Tuning of a system for a particular set of tasks and time horizons may increase its cost and reduce its efficiency for narrower classes of applications and constrain its adaptability for broader classes of applications.

The problem of evolutionary flexibility arises in its most acute form in the context of adaptation to a changing technology. It is a bottleneck in the adaptation of embedded systems to changing environments, since maintenance and enhancement accounts for 80% of total life cycle costs. Adaptive systems which can acquire and subsequently use knowledge, such as expert systems or theorem provers, must have databases designed for evolution. The three examples above are concerned with adaptation to different evolutionary goals: adaptation to changing technology, changing environments and changing knowledge databases. But in all three cases design for evolution involves reusability in response to change. Thus evolutionary systems are capital-intensive according to our definition. Evolution, adaptation and maintainability are additional synonyms for reusability.

9. *Knowledge Engineering*

Knowledge engineering is here defined as "the application of systematic techniques to the management and use of knowledge". It is, in this sense, as old as knowledge itself. Euclid's Elements is an example of a magnificent piece of knowledge engineering which provided a basis for managing geometrical knowledge, while the classification techniques of Linnaeus are an important example of knowledge engineering in botany and biology. Many of the milestones in the development of science are as important for their contributions to the management of knowledge as for their contributions to knowledge itself. Knowledge engineering is capital-intensive in the sense that reusability is a primary consideration in the development of books, expert systems, and other structures for the management and use of knowledge.

The potential of computers as tools for knowledge engineering was realized as early as 1945 by Vannevar Bush, who examined techniques for fundamentally reorganizing knowledge and proposed a device called a memex for the storage, retrieval, and management of knowledge. [Bush, 1945.] In the 1960s, Douglas Engelbart proposed a systematic research program on the use of computer technology to augment man's intellectual capabilities. [Engelbart, 1963.] The personal computer technology of the 1980s may, for the first time, allow us to realize the pioneering ideas of Bush and Engelbart.

The view that the production of knowledge is an economic activity governed in part by market forces of the economy is developed in detail by Machlup in a comprehensive study of the economics and the substance of the knowledge explosion. [Machlup, 1980.] Knowledge is becoming an increasingly important product of our economy, as reflected by the size of the education industry, the growth of the computer industry, and the fact that the average age when people start contributing to the economy has increased from the early teens in the industrial revolution to over 20 for college graduates, and over 25 for professionals like doctors and lawyers. Knowledge is a stock of capital goods and its production is a capital-intensive activity. The growing importance of the knowledge industry reflects the fact that man is becoming an increasingly capital-intensive animal.

Fourth-generation computers will cause fundamental changes in our methods of managing, learning, and using knowledge. New ways of representing and organizing knowledge to exploit interaction, animation, two dimensional objects, multiple windows, and other forms of knowledge presentation will be developed. A more effective man-computer interface for management of knowledge will complement artificial intelligence techniques for knowledge acquisition and problem solving by computers and result in an environment that integrates human and computational management of knowledge. It is predicted that, by the 1990s, knowledge engineering will be as important a subdiscipline of computer sciences as software engineering is today. Some of the characteristics of the emerging field of (computer-based) knowledge engineering will be outlined below.

Knowledge engineering depends on the representation of knowledge by information structures inside a computer. The principle that knowledge as well as numbers can be represented in a computer was recognized right at the outset and led to work in artificial intelligence, natural language translation, and information retrieval in the 1950s. However, computer-based knowledge engineering could not reach critical mass before the 1980s because of inadequate technology. Cheap powerful graphics-based personal computers which may be carried in a briefcase or a pocketbook and used on a day-by-day basis as an extension of the human intellect will entirely change the relation between man and computers.

Knowledge engineering bears the same relation to the management of knowledge that software engineering bears to the management of software. An item of knowledge, like an algorithm, is an inherently conceptual object that can be given a concrete representation by an information structure and manipulated, "used" or displayed by a computer. The creation of computerized knowledge structures representing substantial bodies of knowledge requires techniques for the management of information complexity similar to those required for a large program. The common ancestry of Software- and knowledge engineering as branches of information engineering is reflected in the sharing of certain methodological principles. The computer revolution is likely to spawn many different kinds of information engineering just as the industrial revolution generated many different kinds of "physical" engineering disciplines. They will share with software and knowledge engineering the idea of representing a class of conceptual objects by concrete information structures and the need to manage complexity when structures become large and may evolve over time.

The term "knowledge engineering" was introduced by Feigenbaum in the context of artificial intelligence and defined as "the art of bringing the tools and principles of artificial intelligence to bear on application problems requiring the knowledge of experts for their solution". [Feigenbaum, 1977.] This definition views knowledge engineering as the art of representing knowledge so that it can be used by computers to perform intelligent tasks.

The present view of knowledge engineering is broader, since it includes the building of knowledge structures to aid human understanding. It is closer to that of Stefik and Conway [SC, 1982], who examine the role of knowledge engineering in a rapidly evolving knowledge domain (VLSI), discuss writing of a text for a rapidly evolving knowledge domain (the Mead-Conway VLSI book [MC, 1980]), and consider the application of knowledge engineering techniques by practitioners to the simplification and refinement of their knowledge.

Knowledge engineering for human understanding is motivated by a paradigm different from that which motivates the development of expert systems. Its goal is to amplify human intelligence rather than to substitute computer intelligence for human intelligence. Its methodology involves educational technology, cognitive science, and human-factors research. The technology of managing the modular presentation of complex knowledge structures has some of the flavor of software engineering but requires consideration of human factors associated with animation, user interaction, multiple windows, and other techniques for increasing the effectiveness of man-machine communication.

Knowledge representations should facilitate display for the benefit of users, including multiple views and other forms of redundancy, rather than efficiency and precision for the benefit of computers. Whereas knowledge structures for computer understanding must be very detailed and precise, knowledge structures for human understanding are concerned, not with the precise specification of a computational task, but with organizing knowledge for human readers who possess considerable contextual understanding and are capable of conceptualizing at a level far above that of the computer.

The restructuring of existing knowledge so that it is more accessible to humans involves more than putting existing knowledge repositories such as the Library of Congress on computers and accessing them through information retrieval systems. It involves restructuring existing knowledge so that it can be flexibly presented in different formats for different contexts of use. The technology for such restructuring is not well understood, but its nature can be illustrated by considering recent developments in computerized printing technology and computer-based learning.

Computers are revolutionizing printing technology to allow high-quality text to be quickly and cheaply produced. Word-processing systems provide authors with much greater control over the production, layout, and modification of text. Soon computers will be used not only for writing and printing books but also for reading them. Book-size computers with flat panel displays will make "electronic books" a reality. The greater bandwidth of man-machine interfaces will qualitatively change the nature of man-machine communication, and will make communication of knowledge by interacting with computer books more effective than conventional communication by reading hard-copy books.

Whereas hard-copy books consist of a linear sequence of pages, materials intended to be read on a computer may have a graph structure with different entry points for readers with different backgrounds. Multiple windows allow the reader to pursue several lines of thought simultaneously or view a given object at several levels of detail. Interactive responses by the user can be used by the computer to tailor the mode of graph traversal to the interests and skill level of the student. Each node of the graph structure can include dynamically animated pictures, texts, and programs. For example, the mathematician may wish to animate the development of a proof, while the computer scientist may wish to animate the process of program development and program execution.

An electronic book is a family of different hard-copy books that could be obtained by printing out nodes of the graph structure in a particular linear order for particular kinds of students. It is conjectured that flexibility in adapting the pace and order of presentation of information to the student, combined with the power of animation (possibly augmented by voice input and output) can, if properly used, increase enormously the student's capacity to absorb and understand both elementary and advanced knowledge.

"Knowledge graphs" that may be entered at different points and traversed in different ways represent a paradigm for knowledge engineering that imposes a modular, interactive discipline on both creators (authors) and users (students). They are a basic representation not only for electronic books, but also for computer games such as Adventure which derive their fascination from the fact that they allow players to explore new graph-structured worlds. We do not yet have much experience with building large knowledge graphs since the hardware technology to support effective use of such graphs is only just being developed. Some features of such graphs are briefly described below.

Knowledge graphs should have a domain-independent interconnection structure that facilitates several modes of graph traversal such as browsing, retrieval, learning, reference, authoring, etc. Each node will have a domain-dependent internal structure containing objects such as programs when representing knowledge about programming, and proofs when representing knowledge about mathematics. Creators and users of a graph will have available to them a domain-independent set of operations for navigating in the graph and domain-dependent operations for manipulating objects in each domain. The Zog system is probably the best known current example of a general-purpose system of this kind. [RMN, 1981.]

10. Conclusion

Software technology is becoming a key factor in maintaining industrial (and military) competitiveness. The development of capital-intensive software technology is too important to be left entirely to chance or free enterprise. It requires coordinated national (and possibly international) plans and incentive schemes. The development of long range software technology plans was pioneered by the Japanese fifth generation computer proposal - a ten year university-industry effort for the development of an integrated computing system that combines advanced interface technology and knowledge engineering techniques. The United States has responded with the DOD software technology initiative [*Software*, 1982]. Great Britain and the European Economic Community have also produced long-range software plans in 1982.

The significance of these proposals lies not so much in their detailed recommendations but in the recognition that the post-industrial information age will require fundamentally new ways of doing business and radical technological adjustments. Further progress in making software cheaper and more reliable is likely to come from managerial rather than technological breakthroughs. The Japanese may capture a disproportionate share of the computer market not because of superior technology but because of more flexible management techniques that allow the results of mainstream software technology to be harnessed more quickly and effectively.

The design of assembly lines for automobile production by Henry Ford represented a major breakthrough in industrial technology. What is needed is a comparable management breakthrough in software technology. The fact that specific technical features of the Japanese proposal (such as the use of Prolog as a system programming language) are controversial is of minor importance compared with the fact that the Japanese have a more flexible management structure that can adapt more easily to changing management requirements.

The DOD Software Initiative is concerned with ways of improving software productivity and reliability so that the United States can maintain its competitive edge in the software field and close the widening manpower gap between the demand and supply of qualified software personnel. It advocates a three-pronged strategy for increasing productivity:

(1) **Education:** Improve human resources by increasing the quality and number of experts.

(2) **Tools:** Improve the power of project management tools, application-independent program support tools, and application-specific tools (such as testers and simulators).

(3) **Technology Transfer:** Increase the use of tools by improving business practices and user-friendliness of tools and by increasing the level of integration and automation.

This strategy recognizes that the problems of increasing productivity and using new technology effectively are as much due to educational and social as to technical factors. The United States excels in volume and quality of its research and development, but is sometimes slow in harnessing the results for production purposes. Mechanisms which speed up the process of technology transfer and allow new technologies to be used productively will have an enormous payoff, and merit expenditures hundreds of times greater than current levels. The time for new products and technologies to make the transition from research to production averages fifteen years. Acceleration of the transition to modern software technology by a year or qualitative improvement in its use has enormous leverage, since US software expenditures are expected to exceed 40 billion dollars per year by 1990. If software productivity could be improved by even 1% through systematic education and technology transfer policies it would be worth 400 million dollars per year.

An appendix on "visions of the future" includes an "evolutionary scenario" which explores improvements in productivity by consolidation of tools and techniques within the traditional life cycle paradigm, and a "revolutionary scenario" which explores improvements in productivity through a radically new paradigm based on very high-level problem specifications and "broad spectrum languages" which allow users to perform problem specification, maintenance, and enhancement in terms of very high-level abstractions, and provide automated verifiable transformations into lower-level abstractions. The evolutionary scenario is predicted to increase productivity by a factor of two by 1985 and by a factor of four by 1990, and would be just sufficient to close the projected manpower gap. The revolutionary scenario has higher risk but could, if successful, improve productivity by considerably more than a factor of four.

Since software technology is pivotal in maintaining both military and economic competitiveness, we should simultaneously explore a variety of future technological scenarios. Our course of action in the next decade is likely to be somewhere between the evolutionary and revolutionary scenarios. New computer architectures, better interface technology, and new approaches to the management and engineering of knowledge are likely to cause radical changes in the structure and economics of the software life cycle. With proper integration of technical, educational, and social factors the new technology could increase productivity by several orders of magnitude. But the introduction of very high level abstractions, automated verification, natural language understanding, and intelligent adaptation is likely to be slow and difficult. Increases in productivity are likely to result from integration of current techniques rather than from automating fundamentally new forms of intelligent behavior.

The role of testing in the emerging technology of the 1990s will differ substantially from its current role both because of different life-cycle methodologies and because of new tools for management, testing, and validation. The nature of these differences cannot be entirely predicted but are a fruitful topic for discussion and research.

11. References

[Ada, 1982] Ada Reference Manual, U.S. Department of Defense, July 1980. Revised Version, Summer 1982.

[Basili, 1975] Basili Victor R. and Turner Albert J., Iterative Enhancement, A Practical Technique for Software Development. IEEE Transactions in Software Engineering, December 1975.

[Boehm, 1981] Boehm Barry, Software Engineering Economics, Prentice Hall, 1981.

[Bush 1945] Bush Vannevar, As We May Think, Atlantic Monthly, July 1945.

[EB, 1968] Encyclopaedia Britannica, 1968 Edition.

[Engelbart 1963] A Conceptual Framework the Augmentation of Man's Intellect, in Vistas in Information Handling, Vol 1, ed Howerton and Weeks, Spartan, 1983.

[Feigenbaum, 1977] Feigenbaum Edward A., Case Studies in Knowledge Engineering, Proc. Fifth Int. Conf. on Artificial Intelligence, August 1977.

[Fifth 1981] Preliminary Report on Study and Research on Fifth Generation Computers 1979-1980, Fall 1981, Japanese Information Processing Development Center.

[Good, 1979] Goodenough John, A Survey of Testing Issues, in Research Directions in Software Technology, ed Wegner, MIT Press, 1979.

[Hewitt, 1977] Hewitt Carl, Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, June 1977.

[Holland, 1975] Holland John, Adaptation in Natural and Artificial Systems, An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. The University of Michigan Press, 1975.

[Machlup, 1980] Machlup Fritz, Knowledge, Its Creation, Distribution, and Economic Significance, First of Several Volumes, Princeton University Press, 1980.

[March, 1958] March James G., and Simon Herbert A., Organizations, Wiley 1958.

[MC 1980] Mead Carver and Conway Lynn, Introduction to VLSI Systems, Addison Wesley, 1980.

[MCF] Military Computer Family Report

[Milner, 1980] Milner Robin, A Calculus of Communicating Systems, Lecture Notes in Computer Science #92, Springer Verlag, 1980.

[RMN, 1981] Robertson G, McCracken D, and Newell A, The Zog Approach to Man-Machine Communication, International Journal of Man-Machine Studies, 1981

[Schein, 1973] Schein Edgar H. et al, Organizational Development, Addison Wesley 1973.

[Simula, 1972] Dahl O. J., and Hoare C. A. R., Hierarchical Program Structures, in Structured Programming, Academic Press 1972.

[SC 1982] Stefik Mark and Conway Lynn, Towards the Principled Engineering of Knowledge, Xerox PARC TR KB-VLSI-82-18, April 1982.

[Software, 1982] DOD Software Technology Initiative, October 1982.

[Softech, 1982] Ada Software Design Methods Formulation, Softech Report in Three Volumes, Report for Army Contract, 1982.

[Toynbee, 1947] Toynbee Arnold J., The Study of History, Oxford University Press, 1947.

[Wegner, 1982a] Wegner P., Ada Education and Technology Transfer Activities, Ada Letters, August 1982. Also available as TR CS 82-20, Brown University.

[Wegner, 1982b] Wegner P., Towards Capital Intensive Software Technologies, TR CS-82-23, Brown University.

ABOUT THE AUTHOR

Peter Wegner was educated in England and taught at the London School of Economics, Penn State, and Cornell prior to his present position at Brown University. He is the author of six books, including the first book on Ada (Prentice Hall, 1980) and an edited book on "Research Directions in Software Technology" (MIT Press, 1980). His current interests include programming language design, programming methodology, and the use of graphics-based personal computers for education, technology transfer, and knowledge engineering.

IMPACT OF NEW HARDWARE TECHNOLOGY ON SOFTWARE DEVELOPMENT

Dr. D.A. Giese

W.P. Jones, Jr.

TRW Electronic Systems Group

ABSTRACT

This paper summarizes recent experience in software development efforts for systems using new hardware technology. In particular, the impact of new hardware technology on real-time signal processing applications utilizing embedded high speed programmable signal processors is reviewed. The concurrent design and development of hardware and software presents additional problems to software development and testing. Potential solutions to deal with these problems are discussed briefly. Finally, areas for further investigation are recommended.

INTRODUCTION

Recent advances in hardware technology are providing significant increases in system performance. However, these new capabilities will significantly impact all aspects of the software development process from design through integration and test. This impact has been felt most dramatically in advanced systems which require concurrent hardware and software development efforts.

Ultimately, advanced hardware technology provides the means to improve the software development process. The greatest benefit is the reduction in program complexity and improvement in reliability by eliminating the need for a high degree of optimization. Programming efforts will be reduced with rich instruction repertoires that include high level functions tailored to an application. Advanced hardware technology can also provide sophisticated built-in test capabilities that can aid in system/software integration and test activities. In the future, new processing architectures such as high level language machines and high performance data base machines will reduce the programming effort as well as sensitivities to hardware implementation details.

In the near term, software engineers face considerable problems in exploiting new hardware technology since advances in this area have far outpaced software technology. Typical of the problems which must be overcome are:

- A "hardware-first" design philosophy is no longer feasible. Considerations for programmability and software testing must be made early in design cycle.

- Hardware changes/modifications cause the support software environment to be unstable. Software engineers must find ways of coping with this situation.

- Hardware/software integration and testing will uncover many errors due to lack of hardware maturity. Better analysis tools are needed to isolate hardware/software problems.

This paper summarizes recent experience in the development of real time signal processing software for systems using advanced technology. Technology, in this context, includes both advanced processor architectures and very large scale integrated (VLSI) circuits. In the course of recent software development efforts for high speed digital processors, several techniques have been identified which provide a means to effectively use and cope with new hardware technology insertions. These techniques are briefly discussed in the following sections.

MULTI-LEVEL SIMULATION

With target hardware unavailable at the start of software development, multi-level simulations can provide valuable design information, as well as a development base. Simulation levels of particular interest are:

- Functional or algorithmic

- Processor Instruction Set Architecture (ISA)

The functional simulation models the system processing without concern for the implementation details. It is primarily used as an algorithm test bed. Through the functional simulation, valuable information can also be obtained with regards to candidate hardware modules to satisfy processing demands, preliminary performance estimates and suitabile test cases for system level testing. The Instruction Set Architecture (ISA) simulation models the hardware operation with proper timing and functionality. It is essentially a virtual machine hosted on a general purpose processor, such as a VAX. The results from this simulation should faithfully reproduce the results on the target hardware at the bit level. The ISA simulation provides a means of developing and testing software before the target hardware is available. The ISA simulator should be table driven in some fashion to allow retargeting for different hardware configurations as well as adapting to hardware modifications during development. The ISA simulator development represents a critical schedule item since it requires detailed hardware design information, and must be available for software development. Ideally, the fundamental properties of the hardware architecture would be described in a Hardware Description Language (HDL) and could be used with other support tools.

RETARGETABLE SUPPORT TOOLS

Current software tools for high speed signal processors are relatively primitive, with most support being for machine/assembly language programming. Furthermore, the existing support tools are specialized for each processor.

To fully exploit advanced hardware capabilities, a complete set of software development tools is needed, all of which should be table driven as previously discussed. In particular, retargetable HOL compilers are of prime importance. However, for a retargetable compiler for signal processors to be feasible, the possible hardware architectures must be restricted. Even with architecture restrictions, there would be substantial configuration and implementation freedom which encompasses a great many of signal processors in existence today. Further study in this area can have great leverage on system software development costs.

A retargetable HOL compiler would have the following components:

- Syntax, semantic analysis
- Functional decomposition (high level functions into processing primitives realizable in hardware)
- Resource allocation
- Code generation

Each component becomes successfully more processor dependent.

Other hardware dependent support tools in a signal processing software development environment include, but are not limited to,

- Table driven assemblers
- Graphical coding support
- Timing analyzer
- Symbolic debugger
- Test instrumentation driver

TEST INSTRUMENTATION

High speed signal processors (using clock rates up to 100 MHz) provide very limited control and visibility for integration testing. This phase of testing requires high speed test instrumentation, physically located with/or in the target processors. Only in this manner can the necessary debugging capabilities (i.e. step, examine, halt on condition, etc.) be provided. Even with this test equipment, the architecture of the processor may mask operations of interest in testing. This problem is most evident in microprogrammed processors which are heavily pipelined at the lowest level. The only effective way to overcome this problem is to include testability (for development purposes) as a design requirement.

Efficient testing of high speed signal processors will require hardware resources. This may take the form of advanced memory architectures and associated software that allow data element tracing through the processing sequence and error trapping. Such a capability would be useful for integration tests to investigate data sensitivities.

SUMMARY

This paper has presented a brief overview of the impact of new hardware technology on software development and testing. Current practice for dealing with the associated problems have been outlines. From this general discussion, it is possible to identify several areas for further work:

- Guidelines for better hardware/software coordination
- Retargetable software support tools
- Programmability guidelines for signal processors
- Real time software testing concepts
- Standard for high speed test instrumentation.

ABOUT THE AUTHORS

Douglas A. Giese, Ph.D. is the manager of the Digital Systems Staff, within the Systems Development Organization of TRW, San Diego. The Digital Systems Staff performs digital systems engineering for signal processing systems, primarily centered around Electronic Warfare applications. Areas of interest include algorithm development, distributed processing and systems support workbenches. Doug received his Ph.D. in E.E. from Vanderbilt University. Since joining TRW, he has been involved with LANDSAT image processing, pattern recognition and signal processing projects. He is currently the manager for the VHSIC EW Brassboard software/algorithm definition work package, and for a graphical signal processor coding technique.

Mr. William P. Jones, Jr. is currently manager of the Systems Development Organization in the Military Systems Operations of TRW, located in San Diego, California. This organization is involved with system/software developments for advanced military applications. He holds a Master's degree from Purdue University in Electrical Engineering. Since joining TRW, he has been involved with several signal processing projects, such as Manual Morse demodulation, software telephone modem for secure voice terminals and automatic signal recognition. In addition, he was the system engineer for TRW's VHSIC Phase III Software Architecture Study which resulted in a specification for a software support system for VHSIC Signal Processors. Most recently he has been involved in the TRW VHSIC Program, in particular the Electronic Warfare Brassboard Development.

IMPACT OF NEW HARDWARE TECHNOLOGY

D.A. GIESE

W.P. JONES,JR.

FEBRUARY 1983

SOFTWARE DEVELOPMENT AND TESTING MUST ACCOMMODATE NEW HARDWARE TECHNOLOGY INSERTIONS

- INTRODUCTION
- SIMULATION
- DEVELOPMENT TOOLS
- INTEGRATION AND TEST FACILITIES
- CONCLUSION

THIS PRESENTATION FOCUSES ON PARTICULAR TYPE OF SOFTWARE DEVELOPMENT

- REAL TIME SIGNAL PROCESSING APPLICATIONS
- USE OF HIGH SPEED PROGRAMMABLE SIGNAL PROCESSOR 100MOPS
- CONCURRENT HARDWARE AND SOFTWARE DESIGN EFFORTS
- HARDWARE TECHNOLOGY INCLUDES:
  - PROCESSOR ARCHITECTURES
  - CHIP DESIGNS

ADVANCED HARDWARE TECHNOLOGY WILL SIGNIFICANTLY IMPROVE OVERALL SYSTEM PERFORMANCE

- IMPROVE SYSTEM PERFORMANCE (I.E. THROUGHPUT)

- REDUCE SOFTWARE COMPLEXITY

  HIGH LEVEL FUNCTIONS IN HARDWARE

- REDUCE PROGRAMMING EFFORTS

  RICH INSTRUCTION REPERTOIRE

  SUFFICIENT PROCESSING RESOURCES

- IMPROVE RELIABILITY AND FAULT TOLERANCE

  BUILT IN DIAGNOSTICS, ETC.

- REDUCE OVERALL SYSTEM LIFE CYCLE COSTS

  PROVIDE GROWTH CAPABILITY

  IMPROVE MAINTENANCE FEATURES

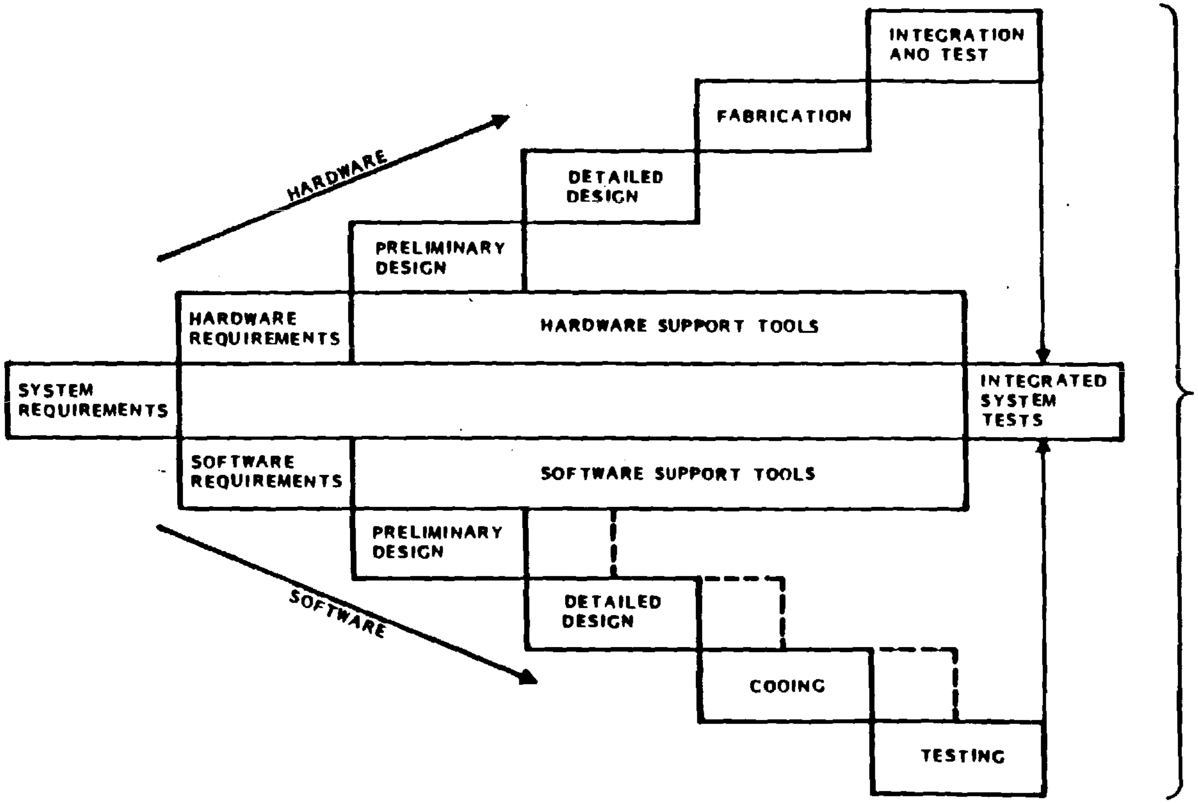
SOFTWARE MATURITY CLOSELY LINKED TO HARDWARE MATURITY

- TOP DOWN DESIGN METHODOLOGY IS CRITICAL
  - "HARDWARE FIRST" DESIGN PHILOSOPHY NOT ADEQUATE
  - HARDWARE/SOFTWARE CANNOT BE PROCURRED INDEPENDENTLY THEN INTEGRATED

- CONCURRENT SOFTWARE/HARDWARE DEVELOPMENT IS RISKY
  - SOFTWARE DESIGN REQUIRES HARDWARE CHARACTERISTICS EARLY IN DESIGN PHASE
  - SUPPORT TOOL DEVELOPMENT LAGS HARDWARE DEVELOPMENT
  - TEST REPRODUCIBILITY ADVERSELY AFFECTED BY HARDWARE EVOLUTION

- SOFTWARE FIXES CANNOT SOLVE HARDWARE DEFICIENCIES

CLOSE COORDINATION BETWEEN HARDWARE AND SOFTWARE IS ESSENTIAL



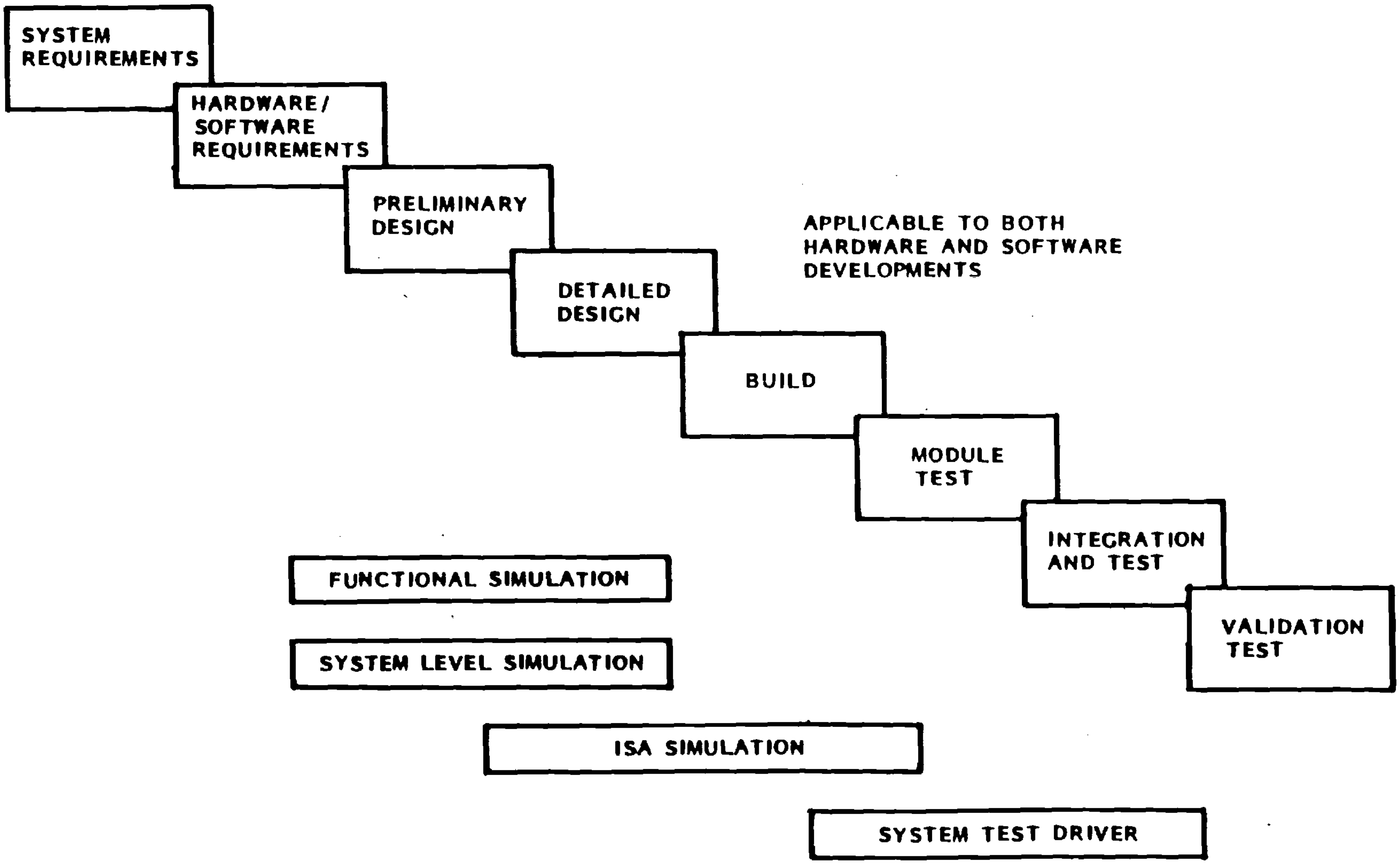
HAROWARE/SOFTWARE INTERACTION

- DESIGN EFFORTS MUST BE PROPERLY PHASEO
- HAROWARE DEFINITION REQUIREO EARLY FOR SOFTWARE SUPPORT TOOLS
- HARDWARE MOOS OFTEN CAUSE COSTLY ITERATIONS IN SOFTWARE DEVELOPMENT.

NEW HARDWARE TECHNOLOGY WILL IMPACT ALL PHASES OF SOFTWARE DEVELOPMENT

PHASE	MAJOR IMPACT	DEVELOPMENT NEED
REQUIREMENTS ANALYSIS	• ALGORITHM DEFINITION TO TAKE ADVANTAGE OF HARDWARE CAPABILITIES • NEW DEVELOPMENT/TEST TOOLS MUST BE SPECIFIED EARLY	• FLEXIBLE SIMULATION TOOLS - FUNCTIONAL - SYSTEM PERFORMANCE
DESIGN ANALYSIS	• NEW COMPUTER ARCHITECTURES • NEED FOR EARLY TOOL DEVELOPMENT	• ABANDON "HARDWARE FIRST" APPROACH - IMPROVE HW/SW COORDINATION • TABLE DRIVEN SUPPORT TOOLS FOR DIFFERENT ARCHITECTURES
CODING AND ANALYSIS	• MORE SOPHISTICATED HARDWARE CAPABILITIES • TIMING CONSTRAINTS FOR MAXIMUM EFFICIENCY	• HOL COMPILERS • RESOURCE MANAGEMENT TOOLS
MODULE TESTING	• HARDWARE NOT GENERALLY, AVAILABLE FOR SOFTWARE TESTING • LARGE VOLUMES OF DATA REQUIRED TO FULLY EXERCISE	• ISA SIMULATORS - TABLE DRIVEN • TEST DRIVERS
INTEGRATION TESTING	• VERY HIGH SPEED LOGIC MAKES DEBUG DIFFICULT WITH HARDWARE • LARGE VOLUME OF DATA ASSOCIATED WITH REAL TIME SOFTWARE TESTING	• SOPHISTICATED INSTRUMENTATION FOR TEST VISIBILITY • DATA GENERATION/REDUCTION TECHNIQUES

MULTI-LEVEL SIMULATION IS NECESSARY WITH NEW HARDWARE TECHNOLOGY



SYSTEM REQUIREMENTS

HARDWARE/ SOFTWARE REQUIREMENTS

PRELIMINARY DESIGN

DETAILED DESIGN

BUILD

MODULE TEST

INTEGRATION AND TEST

VALIDATION TEST

APPLICABLE TO BOTH HARDWARE AND SOFTWARE DEVELOPMENTS

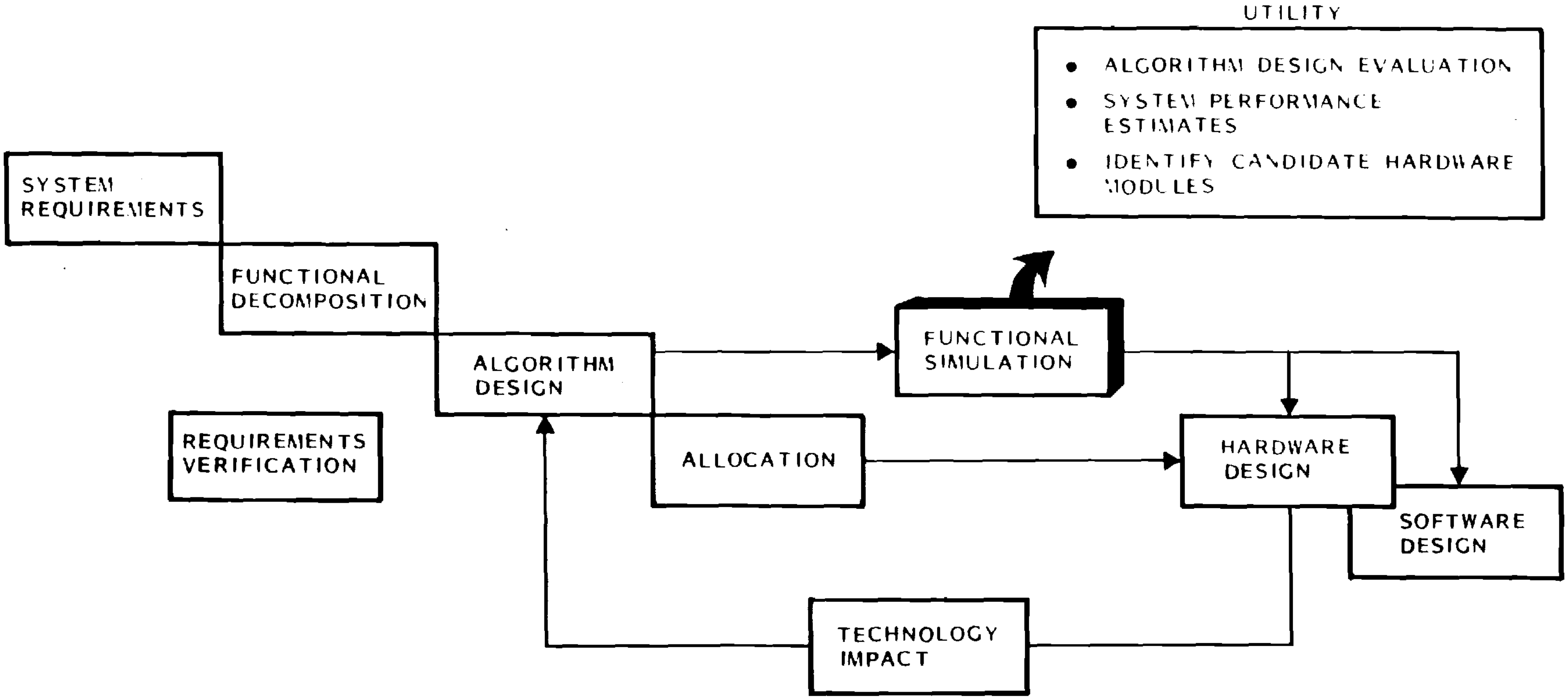
FUNCTIONAL SIMULATION

SYSTEM LEVEL SIMULATION

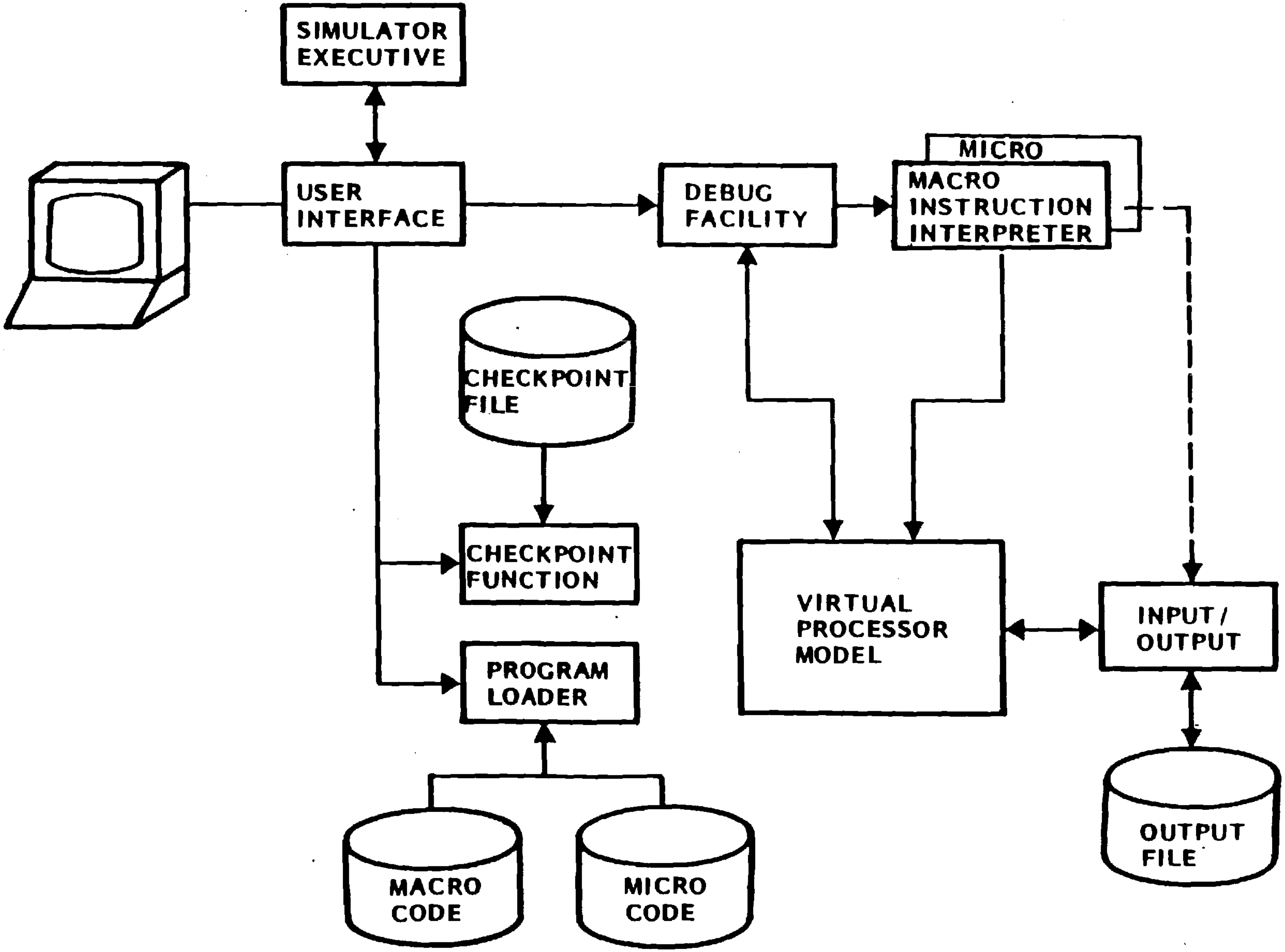
ISA SIMULATION

SYSTEM TEST DRIVER

FUNCTIONAL SIMULATIONS ARE INSTRUMENTAL IN SYSTEM LEVEL TRADE STUDIES



ISA SIMULATION MUST BE RECONFIGURABLE FOR DIFFERENT PROCESSORS



NEW HARDWARE NECESSITATES EFFICIENT METHOD OF DEVELOPING SOFTWARE SUPPORT TOOLS

- SUPPORT TOOLS AFFECTED BY HARDWARE

  HOL COMPILER

  ASSEMBLER

  CRAPHICAL CODING

  TIMING ANALYZER

  SYMBOLIC DEBUGGER

  SYSTEM SIMULATOR

  ISA SIMULATOR

  TEST INSTRUMENTATION

- MUST BE ABLE TO RETARGET SUPPORT TOOLS

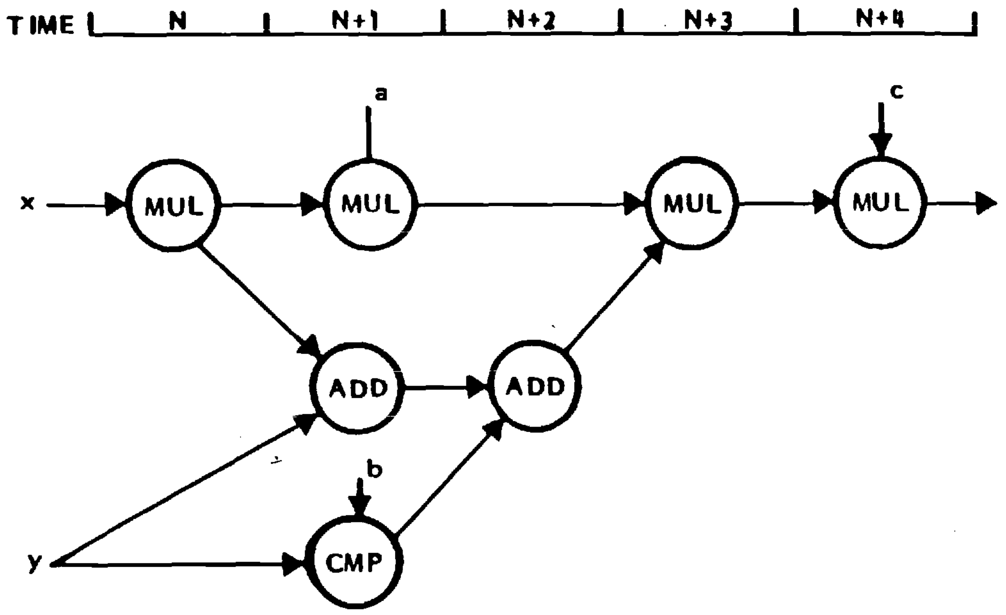
- HARDWARE ARCHITECTURES MUST BE RESTRICTED FOR THIS TO BE FEASIBLE

- NECESSARY HARDWARE TABLES SHOULD BE GENERATED USING HARDWARE DESCRIPTION LANGUAGE

- RETARGETABLE CODE GENERATORS ARE BEING INVESTIGATED - MORE NEEDED

RESOURCE MANAGEMENT TOOLS WILL BE NEEDED TO COPE WITH INCREASED HARDWARE CAPABILITIES

GRAPHICAL CODING

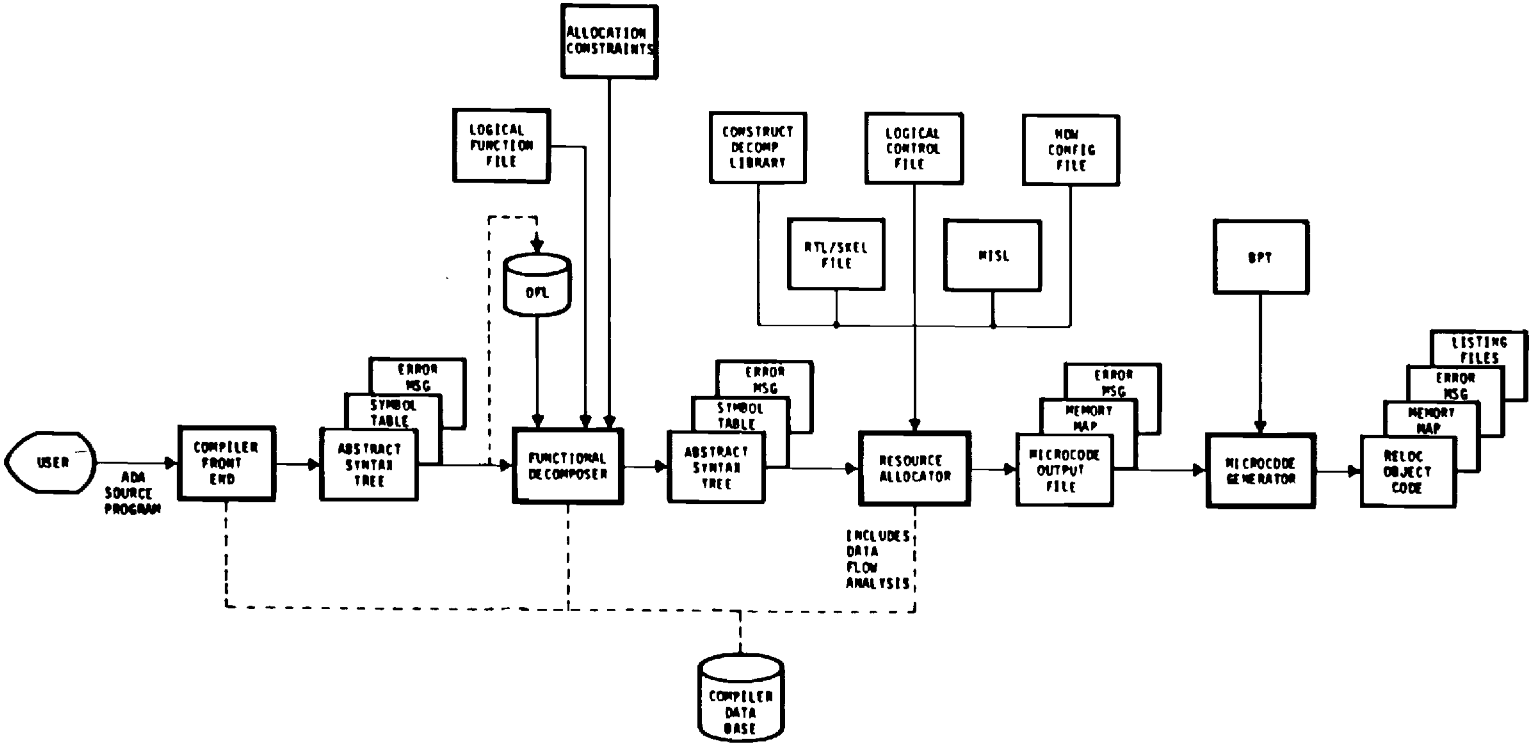


- RESOURCE CONFLICTS AUTOMATICALLY DETECTED
- INTERFACE DIRECTLY TO ASSEMBLER/SIMULATOR
- MODULE TESTING CAN BE PERFORMED IN THIS FORMAT

- SUCH TOOLS REPRESENT INTERIM SOLUTION TO PROBLEM
  - HOL COMPILER FOR SIGNAL PROCESSOR IS ULTIMATE SOLUTION
- TIMING CONSIDERATIONS CRITICAL FOR HIGH PERFORMANCE APPLICATIONS
- DIFFICULT TO FULLY UTILIZE HARDWARE RESOURCES WITHOUT SOPHISTICATED SUPPORT TOOLS

  EX. OVER 500 REGISTERS ACCESSIBLE IN SIGNAL PROCESSOR
- TOOL SWILL IMPROVE CODE PREPARATION AND MODULE TESTING

RETARGETABLE COMPLIER USES MANY HARDWARE DESCRIPTION FILES



DYNAMIC DATA BASE CREATED DURING EACH COMPILATION

INITIALIZED FROM SYSTEM FORMATTER

HARDWARE RESOURCES MUST BE COMMITTED FOR EFFECIVE INTEGRATION TESTING

- TARGET HARDWARE TESTING WILL ALWAYS UNCOVER NEW PROBLEMS

- REAL TIME APPLICATIONS REQUIRE LARGE VOLUMES OF DATA
  - DATA ANALYZER
  - STRESS TESTS

- TEST CASES SHOULD BE IDENTICAL TO SIMULATION TESTING IF FEASIBLE

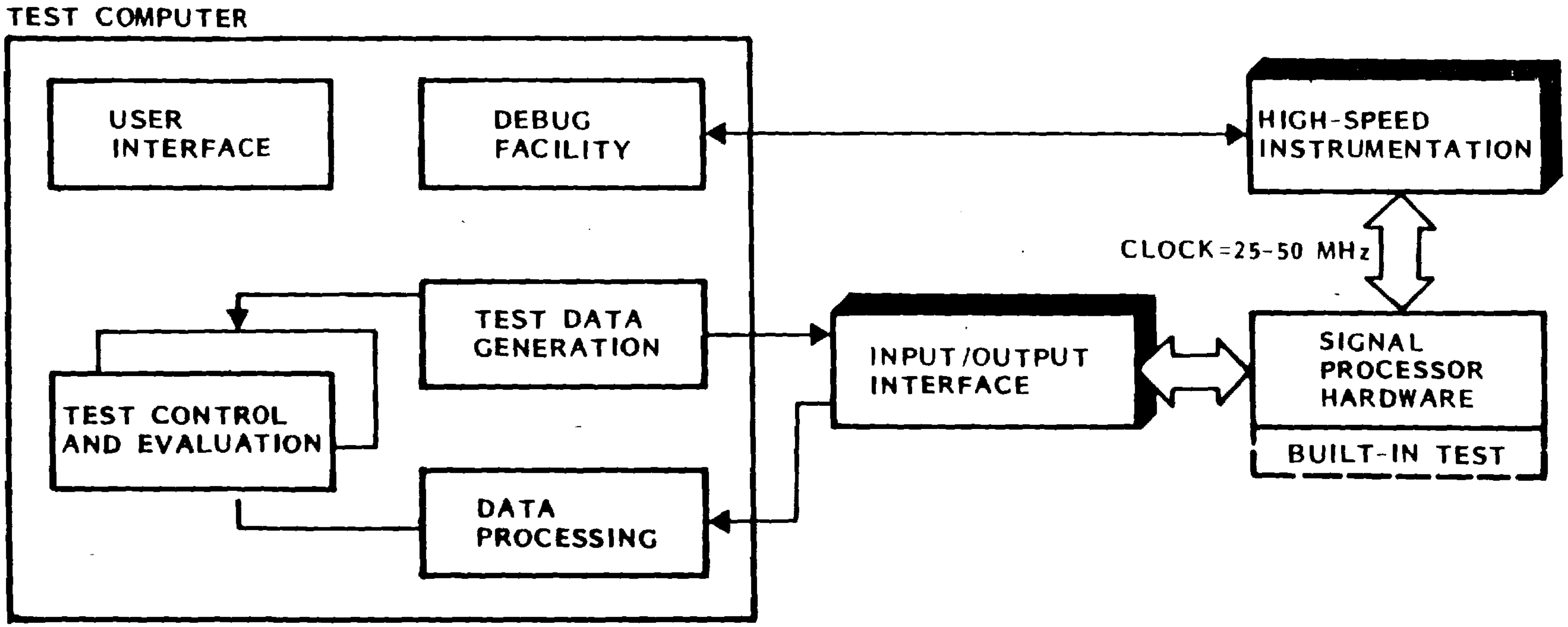
- IDENTICAL USER INTERFACE FOR SIMULATION AND HARDWARE DEBUG FACILITIES REDUCE TRAINING

- DEDICATED HARDWARE RESOURCES CAN EASE TESTING PROBLEM

  EX. DATA TRACEABILITY

DATA	ROUTINE TRACE	STATE

TEST INSTRUMENTATION PROVIDES VISIBILITY NECESSARY FOR INTEGRATED SYSTEM TESTS



ISSUES

- HARDWARE FEATURES TO SUPPORT TESTING
- STANDARDS FOR TEST SETS
- COMMON USER INTERFACE

THERE ARE MANY AREAS FOR FURTHER INVESTIGATION

- GUIDELINES FOR BETTER HARDWARE/SOFTWARE COORDINATION IN NEW SYSTEM DEVELOPMENTS
- RETARGETABLE DEVELOPMENT TOOLS
  - PARTICULARLY HOL COMPILERS
- EXPLOIT SYNERGISM BETWEEN SYSTEM MAINTENANCE REQUIREMENTS AND TEST FEATURES
- PROGRAMMABILITY GUIDELINES FOR NEW SIGNAL PROCESSORS
- STANDARD ISA DEFINITION FOR SIGNAL PROCESSORS
- REAL TIME SOFTWARE TESTING CONCEPTS
- STANDARD FOR TEST INSTRUMENTATION

NEW SOFTWARE DEVELOPMENT CONCEPTS NEEDED TO EXPLOIT HARDWARE TECHNOLOGY

- ADVANCED TECHNOLOGY HAS POTENTIAL TO IMPROVE SOFTWARE DEVELOPMENT
- CONCURRENT HARDWARE DESIGN ADDS SIGNIFICANT RISK TO SOFTWARE DEVELOPMENT
- SIMULATION ALLOWS SYSTEM/SOFTWARE TESTING WITHOUT HARDWARE AVAILABLE
- HIGH EFFICIENCY RETARGETABLE HOL COMPILERS WILL REDUCE SOFTWARE SENSITIVITY TO HARDWARE
- STANDARDS FOR SIGNAL PROCESSORS MUST BE CAREFULLY EVALUATED
    - PERFORMANCE LIMITATIONS
- TESTING MUST BE AN INTEGRAL PART OF DESIGN PROCESS

SOFTWARE TESTING STANDARDS: POLICY AND APPLICATION

Marilyn J. Stewart
Director, Software Evaluation
Booz, Allen & Hamilton
Bethesda, Maryland

ABSTRACT

This paper addresses seven major problems encountered in software testing and discusses their relationship to current and planned software testing standards. Several of these problems have their genesis in that they are not addressed in established software testing standards, while other problems stem from variations among the software testing standards of each military service. A new tri-service software management policy and standard, developed by the Joint Logistics Commanders, will provide a unified approach and correct several shortfalls of the existing standards. Further improvements will be needed to increase our understanding of how to plan and manage an effective software test program.

INTRODUCTION

The purpose of this paper is to discuss Government standards for software testing. Government standards exist in two forms, policy and compliance documents. Policy consists of regulations or instructions directing a Government project manager to conduct an activity, such as software testing, in accordance with an established approach. Policy cannot be referenced in a contract and does not directly govern a contractor's activities or approach, but rather serves as guidance to the Government project manager in managing a contractor. On the other hand, compliance documents are intended to be incorporated into a contract by reference, thereby governing the contractor directly. Compliance documents, such as Government standards and specifications, serve as the usual mechanism for mandating software development activities, documentation approaches, and other software related issues.

In addressing the topic of software testing standards, several aspects of these standards could be considered. It would be useful to know what software testing standards exist and their relationship to software testing approaches currently practiced by the DoD Services and other organizations. Although there is a definate need for a broader understanding of the various approaches used by each of these DoD organizations, the real need of the DoD at present is to consolidate (rather than proliferate) software testing standards. In the process of this consolidation, there is also the need to address several of the common shortfalls in existing software testing standards that often result in difficulties in achieving an effective software testing program.

In the evaluation of software testing approaches proposed by contractors attempting to comply with existing software testing standards, certain issues have surfaced and resurfaced with sufficient regularity to indicate that they represent major problems in existing software testing standards. The following list contains seven common problems in developing a sound software testing approach or implementing an effective software testing program in compliance with existing software testing standards:

(1) Adequate planning and preparation for software testing is difficult in the face of other impending software deadlines.

(2) Determining the proper balance of formal vs. informal testing may be constrained by project resources and varied views of the system's ultimate mission.

(3) There are no generally accepted completeness criteria for software testing.

(4) For contractors dealing with more than one DoD organization, the differing DoD software standards can confuse and complicate the development of a software testing approach.

(5) There is no consensus regarding the extent to which a contractor must test design components in addition to testing software functional requirements.

(6) Although the Government typically witnesses formal software testing, it is often difficult to actually observe detailed software performance in a formal testing environment.

(7) Typical software testing approaches may not meet the Government project manager's needs for establishing meaningful quantitative and demonstrable evaluation criteria for software, as required by DODD 5000.3.[1]

The remainder of this paper addresses how these problems can best be countered using the existing software testing standards or soon-to-be released software standards. In addition, needs for further enhancement of these software testing standards is also discussed.

COMMON SOFTWARE TESTING PROBLEMS

Adequate Planning and Preparation

Software development projects are under constant pressure, with a series of deliverable specifications, documentation, and review meetings that create a seemingly endless chain of impending software deadlines. The software development contractor typically perceives the situation as one in which there is barely sufficient time and money to develop the next increment of requirements or design, with little or nothing to spare for an intensive test planning effort. The potential outcome is a sketchy test plan that is not properly related to the software requirements, then hastily prepared test procedures that have not been checked for adequacy, and finally a testing effort in which tests fail (or cannot be executed) and testing must halt while the test team tracks down the problems that lie in both the test procedures and the software under test. At this point, the Government project manager must decide whether to slip his testing schedule while the contractor remedies the problems in the testing approach or to potentially compromise the testing effort by accepting less than adequate testing.

Unfortunately, no simple remedy exists for this problem. It is, or course, important for the Government project manager to be aware that early delays or compromises in planning the software test program are a definite danger signal. But beyond awareness, the Government project manager must give adequate emphasis to software testing, ensuring that the contractor is fully aware that software testing is not an aspect of the project to be compromised. In evaluating the contractor's early software test planning documents, the Government project manager should ensure that the proposed software test program will adequately serve as a primary means for determining the acceptability of the software.

One of the preferred software testing standards is contained in MIL-STD-1679.[2] This compliance document requires that the contractor accomplish software testing on four levels: module tests, subprogram tests, program performance tests, and (if the developed software is as element of a larger system) system(s) integration tests. Software tests must be defined in a software test plan, then further amplified in software test specifications, and finally detailed in software test procedures containing step-by-step instructions for accomplishing each test.

A similar approach has been adopted by the Joint Logistics Commanders in forthcoming policy [3] and a compliance document.[4] When approved, the Joint Logistics Commanders' approach will entail unit testing, software integration testing, and software performance testing. Software testing may also be included in system integration testing, if necessary. The software test program will be defined in an evolutionary series of documents, including a software test plan, software test descriptions, and software test procedures. Using the Joint Logistics Commanders' approach to developing software, the software test plan will be developed in parallel with software top-level design, the software test descriptions will be developed during the detailed design activity, and the software test procedures will be completed prior to beginning software performance testing. This evolutionary approach meshes perfectly with software development activities and allows the Government project manager to gain visibility into and control the direction of the software test program as it evolves. It also is consistent with top-down design and either top-down or bottom-up testing.

To summarize the standards issues related to planning and preparation for software testing, it is vital that any software testing standard promotes the evolutionary development of the software testing program in parallel with the evolution of the software itself. The Government project manager must also closely review and monitor early software testing activities, especially software test planning, to ensure that the software test program will be effective.

Formal vs. Informal Testing

Each software development project should include both formal and informal testing of the software. Formal testing is conducted by the contractor, with Government witnesses, to demonstrate that the software performs the required functions and is suitable for Government acceptance. The Government must approve all test plans, specifications or descriptions, procedures, and results for formal testing because of the key role that formal testing plays in determining whether the Government will accept the software. The contractor conducts informal testing to determine that the software is operating correctly and that the software is ready for formal testing. Informal testing is subject to Government review, but remains under the control of the contractor.

In the software test plan, the contractor must determine which levels of testing will be formal and which will be informal. Not surprisingly, contractors typically bias this determination towards excessively informal testing while the Government project manager pushes in the direction of increased formal testing. When discussing formal versus informal testing, the contractor seeks to minimize the Government's participation in software testing and the Government seeks to maximize the Government's monitoring of software testing.

The difficulty of determining an optimum balance of formal testing, with Government witnesses, and informal testing, under contractor control, stems from the lack of an accepted definition of the scope of formal testing. Considering this problem as objectively as possible, two general consensus points emerge:

- o The formal testing must demonstrate to the Government that every major software function has been correctly implemented, otherwise the Government lacks demonstrable criteria that the software is acceptable.

- o The contractor must have the latitude to conduct check-out testing informally in order to proceed efficiently through the check-out process. Extensive Government participation in unit-level testing is usually not cost-effective for the Government or for the contractor.

These two consensus points are useful in bounding formal and informal testing. Formal testing typically includes acceptance testing but typically does not include check-out testing.

The proper balance point for versus informal testing is project specific and lies between these two boundary points. Formal testing should also include demonstrations of all critical intra-system or inter-system interfaces. In addition, formal testing must demonstrate the correct operation of any critical components of the software design. Testing of these critical aspects of the software may require that a portion of the intermediate-level testing be conducted formally. The amount of formal intermediate-level testing varies from project to project, depending on the criticality of the system, the software design, and the testing approach.

The Joint Logistics Commanders have adopted the software test philosophy summarized above for properly balancing formal versus informal testing. Of the three software testing levels (unit testing, software integration testing, and software performance testing), unit testing is conducted as informal testing and software performance testing is conducted as formal testing. Additional formal testing is conducted as part of software integration testing to confirm correct operation of critical aspects of the software. The scope of the formal and informal testing programs is defined in the software test plan, which is reviewed and approved by the Government project manager to ensure that the contractor has determined the proper balance of formal versus informal testing.

Test Completeness Criteria

There is an often repeated maxim concerning test completeness, stating that testing can reveal the existence of an error but can never demonstrate that software is error-free. Early attempts to overcome this inherent limitation of software testing led to a concept of software logic path testing, with the further conclusion that it is impossible to test all the paths through any non-trivial program. Experimentation with test data selection also demonstrated that it is difficult to determine "meaningful" test data for non-trivial programs.

In the meantime, many contractors and Government project managers have been faced with the problem of determining whether their test approaches, plans, and procedures would completely test developed software. The issue of test completeness and test acceptance criteria became a sufficiently pressing issue to be selected as one of four major problem areas to be discussed at the first Joint Logistics Commanders' Software Workshop in 1979.[5]

At that time, the best example of test completeness criteria was contained in TADSTAND 9,[6] developed by the Naval Material Command. TADSTAND 9 was primarily concerned with the final software testing to be conducted prior to Government acceptance rather than incremental software testing at meaningful events during software development. Since that time, TADSTAND 9 has been replaced by TADSTAND E[7] and MIL-STD-1679,[2] which requires an incremental software testing approach culminating in software quality testing prior to initial operational use of the system.

The need for test completeness criteria presently remains an unsolved problem in software testing. It appears unlikely that a general set of completeness criteria for software testing, suitable for all systems, can be incorporated into a general software standard. Government project managers will continue to develop project-specific software acceptance criteria, including test completeness criteria. To do so effectively, Government project managers need guidance, probably in the format of a military guidebook containing instructions on developing and applying software test completeness criteria within the context of the software development and acceptance process.

Diverse DoD Software Standards

Let us consider the hypothetical, but not unrealistic, situation that arises when a software contractor that achieved an outstanding software test program for one DoD service is awarded a software contract by another service. For example, the contractor's competency inconducting Preliminary Qualification Tests and Formal Qualification Tests (Air Force) may not easily map into competency in Software Quality Test (Navy). Contractors are often faced with a learning curve as a result of the need to adopt a different approach and service-unique terminology.

Differing approaches and terminology are also a hinderance to joint service programs. Each participating project manager is constantly confronted with service-unique terminology and finds himself dealing with questions like, "Is DT II (Army) like FQT (Air Force) or is it more like TECHEVAL (Navy)?" Such problems of understanding arise in communicating with project managers from other services and in communicating joint project information back to one's own service. Because there are very few individuals who are fully conversant in the approaches and terminology of all three services, joint service projects often select the software policy and standards of one service to be adopted by all services participating in the project.

An initial step towards a solution to the problem of service-unique software policy and standards, which causes service-unique approaches and terminology, has been taken by the Joint Logistics Commanders. The cornerstone of the Joint Logistics Commanders' efforts has been the formulation of a software management policy(3) that is applicable to all of the Joint Logistics Commanders' organizations: Air Force Systems Command, Air Force Logistics Command, Naval Material Development Command, and Army Materiel Development and Readiness Command. As a companion to the tri-service software management policy, the Joint Logistics Commanders have also developed a compliance document for use in software development contracts.(4) These two documents represent a tremendous stride in adopting a common set of software terminology and a common software management approach that can be easily utilized by each service.

Approval of the policy and compliance document for use by Joint Logistics Commanders' organizations will mitigate the problem of diverse software standards for much, but not all, of the DoD community responsible for developing embedded computer systems. It is hoped that, after initial release, the Joint Logistics Commanders' policy and compliance document are adopted by DoD for use on all projects that develop or modify software for embedded computer systems.

Testing Against Requirements vs. Design

A basic dilemma the software test planner must resolve is whether the software testing criteria should be derived solely from the software requirements or also from the software structure embodied in the design. Most embedded computer systems automate mission critical functions of a weapon system or other defense system, hence software testing criteria have historically focused on software requirements.

In extremely critical applications (for example, nuclear weapons or intelligence applications), it may also be necessary to demonstrate that the system is invulnerable to certain classes of failures or threats. In addition to the normal requirements-based testing, this type of system typically undergoes a set of design-based tests to demonstrate that the access control logic is robust or that software algorithms operate correctly on extreme or boundary values. Such test cases cannot effectively be devised without an in-depth understanding of the design.

Upon further consideration, it is logical to extend this rationale to more typical systems containing embedded software. For example, the software requirements for a typical fire control system seldom include explicit requirements for a software executive that maintains control over the fire control system. From the perspective of the software requirements, the need for an executive is transparent to the system functions to be performed by the software. From the perspective of the software designer, a software executive may be essential to achieving a timely response to operator commands. If software testing of such a system were solely based on software requirements, the software executive would be tested only as a black box that enabled other system functions. Any weaknesses in the executive would only be detected if the requirements-based testing scenarios included the right (i.e., failure-prone) set of commands. In design-based testing, a software component such as an executive would be tested for scenarios that are likely to stress the limits of its operation, thereby greatly increasing the changes of detecting an error in that component.

The need for requirements-based testing remains clear--requirements should always serve as the primary basis for determining the acceptability of the software. In addition to requirements-based testing, certain components of the software should be subjected to design-based testing, particularly when their functioning is transparent to the software requirements but critical to achieving major system functions.

Witnessing Formal Testing

During the test planning phase of the software development cycle, the Government project manager carefully reviews the contractor's approach to testing to ensure that there is an adequate amount of formal testing. As a primary basis for determining the acceptability of the developed software, formal testing is witnessed by the Government project manager (or his designated representative). It often is necessary for the Government project manager to commit significant investments of project resources, including travel to the test site and staff to witness all formal tests, from start to finish.

Now, consider the extent to which the Government project manager can determine the acceptability of the software by observing data available to him during the formal testing process. There are several opportunities for misunderstanding the formal tests being witnessed. If test procedures have not been maintained under the proper change control, the test procedures are often "red-lined" with last minute changes prior to commencing a test. If test software, such as a simulator, is used to demonstrate a software function, and the test is less than completely successful, it is often difficult to trace the problem's cause to the simulator or the actual software.

Even if we assume that there are no problems in the test procedures or in the test software, it is difficult for the Government project manager to observe the software's performance in sufficient detail to develop a valid determination of the software's acceptability. For example, is it possible to positively observe a 100 millisecond delay in software response time by witnessing system execution? Is it possible to positively observe that a velocity calculation is not accurate to within a one foot per second requirement? Many of the technical characteristics of embedded software are indeed extremely difficult to evaluate by observing formal testing and reviewing formal testing results.

It is essential, therefore, that the Government project manager employ additional mechanisms for determining the adequacy of software. It is strongly recommended that the formal testing process be viewed as the culmination of a series of evaluations, all of which contribute to determining whether the software is suitable for acceptance by the Government. Use of the activities, products, reviews, and baselines of the software development cycle defined by the Joint Logistics Commanders[3,4] will lead to a more evolutionary software development approach with multiple opportunities for evaluation by the Government project manager.

Specifically related to improving the Government project manager's visibility into software testing itself, it is further recommended that the Government and contractor jointly determine a suitable level of participation of the Government project manager during informal software testing. This participation may consist of attending the walk-throughs of designated critical software modules or may entail monitoring the informal testing approach and results for certain high-risk software functions.

The Government project manager should firmly adhere to the requirement to develop a test plan, descriptions, and procedures and to conduct a Test Readiness Review prior to beginning formal testing. The Test Readiness Review should assess the contractor's software test procedures and readiness to commence the formal software testing program. Results of informal testing should also be available for Government review at the Test Readiness Review.

Quantitative and Demonstrable Test Criteria

DoD software testing policy requires that the Government project manager employ quantitative and demonstrable test criteria to demonstrate that the system, including embedded software, is suitable for use in the operational environment.[1] Unfortuantely, most compliance documents that govern a contractor's implementation of software testing do not levy a corresponding requirement on the contractor to maximize the use of quantitative test methods in place of qualitative or subjective test criteria.

Certain quantitative testing requirements for software are found in MIL-STD-1679, specifically in the requirements for software quality test.[2] This compliance document sets firm requirements for a duration test, in which the software must successfully operate, continuously, for a period equal to the anticipated duration of a system mission (or a maximum of 25 hours for systems that will be in continuous operation). In addition, quantitative error and patch limits are set, along with a requirement for repeating any tests that detect errors exceeding predetermined severity thresholds. Finally, stress testing must constitute one-third of the duration of the software quality test.

Although these quantitative testing criteria are among the most definitive available, the Government project manager must employ further careful judgment in their application. First, meeting these quantitative test criteria does not constitute an assurance of software acceptability and should not be viewed as a guarantee of project success. Second, the most meaningful quantitative and demonstrable testing criteria for software will reflect project-specific conerns, such as mission profiles and system risk areas. The Government prject manager retains the responsibility for ensuring that the contractor develops a valid testing approach that embodies quantitative and demonstrable software test criteria most meaningful in the context of a particular project.

SUMMARY

Software testing standards are, for the greatest part, defined within the policy and compliance documents for developing software. The seven major problem areas discussed above have persisted despite the application of existing software testing standards. A strong potential for improvement is seen in the Joint Logistics Commanders software management policy and standards. Of particular importance are the improved software development cycle, including a more evolutionary approach to software testing, and the consolidation of software management guidance into a single approach for use by all services. Additional improvements are also found in specific areas such as formal versus informal testing and clarification of the need for design-based testing. Even with this improved software management policy and standards, the Government project manager will retain significant responsibility for ensuring the adequacy of the software testing program. The most difficult software testing issues facing the Government project manager relate to developing a test approach that embodies system objectives in the definition of software testing completeness criteria and quantitative and demonstrable software performance criteria.

REFERENCES

(1) DoD Directive 5000.3, Test and Evaluation, December 26, 1979.

(2) MIL-STD-1679, Weapon System Software Development, 1 December 1978.

(3) Joint Regulation, Management of Computer Resources in Defense Systems, (in coordination).

(4) MIL-STD-SDS, Defense System Software Development, (in coordination).

(5) Proceedings of the Joint Logistics Commanders Joint Policy Coordinating Group on Computer Resource Management, Computer Software Management Subgroup Software Workshop, 21 August 1979.

(6) TADSTAND 9, Software Quality Testing Criteria Standard for Tactical Digital Systems, 18 August 1978.

(7) TADSTAND E, Software Development, Documentation, and Testing Policy for Navy Mission Critical Systems, 25 May 1982.

ABOUT THE AUTHOR

MARILYN J. STEWART is the Director of Software Evaluation within the Defense Technology Division of Booz, Allen & Hamilton. She manages the independent software analysis and test evaluation of communications, tracking, and surveillance applications. Previously, she was Manager of the Systems Evaluation Department of Logicon, where she managed independent verification and validation for strategic missile and range safety programs. She also directed a software technology program that developed advanced tools and methodology for software analysis and testing. Ms. Stewart has managed independent testing and evaluation programs that have totalled over 150 staff-years of effort and has developed tri-service policy for managing the development of software for defense systems. Ms. Stewart holds a Bachelor's degree in mathematics from California State University at San Diego and is a member of ACM and IEEE.

SOFTWARE QUALITY ASSURANCE AND ACQUISITION POLICY

Captain William P. Nelson, USAF

Electronic Systems Division

Hanscom AFB, MA

ABSTRACT

A brief review of current Air Force Systems Command (AFSC) and Electronic Systems Division (ESD) software quality assurance requirements is given. Current policy and guidance changes being considered by an AFSC work group are presented. An ESD program for reviewing contractual implementations of software quality assurance is described and results of seven such reviews are presented. The conclusion drawn is that contractual implementations tend to be weak and that this is caused by a weak specification. The implication is that current and future Department of Defense efforts should concentrate on developing a clearer standard for software quality assurance requirements. A need to provide a training program to alleviate a perceived shortage of software quality assurance personnel is also noted.

INTRODUCTION

Software quality assurance is not a new concept. What does appear to be new is the increased amount of attention being paid to it, and the number of different concepts and techniques that now have a software quality assurance label attached to them. This paper will not attempt to delineate all the different approaches possible or recommend one particular approach over another. What will be presented is one approach to requiring a software quality assurance program on contracts for defense systems with software. Some results of this approach will also be presented.

For the purpose of this paper, software quality assurance will be discussed in terms of the requirements of the military specificaton MIL-S-52779A, Software Quality Assurance Program Requirements. While this is not the only military specification or standard that addresses software oulaity, it is the primary one used on contracts at the Electronic Systems Division. It is also the only one for software specifically called out in current Air Force Systems Command quality assurance regulations. Many different specifications and standards affect the total quality of both the software and the system, MIL-S-52779A is the primary one used to require a contractor to implement a software quality assurance program.

Four areas will be discussed. The first will be a brief overview of the current policies for requiring software quality assurance on contracts overseen by Air Force Systems Command and the Electronic Systems Division. This will be followed by a discussion of newly evolving policies. The third major topic will be a view of how successful some contractors have been in implementing Electronic Systems Division's expectations of MIL-S-52779A. This will be based on reviews performed by the technical staff. Finally, some opinions on the underlying reasons for contractor implementations will be given. How these should affect future Department of Defense efforts in the area of software quality assurance will also be briefly discussed.

CURRENT POLICY

Air Force Systems Command Policy. The requirements for applying software quality assurance is stated in AFSC Regulation 74-1, Quality Assurance Program, with guidance on application given in AFSC Pamphlet 74-4, Guide for Quality Assurance Managers. The 29 July 1980 version of the regulation requires that "For contracts with significant computer software development, MIL-S-52779A, Software Quality Assurance Program Requirements, will be a contractual requirement but may be tailored for individual program needs" (4:4). Chapter 18 of the pamphlet gives very high-level guidance on the use of MIL-S-52779A. The 22 July 1980 version suggests that the specification is "applicable to complex software programs" and that it "should also be considered for use with nondeliverable software which is used to manufacture, inspect, or test deliverable contract items," but notes that the specification must be selectively applied in such cases (5:33). The detailed implementaton of MIL-S-52779A has basically been left to the individual divisions, centers, and ranges of the command. Chapter 18 of AFSCP 74-4 references the regulations and guidance used by the different organizatons (5:33). This approach has changed in the last year to one of providing more detailed command-wide guidance based on the consensus of a command working group, as will be discussed in more detail later.

Electronic Systems Division (ESD) Policy. At ESD, policy has been to have a software quality assurance program required on all contracts involving software development. Prior to the issuance of the "A" version of the specification in 1979, MIL-S-52779(AD), dated 5 April 1974, was used. An ESD Software Quality Assurance Guidebook was developed to aid in its application (6). While this guidebook is now outdated, the differences between the "A" and "AD" versions of the specification are relatively minor so that the guidebook is still used as a reference document. The primary document used for interpreting MIL-S-52779A has become DLAH 8250.1, Evaluation of a Contractor's Software

Quality Assurance Program, dated May 1981. This handbook has been republished as MIL-HDBK-334, same title, which is the actual document in current use. Updated guidance on application has been prepared by the division technical staff for publication in a general software acquisition guide for the division.

Contractual Application. In practice, the typical contractual application of software quality assurance on an ESD contract has been a single paragraph in the statement of work requiring the contractor to implement a software quality assurance program in accordance with MIL-S-52779A. The software quality assurance (SQA) plan required by the specification is usually asked for as a deliverable. The SQA Plan is usually either included in the computer program development plan (which is a complete description of the contractor's development and management methodology for software), required as a separate deliverable, or included in another plan (typcially an overall company quality assurance plan). It should be noted that the first of these methods, inclusion in a computer program development plan, is allowed by AFSCR 74-1 (4:4) and recommended by AFSCP 74-4 (5:33). Although the specification itself states that it is applicable to all software, both deliverable and nondeliverable (1:1), it is usually only applied by ESD program offices to deliverable software. The primary exception is that it is expected that the contractor will apply it to software tools used to develop other software, and to any software used to manage the configuration of deliverable or nondeliverable software (e.g. compilers, operating systems, automated computer program library systems, or the like). The responsiveness of contractor implementations has been mixed. Findings concerning some implementations will be discussed later.

EVOLVING POLICY

Within approximately the last year, Air Force Systems Command has begun to take a more comprehensive look at software quality assurance. Specifically, the Air Force Systems Command Quality Assurance Council directed the formation of a Software Quality Assurance Subgroup. All command activities involved in software acquisition are invited to participate. The subgroup met three times in 1982 with a fourth meeting planned for March of 1983. The result has been a valuable exchange on applying software quality assurance (SQA) to defense systems contracts, and the evolution of more comprehensive command policy and guidance on SQA. The following paragraph will highlight the issues studied by the subgroup. (This summarization is the interpretation by the author of discussions at the meetings and of the published minutes. It should not be considered the official positions, recommendations, or actions of the subgroup or the Air Force Systems Command.)

Standard Terminology

One of the first issues addressed by the group was the need for a standard terminology for software within the quality assurance community. While standard definitions existed within computer resource related regulations for most software or computer terms, the consensus was that these definitions were not widely known or used by quality assurance personnel. Using the Defense Acquisition Regulations and current Department of Defense and command regulations as a base, the subgroup agreed upon definitions for such terms as products, technical data, computer, computer software, computer program, embedded computer resources, and firmware. While there is nothing new in the definitions, and in fact most are drawn from existing regulations, a standard usage of the terms within the Air Force Systems Command quality assurance community is now agreed on. The definitions examined will be submitted for inclusion in MIL-STD-109B, Quality Assurance Terms and Definitions, on its next update.

Use and Deliverability of Software Quality Assurance Plans

Another area addressed by the subgroup at its first meeting was the government's handling of contractor software quality assurance plans. The two basic issues were whether plans should be delivered to the government, and whether the government should approve software quality assurance plans, delivered or not. Air Force Systems Command policy has been that no quality assurance plan should be approved by the government (5:3). Additionally, the contractor is seldom required to deliver hardware quality assurance plans. Instead, the program office depends on the cognizant contract administration office to review contractor quality assurance efforts and use disapproval of programs when necessary. On the other hand, a program office almost always has a contractor deliver a computer program development plan documenting his technical and management approach to software. The plan is often required as part of contractor's proposal and is always subject to approval by the government. It is made contractually binding by either including it in the compliance document section of a statement of work or by including a task in the statement of work that requires the contractor to prepare, update, and follow the computer program development plan (CPDP). As previously noted, the software quality assurance plan can be included in the CPDP. This causes a policy conflict and a departure from the practice of not requiring quality assurance plans for delivery. While there was support and rationale for many approaches among subgroup members, the consensus was that at this time there does not appear to be a preferred method. It was decided to clarify the policy statement on delivery and approval of plans in the governing regulation, but to leave the actual decision to approve or disapprove the software quality assurance plan to the individual program offices. Approval should be used sparingly and only when carefully defined so as to avoid conflict with other requirements of the contract, with provisions of Defense Acquisition Regulations, and with other quality standards. This position should be included in an update to AFSCR 74-1 expected in early 1983.

Application of MIL-S-52779A

Statement of Command Requirements. A position of some subgroup members was that the command requirement for software quality assurance on contracts was vague. As noted earlier, the governing regulation requires MIL-S-52779A for all contracts with "significant" software development. The pamphlet that gives further guidance uses the term "complex software" as the decision driver. The group consensus was that these terms were too subjective to guarantee any standard application of the specification It was agreed that generally MIL-S-52779A should be applied to any contract with deliverable software or with nondeliverable software used in the acceptance or quality testing of deliverable items. The exact policy change will be contained in the expected early 1983 update of AFSCR 74-1.

Application to Non-Deliverable Software. Additionally, the guidance on what parts of MIL-S-52779A should be made applicable to nondeliverable software was felt to be weak. Guidance was therefore developed for tailoring the application of the specification to nondeliverable software depending on which of five possible categories a given item of nondeliverable software fell into. The definitions of the categories (which include two categories for automated test systems as well as categories for support tools, computer aided design, and computer aided manufacturing software) and a table showing suggested tailoring will be included in a revision to AFSCP 74-4 to be released in early 1983. (It should be noted that primary credit for writing the guidance belongs to the Air Force Contract Management Division.)

Contractual Tasking. A third area in applying MIL-S-52779A addressed by the subgroup was how to best express requirements for it in the statement of work. A related issue was how to strengthen areas of the specification perceived to be weak. The Electronic Systems Division technical staff has drafted a model tasking paragraph for a statement of work. The tasking is a short paragraph that requires a contractor to plan, develop, and implement a software quality assurance program in accordance with a tailored application of MIL-S-52779A. While this requirement may seem self-evident from putting the specification on contract, statements of work have been written that require a contractor to write a plan without requiring him to implement it or update it. Drafting of suggested revisions to the specification will probably be overcome by other Department of Defense efforts, notably the expected release in 1983 of a draft MIL-STD-SQAM, Software Quality Assessment and Measurement, produced under the auspices of the Joint Logistics Commanders. Work on the draft statement of work tasking will continue so that it can be used in the interim. Final action will depend on member comments, but it is likely that the result will be added to AFSCP 74-4 as additional guidance in using MIL-S-52779A. The subgroup will probably try to present a consensus opinion on the MIL-STD-SQAM when it is available for review.

Government Roles and Responsibilites

Organizational Roles. A problem expressed by some subgroup members was reaching agreement on the roles and responsibilities of the different government activities typically involved in software acquisition. The question is still being addressed. Each subgroup member is working on a statment of the roles and responsibilities of his organization as viewed internally. The expected result of this effort is not a redefinition of roles, but a better understanding within the command of how the different organizations support acquisition of software within defense systems acquired by the command buying agencies.

Contract Administration Office Support. An opinion expressed by some members was that program offices were generally unaware of what contract administration offices could do for them in the area of software quality assurance. This area was already being at least partially addressed by the Air Force Contract Management Division. They had been working on a checklist of possible items to be included in a Memorandum of Agreement or a Quality Letter of Instruction written between a program office and an Air Force Plant Management Office. This checklist has been through several draft iterations and is not yet in final form. However, the draft versions have been used on several occasions at the Electronic Systems Division as the starting point in requesting contract monitoring support for software by not only Air Force plant personnel, but also with Defense Contract Administration Office Personnel. When final concurrance on the checklist is reached it will probably be included in AFSCP 74-4.

Application of Other Quality Assurance Standards to Software

A final major area addressed by the subgroup is the applicability of other military quality assurance standards to software. Specifically, the subgroup has addressed the applicability of MIL-Q-9858A, Quality Program Requirements, MIL-STD-1535A, Supplier Quality Assurance Program Requirements, and MIL-STD-1520B, Corrective Action and Disposition System for Nonconforming Material.

MIL-Q-9858A. In general, MIL-Q-9858A is considered to apply to software as the "umbrella" under which all other military quality assurance standards and specifications, including MIL-S-52779A, fall. Its provisions are also considered to apply to software in the absence of MIL-S-52779A on contract.

MIL-STD-1535A. The subgroup concluded that the subcontractor control portions of MIL-S-52779A were comparatively weak, but at any rate did not conflict with MIL-STD-1535A requirements. MIL-STD-1535A was considered to be general enough to be applied to software without revision, but in order to strengthen its application to software some revision was felt to be necessary. In drafts so far, this consists primarily of adding a few software peculiar items to the lists of products and services covered by various paragraphs of the standard, an example addition being software developement facilities. This action is currently in progress and it is expected a final draft revision will be ready for consideration at the next meeting of the subgroup.

MIL-STD-1520B. The subgroup members could not reach a decision concerning the applicability of MIL-STD-1520B to software. While there was general agreement that some revision would be required, the nature of the revision could not be agreed upon. Discussion tended to revolve around reservations by some members that basic concepts necessary for implementation were not well defined enough across industry to permit implementation of a standard for correction and disposition of nonconforming software. Examples of concepts felt to be ill-defined in the current state-of-the-art are exactly what constitutes a software error, how to define nonconforming software, what a suitable error classification system for software should consist of, and whether or not the provisions of current software quality assurance and computer resource contract requirements are sufficient for error correction and reporting needs. It was decided that a task proposal should be submitted to the Air Force program office for Computer Resources Management Technology to contract for a more detailed study of the issues. A final position on this issue is not expected in the near future.

EXPERIENCE WITH IMPLEMENTATIONS OF MIL-S-52779A

Program for Assessing Software Quality Assurance Implementations

Background. Policy on software quality assurance at the Electronic Systems Division has been evolutionary. Current policy is based on the concept that the ultimate quality of a delivered system, while the responsibility of a program manager, can only be implemented or acheived by the organization responsible for physically building the system. The approach of the technical staff has therefore been one of making sure that appropriate technical and management requirements are on contract, and then assisting the program offices in monitoring the contractor's implementation and progress. While this is accomplished in many different ways, one form of assistance is especially pertinent to this discussion. This is the review of contractor's software quality program implementation by the technical staff. Since the results to be presented shortly are the product of these reviews, the method of conducting them will be briefly discussed.

Review Purpose and Product. The purpose of the review is not to provide a passing or failing mark for a contractor's software quality assurance (SQA) program, but to give a government program manager an independent view of how well the contractor's implementation of SQA is meeting contract requirements. The review is typically performed prior to prelimary design review in order to help insure that the program is on the right track as soon as possible. A report is produced from the review that goes only to the program manager. It is intended to provide him with information and associated recommendations, and is not directive in nature. Any decisions on resultant actions are solely his. The technical staff is available, however, to provide further assistance either in implementing recommendations or by providing follow-up reviews.

Review Preparation. Reviews typically follow the same pattern. Computer resource personnel assigned to the technical staff meet with program office personnel to discuss the general objectives of the review. The purpose of the initial meetings is to find out if the program office has any specific concerns relative to software or software quality assurance, and to coordinate a date for an in-plant review of the contrator's efforts. The review team (usually two people) then review documents related to the contractor's efforts. An important part of this review is the analysis of the contractor's computer program development plan (CPDP). As noted under the policy discussion, the CPDP is expected to be a detailed presentation of the contractor's methods for developing software and managing the associated efforts. This is important because quality will be built into the software based on the development method and environment. It is also expected that any SQA program will be closely tied to development techniques and methods. The document containing software quality assurance planning is also reviewed if this data is not contained in the CPDP. The review team also examines any delivered software products (primarily specifications given the timing of the review in the development life cycle). These documents are not reviewed for the purpose of providing a technical assessment, but to get a general view of the development effort and the system as a whole. The meetings with the program office and the document reviews are followed by an in-plant visit to the contractor's software development facility.

In-Plant Visit. The in-plant visit is actually the shortest part of the review, but is also the most intensive. A date is coordinated with the contractor, through the program office, at least two weeks in advance of the visit. To shorten the time necessary for the visit, the review team provides questions in advance to the contractor based on the team's review of plans and documents. The intent is to let the contractor know what areas the team wishes to discuss and to allow him adequate time to prepare while trying to prevent any negative impact on his operations. The visit itself is usally two days long. On the first day, the team meets with the lead software development manager, the lead software quality assurance manager, and some of their key personnel to discuss the software development environment and software quality assurance program. The set of prepared questions previously sent to the contractor forms the initial base of the discussion. The second day is used to examine the actual implementation of the contractor's software quality assurance program and to clarify any additional questions the team may have as a result of discussions. At the end of the review, the team will usually give the contractor an overview of their general findings. Normally, a presentation of detailed findings is left to the government program manager. A final aspect of the in-plant review is that the team will meet with the cognizant government contract administration office personnel for the contract. These personnel will also have been informed of the visit and will be asked to participate in all meetings with the contractor. Additionally, the team will discuss the capability of the in-plant government organization to monitor the contractor's software development and quality assurance efforts.

Review Close-out. The final result of the review is a written report of the findings to the program manager. Often, an informal briefing on the major findings is also given. The report covers the good points of the contractor's program as well as any areas the review team feels could be improved. A second major portion of the report will cover the team's meeting with contract administration office personnel and their capabilities relative to software. For any areas in which improvement is suggested, either contractor implementation or government monitoring, the team will make recommendations. While the report closes out the review, the technical staff will assist in implementing recommendations or providing follow-up reviews on request.

Basis of findings.

The findings presented below are based on review of seven contractor software quality assurance programs, as well as several associated follow-up reviews on some of these same programs. No attempt is made to relate the findings to size of development effort, cost of software being developed, or similiar factors. Each review was guided by the contractual requirements on that particular contract. Further, the teams did not attempt to give a pass/fail mark, as discussed above. The primary question being answered in each case was "How well does this implementation meet contractual requirements for software quality assurance and can it be improved to obtain maximum effectiveness?" In all cases, the basis of the contractual requirements was MIL-S-52779A or the earlier MIL-S-52779(AD) version.

Inadequacies of SQA Implementations

SQA Manning. In five of seven reviews, the manning assigned to software quality assurance was deemed inadequate. This judgement was not based on any magic number or ratio, but on the perceived ability of the assigned personnel to implement contractual SQA requirements in a manner appropriate to the size of the development effort. It should be noted that the problem was sometimes caused by factors not under the control of the contractor, such as having personnel leave the company. More commonly it seemed due to a failure to seriously analyze and schedule SQA activities. An early undertaking of such planning would have demonstrated a shortage of manpower. In one case, however, the contractor was inexperienced in the application of MIL-S-52779A and admitted to underestimating the workload involved in its implementation.

Corrective Action. In four programs, the involvement of software quality assurance (SQA) personnel in corrective action was inadequate. Typical shortcomings were no involvement of SQA personnel at all, no formal tracking procedures, or no independent verification that errors were corrected properly. In the latter case, the only procedure for assuring correction was the action of the individual programmer to effect the change. The expectation of the review teams was that quality assurance would play a prime role in corrective action, especially in verifying that any given software error had been corrected without causing more errors.

SQA Procedures. All programs but one made an attempt to delineate the procedures to be used by SQA personnel. Unfortunately, in four programs there were no written directions or checklists to be used in accomplishing the procedures. In such cases, SQA personnel had a general idea of what they wanted to accomplish but could give no specifics of how they would do it. The review teams felt that without some form of written guide or checklist, there would be no uniformity or consistency in the application of software quality assurance.

SQA Involvement in Reviews and Walkthroughs. In four of the examined programs, SQA involvement in reviews and walkthroughs could have been improved. The review teams did not expect to find SQA participation in every review and walkthrough (especially since, if taken at all levels, there could conceivably be several hundred total). They did expect to find SQA involvement in all formal reviews (e.g. preliminary and critical design reviews) and some sort of sampling of less formal reviews and walkthroughs. While the worst case was no attendance by SQA personnel at any reviews and walkthroughs, the more common problems were spotty, undocumented attendance or participation in formal reviews only. The review teams felt that unless SQA has a scheduled involvement throughout the entire development process, accompanied by some sort of written audit report, quality cannot be adequately assured.

Other Problems. All shortcomings discussed so far occurred on at least four of seven programs. There were other problems that occurred in common on two or three programs. These were inadequate software library procedures, inadequate SQA involvement in formal test, and inadequate review of deliverable documentation. Two problems noted with libraries were lack of backup provisions and lack of formal control procedures. In SQA involvement in formal test, there was a lack of any planned participation. In the case of deliverable documentation, SQA personnel were not required to "sign-off" in any way on software documentation before release to the government.

Positive Aspects of Implementations

The presentation of findings so far has been on the negative side. There are several positive aspects that should also be stressed about the examined programs. On two of the seven programs, overall implementation was judged to be quite good with only minor improvements to be made. In practically all cases, independence of the quality assurance organization was good. The people assigned to software quality assurance were typically well-qualified, experienced personnel. Programs were usually in place, and with one exception they all presented at least a good partial implementation of MIL-S-52779A requirements. When follow-up reviews have been performed an improved program has been witnessed.

Contract Administration Office Support

As described in the review approach, the review teams also met with cognizant contract administration office (CAO) personnel. The organizations involved were Air Force Plant Representative Office (AFPRO) and Defense Contract Administrative Services (DCAS) Offices. In four of seven cases, the cognizant CAO did not have personnel with experience or background in software quality assurance. The review teams felt that these organizations could not provide adequate verfication of contractor efforts in SQA. The root of the problem seemed to be a lack of SQA qualified personnel. In addition to this problem, the teams found that in almost all cases tasking from the program office to the cognizant CAO did not include SQA concerns. Again, the problem seemed to be a lack of personnel knowledgable in software quality assurance, this time in the program offices.

Analysis of Findings

It is difficult, if not dangerous, to generalize from only seven cases. But even if no firm conclusions can be drawn, analysis of the results from the seven reviews leaves some strong impressions.

Weakness of MIL-S-52779A Perceived as Possible Cause of Inadequacies. Contractors appeared to be willing to implement software quality assurance programs in accordance with MIL-S-52779A, but the actual implementations tended to be weak. Yet, when presented with suggestions for improving their programs they generally accepted them. No common thread can definitely be picked out, but it is possible that this vagueness in programs could be due to the vagueness of the basic specification. A reading of MIL-S-52779A generally leaves one with an idea of what general areas should be covered in a software quality assurance program, but with few specific requirements. From one aspect this is good, since the government doesn't want to constrain a contractor or tell him how to do the job. But we shouldn't be surprised if our expectations aren't met when they don't exist in writing.

Government Monitoring Weak. A second impression left by an analysis of the reviews is that a contractor will only be as serious about a requirement as the government is. This is not meant to imply that the contractor is at fault. In fact, response to suggested improvements was good, as just noted. However, if the government stresses delivery schedules and end results, a contractor will probably respond favorably in those areas also. The point being that if we're going to put a software quality assurance requirement on contract but not seriously examine its implementation, we again can't be surprised if the contractor's implementation reflects our interest.

Shortage of Software Quality Assurance Personnel. Finally, a word about manning. Low manning could be caused by many things. It could be a simple failure to assign personnel in adequate numbers. After all, those contractor personnel assigned to software quality assurance were typically qualified, experienced personnel. But there were also cases of personnel having left the company and difficulties in filling vacancies. No software quality assurance organization seemed to have an over-staffing problem. This parallels the observed contract administration office shortage of software quality assurance personnel. The impression is that software quality assurance personnel are not easy to come by.

CONCLUSIONS AND IMPLICATIONS FOR FUTURE POLICY

Stronger Software Quality Assurance Standard Needed

A good deal of work has been done within Air Force Systems Command (AFSC) to improve the application of software quality assurance to defense system contracts. A fair part of this work has dealt with what the requirements should really be. In examining contractor software quality assurance programs, the Electronic Systems Division technical staff has found weak applications. Although not specifically discussed in terms of the AFSC effort, an underlying cause of both observations seems to be a vagueness within MIL-S-52779A of what is required. The conclusion drawn is that a software quality assurance standard of some sort is needed either in addition to, or in place of, MIL-S-52779A. Such an effort has already been undertaken by the Joint Logistic Commanders in the drafting of a Military Standard for Software Quality Assurance and Measurement. The Department of Defense, while insuring the standard does not become too directive in nature, should also insure that it contains actual minimum standard requirements for software quality assurance implementation.

Software Quality Assurance Training Program Needed

In conjunction with a more explicit standard, the Department of Defense needs to stress software quality assurance training. It may be necessary to establish a formal training program, since while some courses exist, there is no clear training progression. Establishing one is a reasonable approach to alleviating the shortage of qualified software quality assurance personnel.

REFERENCES

1. Software Quality Assurance Program Requirements, MIL-S-52779A, 1 August 1979. US Army Computer Systems Command, Fort Belvoir, VA 22060.

2. Evaluation of a Contractor's Software Quality Assurance Program, DLAH 8250.1, May 1981. Headquarters Defense Logistics Agency, Cameron Station, Alexandria, VA 22314.

3. Evaluation of a Contractor's Software Quality Assurance Program, MIL-HDBK-334, 15 July 1981. US Army Computer Systems Command, Fort Belvoir, VA 22060.

4. Quality Assurance Program, AFSC Regulation 74-1, 29 July 1980. Department of the Air Force, Headquarters Air Force Systems Command, Andrews Air Force Base, DC 20334.

5. Guide for Quality Assurance Managers, AFSC Pamphlet 74-4, 22 July 1980. Department of the Air Force, Headquarters Air Force Systems Command, Andrews Air Force Base, DC 20334.

6. Neil, George and Gold, Harvey I. Software Acquistion Management Guidebooks: Software Quality Assurance, August 1977. Produced for Electronic Systems Division, Hanscom AFB, MA. National Technical Information Services (NTIS) accession number AD A047318.

ABOUT THE AUTHOR

CAPTAIN WILLIAM P. NELSON is currently chief of the Computer Systems Engineering Branch within the Deputy of Technical Operations and Product Assurance, Electronic Systems Division. He is responsible for aiding in the implementation of computer resource policies and providing technical assistance to the program offices. He holds a Masters degree in Computer Science from the Air Force Institute of Technology (AFIT). In his previous assignment he was an Instructor in the Electrical Engineering Department, School of Engineering, at AFIT. He taught courses in introductory digital engineering, systems software, and software acquisition management. He has also been assigned to the Air Force Data Systems Design Center where he was responsible for maintaining, modifying, and providing user support to an automated medical materiel management system. He is a member of the ACM, the IEEE Computer Society, and holds a Certificate in Data Processing.

AIR FORCE SOFTWARE TESTING/ACQUISITION POLICY

Major Arthur E. Stevens

Deputy Director, Computer Resource Policy and Practices

HQ Air Force Systems Command

Andrews AFB DC

ABSTRACT

This paper identifies the Air Force policy for software testing during the weapon system acquisition process. The implementation of this policy is through development test and evaluation (DT&E) and operational test and evaluation (OT&E). Testing is systems oriented within AFSC with requirements eminating down from the system level to the subsystem/configuration item and ultimately to software components and modules. The typical acquisition cycle for a weapon system is discussed outlining the mechanisms that exist for ensuring software requirements, including testing, are identified and satisfied during the various acquisition phases. Many of these mechanisms have been available to program managers for sometime but required considerable management innovation to utilize effectively. These mechanisms have been modified and new ones have been created to accommodate the expanding role of software in weapon systems. Finally, program managers are being required to utilize these mechanisms in determining and reporting program development status.

Introduction

The DOD and Services are undergoing an evolutionary change in how they acquire and manage software-intensive weapon systems. The need for this change is readily apparent when one investigates the problems that have plagued the development of software-intensive weapon systems over the past ten years. Who has not heard at least a few of the horror stories associated with developing weapon systems with embedded computers? The "software acquisition disease" has many professed symptoms, several of the most common being: long delays in fielding a system that is often operational deficient when delivered, huge cost overruns associated with software development, software not supportable with government resources, and even program termination. Since future weapon systems are expected to have increased need for embedded processing, the disease can only get worse if a cure is not found.

The DOD and Services are working closely together to identify the causes of the "software acquisition disease" and are developing new strategies to curtail and eventually cure the disease. Many initiatives are in progress today, or are planned in the near future, that will investigate almost all facets of weapon systems acquisition and will make recommendations, develop new policy, and even create new mechanisms for improving software development. One element of this investigative effort is software testing.

Software Testing Policy

DOD policy requires software be considered a major component of the overall system. For software-intensive weapon systems, software performance and test objectives must be identified and evaluated to determine the level of software maturity at each phase of the acquisition process. The Air Force acquisition policies for software-intensive weapon systems compliment the DOD policy and are specifically directed at improving the management and engineering process for software.

Within Air Force Systems Command (AFSC), overall policy for testing remains systems oriented. All testing eminates from the system level and filters its way down to the subsystem/configuration item (CI) level and ultimately to software components and modules. Allocation of system test, and often as difficult as, requirements is comparable to the allocation of system performance requirements. In fact test objectives must support the system performance requirements. One major problem associated with requirements allocation is that a certain level of system definition and design must be completed before an adequate allocation of testing or performance requirements to all system components, including software, can be accomplished.

Once testing requirements are allocated, a decision has to be made as to "how much" software testing is required. There are a number of factors which must be considered in making this determination, a few of the factors being risk (technical, schedule, and cost), existence of any nuclear safety requirements, and mission criticality of the weapon system. Ultimately the final decision on "how much" software testing rests with the weapon system program manager.

Testing within the Air Force consists of development test and evaluation (DT&E) and operational test and evaluation (OT&E). Where DT&E is directed towards demonstrating that system engineering, design, and development are complete and satisfy the specified requirements, OT&E is conducted to ascertain whether the system satisfies the users operational requirements when operated and supported by Air Force personnel in the actual operational environment. The management plan for the conduct of DT&E and OT&E is the Test and Evaluation Master Plan (TEMP). Development of the TEMP is the responsibility of the AFSC program manager with OT&E inputs provided by the Air Force Test and Evaluation Center (AFTEC).

The TEMP is designed to bring all major test requirements together for the system and its components to insure that a correctly functioning system is demonstrated that satisfies operational needs. DOD policy requiring establishment of quantitiative and demonstrable performance objectives for computer software is reflected in the TEMP. In addition, the TEMP is responsible for identifying software test objectives and defining responsibility for accomplishment of these objectives during both DT&E and OT&E. It must be recognized that the TEMP is a management plan and does not include detailed test procedures, evaluation criteria, etc. Specific details of DT&E and OT&E are usually contained in detailed test plans developed by AFSC and AFTEC. It should also be noted that creation of the TEMP and detailed test plans actually requires close coordination with the Air Force using and supporting agencies.

AFSC Test Implementation

At the system, subsystem/CI, and component module levels, practically all testing is conducted by the development contractor in accordance with government-approved test plans and procedures. Emphasis within AFSC is placed on ensuring that tests for all levels are properly specified to demonstrate documented/baselined requirements and that they are sufficient to demonstrate realistic performance. In almost all cases, AFSC relies on the contractor's development facility for the conduct of the initial DT&E tests moving to an operational site for final system DT&E. The OT&E within the Air Force is conducted by AFTEC utilizing Air Force operators and maintenance personnel in as near an operational environment as possible.

Emphasis upon software testing revolves around the development of test plans and procedures that demonstrate B-5 specification requirements (as defined in MIL-STD-483). Contractor-prepared test plans and procedures are reviewed in-depth by government personnel to ensure proper and complete testing. Tests at the subsystem/CI level are formally witnessed by AFSC with deficiencies and discrepancies noted for resolution. Internal contractor testing exists at the component, module, and computer program configuration item (CPCI) levels in the form of preliminary qualification tests (PQTs) and formal qualification tests (FQTs). PQTs and FQTs are not generally witnessed by government personnel. As software is integrated into and tested as part of the system, AFSC software engineers are responsible for ensuring the adequacy of and compliance with CPCI, subsystem/CI, and system level requirements.

Once the software is integrated into the system, its functions become intricately intertwined with subsystem and system functions. It becomes very difficult to identify strictly software performance and test objectives at the system level. AFTEC has wrestled with this problem for several years and has not been able to adequately segregate software performance from system performance. OT&E requires an integrated system at which point software peformance becomes part of the system. Also once a system is integrated, performance tests are directed against system requirements which generally do not require vigorous testing of the integrated software to satisfy general operational requirements.

The point being made is that extensive software testing at the component and module level must occur prior to system integration. System buildup is a phased process that requires successful accomplishment of each phase prior to beginning the next phase. Each phase becomes progressively more complex until at the system level software component and module deficiencies may be difficult to detect or even isolate if encountered. Detailed and successful testing of software at the component and module level will allow a smoother integration of CPCIs/CIs/subsystems. Although there will exist software intensive subsystems, particularly in support areas, where detailed software testing can continue throughout DT&E and OT&E, most weapon systems utilizing embedded processors do not have this luxury.

Acquisition Policy Changes

The Air Force has made a number of improvements to its acquisition practices that in many instances compliment the improved software testing policy. Existing, as well as proposed, changes to the acquisition policy are directed at improving the quality, integration, supportability, overall performance, and timeliness of weapon systems software. When successful, the combination of these acquisition policy changes should improve the quality, ease of integration, and testability of software. The following paragraphs address the major improvements to acquisition policy within the Air Force that collectively are expected to improve the overall software development process.

Software Requirements

Present policy requires identification of software performance and testing objectives as previously discussed, but at what point in time during the acquisition process can or should this requirement be satisfied? The answer is as soon as software is defined sufficiently to allow proper allocation of system performance and requirements. Initial efforts should begin as early as the Statement of Operational Need (SON). At the time the SON is generated the system design, to the level required to understand the details of the software, is generally unknown for a new system, but certain software requirements may be known and should be addressed, e.g., performance based on upgrade or replacement of an existing system and whether a software support facility/capability is required.

Generation of the request for proposal (RFP) is the next major step where software requirements need to be identified. This is an extremely critical activity which actually initiates the design and development process. The RFP identifies the requirements which the contractor must satisfy during system development including testing. Where contractor flexibility in design is desired, requirements should be general, but where specific software performance, test, and support requirements are known they must be expressed in sufficient detail and clarity for a contractor to accurately determine his development approach and estimate costs. Software documentation and development facility requirements need to be addressed when known. Costs for these items escalate considerably when they are identified during program development.

The following are the major changes that have been implemented to improve the RFP generation and contractor selection process.

- Use of Air Force Contract Management Division expertise to review RFPs, review of RFPs by the using, supporting, and testing organizations.
- Ensuring the most qualified government personnel are identified for the source selection evaluation.
- More emphasis during source selection on analysis of prospective bidders management and technical proposals.
- Greater use of pre-award surveys to determine the soundness and past performance of each prospective bidders management organization and technical expertise.

Several groups exist to ensure that software requirements are adequately covered for testing, performance, and support. Several documents exist that provide a formal vehicle for identifying system, including software, requirements and, when necessary, problems. In most instances, these documents, and the groups that develop them, are direct inputs to the program manager to support him in determining system development progress. I have already mentioned the SON and RFP. Also included is the Program Management Directive (PMD) which identifies the overall acquisition policy and organizational responsibilities for accomplishing the weapon system acquisition and provides the basic charter for creation and operation of the program office.

Test Planning Working Group (TPWG)

The TPWG exists to integrate all test requirements for a weapon system and to advise the program manager on all test matters. The TPWG is tasked with ensuring that all resources necessary for support and accomplishment of testing are identified and available for DT&E and OT&E. The TPWG includes representatives from the DT&E and OT&E organizations.

Test and Evaluation Master Plan (TEMP)

As previously discussed, the TEMP is the master plan for ensuring adequate DT&E and OT&E and identification of test responsibilities. Test objectives, including software, as well as critical system issues and overall system test philosophy are identified. This is a coordinated document between the program office and testing organizations.

Computer Resources Working Group (CRWG)

The CRWG is convened by the program office for the primary purpose of identifying all requirements of the using and supporting organizations for supporting the weapon systems software after program management responsibility transfer (PMRT). Management and support responsibilities are identified and all resources required to manage and support the software are determined. The CRWG will also recommend the level of independent verification and validation (IV&V) that should be provided. As in the case of the TPWG, the CRWG provides recommendations and status on computer resource issues to the program manager but the program manager has significant latitude in making management decisions.

Computer Resources Integrated Support Plan (CRISP)

The CRISP is developed by the CRWG and documents the software support approach, to include resource requirements and organizational responsibilities, for the fielded operational software. The CRISP is directed at software management and support after PMRT but can also identify such requirements for the pre-PMRT phase.

Independent Verification and Validation (IV&V)

Utilization of an independent verification and validation (IV&V) is receiving stronger emphasis. IV&V can be a powerful tool when applied effectively during program development. The level of IV&V can range from requirements and design validation to actual software testing. IV&V can be performed by: a contractor with existing tools to support IV&V tasks, the eventual Air Force support agency, or a mix of contractor and Air Force personnel. The latter two cases allow early involvement by the supporting agency and can provide a smoother and more timely transition for Air Force support of the operational software.

Most of these tools created early in a program's development continue throughout the acquisition period to monitor and evaluate system and software design, integration, test, and production progress. In fact the CRWG continues throughout the life of the program. IV&V, although initially the program manager's tool, can easily transition to the operational phase to the extent it supports future software support requirements.

System Design

Following contract award, many activities, mechanisms, and documents are initiated that are directed towards ensuring system and software requirements are adequately reflected in the system design and demonstrated during DT&E and OT&E. Increased emphasis is being placed on the software design process to the extent that a separate software requirements review has been proposed for conduct prior to the preliminary design review. Software design must undergo the rigors of the preliminary and critical design reviews as well as development of appropriate development and product specifications for each computer program configuration item (CPCI).

Present activity in this area includes development of a software development standard, a more concise set of software documentation/data requirements, and development of a software quality assessment and measurement program. Initiatives are in progress to investigate new tools, both management and technical, that will aid the program manager in determining the adequacy of the software design and development progress.

System Integration and Test

Although this area was briefly discussed earlier, it is important to identify the mechanisms that exist for ensuring the adequacy and actual accomplishment of system and software testing. Planning includes the TPWG and preparation of the TEMP and is supported by government approved contractor test plans and procedures, the DT&E test plan, and the OT&E test plan. The quality of the test program is strongly dependent upon the quality of these plans and procedures. The success of the test program requires adherence by the program manager to a vigorous test, and he must be critical of deficiencies encountered at any level of integration and testing.

Initial software testing is accomplished by the contractor at the component, module, and CPCI levels culminating in PQTs and FQTs. DT&E is first used to demonstrate successful integration of CIs and CPCIs into a system by the contractor. Government DT&E begins once contractor testing (sometimes called contractor test and evaluation (CT&E)) is complete. Ideally OT&E follows a complete and successful DT&E. In reality, schedule economics are forcing preliminary OT&E to be conducted concurrent with DT&E and dedicated OT&E is often begun with significant known deficiencies unresolved from DT&E.

Finally, functional and physical configuration audits (FCA/PCA) are performed to ensure all requirements are met, all deficiencies are documented for resolution, and the deliverable products (documentation and code) are accurate. Completion of FCA/PCA establishes the product baseline for the system.

Summary

Many tools, mechanisms, and activities exist to support the program manager in monitoring and assessing the progress and adequacy of software development throughout the acquisition process. Additional tools and technology are being investigated particularly in the area of software quality assessment and measurement. The program manager is the sole focal point for successful implementation of the tools and activities discussed in this paper. He also has significant flexibility in making management decisions. How the program manager utilizes the data produced by these tools and activities will determine how smoothly a system will be acquired including the resolution of problems, and how well it performs and is supported. Although a program manager's decisions are effected by a number of factors such as costs, schedules, and political environment, the judicious use of these tools can significantly improve the acquisition of better quality software in a more timely manner.

ABOUT THE AUTHOR

Major Arthur E. Stevens is presently assigned to the acquisition logistics directorate at HQ AFSC. He is deeply involved in the development of acquisition policy for computer resources. Major Stevens has been involved with the acquisition, modification, installation, and testing of Air Force weapon system computer resources since 1968. His first assignment with AFLC gave him the opportunity to get his "hands dirty" in software support. He modified maintenance and diagnostic software for phases array radars and installed third generation processing systems at two space track radar sites. While at Space Division, he helped acquire AFSATCOM terminals and managed improvements to the Defense Support Program. His last assignment was to the Air Force Test and Evaluation Center (AFTEC) where he prepared operational software test approaches for a variety of cruise missile, avionics, space, and electronic warfare programs. He was also involved in AFTEC's initial efforts to develop standardized evaluation tools for determining software supportability.

ACQUISITION PROBLEMS INFLUENCING SOFTWARE DEVELOPMENT AND OPERATIONAL TESTING

Michael A. Blackledge, Lt Col, USAF

Walter G. Murch, Major, USAF

HQ Air Force Test and Evaluation Center
Kirtland Air Force Base New Mexico 87117

ABSTRACT

This paper outlines the basic acquisition process for embedded computer systems and examines the "real-world" problems inherent in this process. The concepts of software development testing and operational testing are next examined, and finally the cause and effect relationships between the acquisition structure and the operational testing world are contrasted. Recommendations are presented for improving each phase, both through changes in Government policies and procedures with regard to contractors, and through adherence to software development standards. Although examples are drawn from Air Force acquisition experience, the problems are common and recommended solutions are applicable to many embedded computer resource acquisition and testing programs.

I. INTRODUCTION

There are numerous problems influencing development testing and operational testing, those two separate and not altogether distinct activities that must be a part of every successful acquisition program, in particular any embedded computer system software acquisition. Embedded computer systems (ECS) account for more than half of the Department of Defense's multibillion dollar budget for software and are an inherent part, if not the heart, of every major communications system and weapon systems being acquired today. The development and operational testinq of these systems is an essential part of their acquisition. Development testing is generally thought of as testing against the developing specifications and operational testing is usually considered to be testing against the user's requirements, as performed in the user's defined operational environment. However, the actual differences are somewhat more subtle than this.

There is considerable blurring of any boundary lines between development and operational testing. This paper will define the two types of testing, outline who usually does what specific types of testing as pertains to software, examine the differences and similarities between the two, describe the problems involved with both, and then summarize the overall influence of the acquisition process on these areas. The paper will present viewpoints and recommendations primarily from experience with Air Force systems, but the problems described and the relationships explored are expected to be common throughout the software community.

The president of International Test and Evaluation Association, Dr A. R. Matthews, has recently stated (1) that

> "OT&E clearly directs its role into performance...but this role is not exclusive of DT&E performance testing. OT&E operational performance testing primarily concerns application and is downstream in the time-phased program conducted by definition with operational personnel in an operational environment without extensive instrumentation. This latter fact emphasizes the need for technical performance and preliminary operational performance during DT&E to ensure avoiding final changes that are no longer cost effective or within the required time schedule."

Department of Defense Directive 5000.3 (2) describes development test and evaluation (DT&E) as that test and evaluation conducted to verify attainment of technical performance specifications and objectives. The same directive describes operational test and evaluation (OT&E) as that test and evaluation conducted to estimate the system's operational effectiveness and operational suitability, identify needed modifications, and provide information on tactics, doctrine, organization, and personnel requirements.

To establish the background against which both development and operational testing must operate, let us take a brief look at the initial acquisition process in the Air Force as it affects the quality of a delivered system. The process is similar to that in any service, and comparisons can be made to industry as well.

II. THE ACQUISITION PROCESS.

There are two critical phases of the acquisition process during which the foundations are laid for both development and operational testing. Rather than attempt to rephrase these phases in terms of the recent acquisition improvement program, they will be referred to as the requirements definition phase and the full scale development phase. Let us examine these phases and what should be done during them to control the length of the acquisition process while strengthening the basis for testing.

PHASE I - REQUIREMENTS DEFINITION.

This phase is generally a Government activity which establishes the quality of the rest of the acquisition process. In particular, it directly affects the amount of both development and operational testing that will be needed. There are a number of activities and critical milestones associated with this phase, some of which are described below. There is only one area that transitions to the second phase and physically influences acquisition activities: development of the statement of work (SOW) (weapons requirement document) as part of the request for proposal (RFP) and source selection process.

a. System Program Office (SPO) Planning. This planning is done with the best intentions of arriving at a reasonable cost/schedule for the upcoming acquisition. Depending on internal pressures for estimates, the program office tries to determine a reasonable system configuration, assess the risks for elements of the system, and develop the cost and schedule profiles. This is one of the first places that problems can develop. If the program office has been a priori constrained on cost or schedule, the estimates and rationale can be shakey. Optimism, misreading of availability of technology, lack of understanding of development cycle, ignoring major features (e.g., support equipment, adequate software development facility, sufficient time/effort for an adequate integration test period), among other factors, can lead to badly flawed estimates of cost/schedule.

b. OSD/Air Staff Approval. At these levels, the staff is at least one step away from the planning environment for the system in question and are, perhaps, a part of the budget cycle. Thus the politics/motives are different. In order to shoe-horn an acquisition into the budget to insure its continuation, adjustments to cost/schedules are made, sometimes with little appreciation or concern for the actual down-stream effects. And this happens yearly, causing a continuing redefinition of system effort. Further, politics dictate the definition of "planning milestones" such as initial operating capability (IOC), defense system acquisition review council (OSARC), etc. These dates then become "holy", presenting (from a system development viewpoint) artificial schedule milestones.

c. Contract Prework at SPO. This is the point where the previous work is performance/schedule/documentation/qualification requirements to be levied on a contractor. All the generalizations of step a above now get more specific in the statement of work. The problem is that the level of effort implied in the statement of work may be inconsistent with the generalized schedules and costs (which themselves may not be good). Contract milestones (e.g., preliminary design reviews (PDR)), are overlayed on the schedule--sometimes without adequate thought of the implications. For example, PDRs scheduled early in the contract without consideration for contractor hiring problems, length of time for new people to understand operational requirements, time span to really get documents developed, delivered, authenticated... the result is a procurement document consistent with steps a and b, but which may not be consistent with reality. This leads to the next block.

d. Statement of Work (SOW). The SOW must accurately and unambiguously reflect known and required (not necessarily desired) operational requirements for a specific weapon system. Two critical problems occur at this point in the acquisition "definition:" the developers of the SOW over-specify in areas where requirements are not really understood and could not be easily verified during testing, and/or known requirements are not specified in detail. This latter problem causes considerable anguish during the initial phase of contractor design and development when various individuals confuse the issues with "what the Government meant was...." It is absolutely essential that Government distinguish and understand the difference between hard and fast system requirements, for which an operational need exists, and areas of general functional performance where the development contractor should be allowed to research and propose the best cost effective alternatives. The SOW should reflect this difference, with the contractors allowed to describe their system expertise in their response to the RFP in those areas where tradeoffs exist.

e. Administration of the RFP/SOW. This is the transition point to the second phase, namely contract award and full scale engineering development. A well-written SOW makes contractor selection easier. Two critical errors occur at this point: the Government does not adequately consider all contractor proposals and the contract is awarded primarily on least cost. Included here is the issue of small business awards to less than fully qualified contractors.

Regardless of how well the SOW is written, the Government should make available the most knowledgeable people, both on the system being procured and on the development process, to form the source selection team. In addition to evaluating the technical components of the contractor's RFP response, those areas that most affect development need to be evaluated in more detail: the contractor's organization (for the weapon system being developed, not necessarily that used in past successes), contractor's past performance in similar/other Government programs, cost and schedule proposals based on the contractor's understanding of requirements, etc.

The source selection team should not be hindered by irrelevant or unrealistic external factors such as arbitrarily imposed review limitations or access to the contractors for more detail. Time used at this point, if used effectively, will save time by many factors later during the development process. If during the source selection review, Government personnel notice an ambiguity based on a poorly-stated, or overstated, Government requirement, there should be an easy mechanism to correct, update, and improve the SOW as problems are encountered. If it is not done at this time, it will create numerous waivers, engineering change proposals, and confusion during system development.

In the final selection of the development contractor, realistic consideration should be given to past contractor performance. If major differences occur in the cost proposals, a debate between contractors to discuss differences is not unrealistic to provide the Government a better insight into contractor's attitude, abilities, etc. Too often, we have seen the winning contractor selected according to lowest cost and later find out the more competent contractor easily pointed out the pitfalls associated with the winning contractor's proposal. We should not place so much emphasis on proprietary information. If the contractors don't want to share, we can always find one that will.

Last point. Some companies have technical proposal teams with extremely qualified personnel, but once the contract is awarded, less qualified and less knowledgeable people are put in charge of the program. The Government should be extremely concerned when one contractor significantly underbids all competitors.

Summary Recommendations to Improve this Phase

(1) Spend more time on all weapon system SOWs to reduce ambiguities, overstated and understated requirements, and to ensure the SOW requirements can be translated into realistic and testable requirements. This might require a special technical board with engineers, operators, contracts people, etc., to review SOWs.

(2) Make the source selection effort realistic and effective by selecting good personnel and giving them the time, authority, and flexibility to do their job.

(3) Give more emphasis to past performance in selecting winning contractors and take special review action when a significant difference exists between competing contractors.

(4) Let the contractors prepare cost and schedule milestones.

(5) Along this same line, artificially imposed operational dates cause numerous problems. Although such dates on paper cause no problem, forcing the contractor to design a system by working backwards from a fixed date is unrealistic. The contractor should be allowed to propose a realistic development schedule; then the Air Force should define, perhaps with some negotiation, realistic operational availability dates.

(6) At the time of RFP release, emphasis should be placed on having the contractors propose weapon systems from a system perspective for hardware and software development. Too often, computer hardware is proposed by the contractor based on what he thinks the Government wants to see or because the Government specified the hardware in the SOW. The result is a system design with no hardware flexibility and possibily incompatible functional requirements and allowable software design. We must realize that standardization often conflicts with design flexibility. One of the first deliverables should be a life cycle costs/benefits study of standardization vs new hardware to help make this decision. And the Government's budget should be based on the most expensive decision. NOTE: Although there are numerous other factors that affect this phase, e.g., preordained development, test, and production schedules or contract types, most of these "other" factors tend to be political or artificial. We feel if a commitment is made to improve the statement of requirements for new systems and the methods of contractor selection, there can be significant improvement in the acquisition process.

PHASE II - FULL SCALE DEVELOPMENT.

The second phase is full scale engineering development or popularly known as contractor interpretation of AF requirements into a physical system. There are too many problems associated with this phase to discuss them here. The majority of the problems in the authors' opinions are related to too much Government interference in the development when the contractor should be allowed flexibility and too little Government involvement in the areas of contract administration and general contractor internal management when it would be most effective. Assuming a competent contractor is selected and the Government has confidence in its selection process, the contractor should be allowed the flexibility to propose and design the system that meets SOW requirements. What too often occurs is that the Government (always different people than were involved in the first phase activities) starts playing the requirements changing game or "interpretation" game. If a new requirement is not absolutely essential or the contractor's interpretation is at least minimally acceptable, "LEAVE THE CONTRACTOR AND CONTRACT ALONE." Provide contractual incentives for exceeding specified system performance thresholds or for adding improved performance where not specified.

Artificially imposed design milestones should not be established. Technical interchange meetings are mandatory and so is a systems requirements review as a reliability check to be sure the contractor's proposal and his program management after contract award are in agreement. The contractor should be allowed to identify in his proposal realistic design review milestones.

The Government should impose "strict" penalties if the contractor cannot meet his own proposed milestones. Note that penalizing a contractor for Government-imposed milestones is risky even if the contractor agrees with the milestones. "Insufficient information at time of agreement" is a good rebuttal when schedules slip or inadequate reviews are held.

To ensure that the contractor supports his end of the bargain, the Government should spend more time evaluating the contractor's administration and management activities. Often times development delays are caused by incompetent or even deceptive contractor practices that may or may not be evidenced through technical development.

Those areas in which Government is least knowledgeable and yet cause numerous problems, e.g., software development, integration, and testing, should be supplemented by well-qualified commercial organizations. A major problem is that the Government does not effectively utilize independent contractor support. Too often, we let independent contractors propose to the Government what is required rather than the Government having enough knowledge on the system to know what is needed. This causes two problems: wasted Government funds during a period when funding is critically monitored and not addressing critical development problems because of misdirected activities in both critical and noncritical areas.

Summary Recommendations to Improve this Phase.

(1) Contractor, if selected with confidence, be given more freedom in design, development, and schedules.

(2) Government emphasize and monitor contractor management and contracting practices.

(3) Penalize the contractor for not meeting his own milestones and reward him for efficient development. This requires more flexibility in selecting the type of contract to be awarded (getting away from preordained contracting philosophies, e.g., fixed-fee, no-incentive contracts).

(4) Utilize independent evaluators more often and more effectively.

(5) Take steps to minimize the impact of budget revisions on the SPO. Considerable SPO management time is consumed rearranging resources every time the budget process changes the SPO's available funds. Any time the SPO spends managing this problem is time not spent managing development problems, with the resulting implicit delays.

III. CHARACTERISTICS OF TEST PROGRAMS FOR SOFTWARE

DEVELOPMENT TESTING (SOFTWARE).

Planning for test and validation of a software intensive project must begin early in any software development project. The project manager must ensure that test and verification and validation plans are included in the system (and software) requirements and design specifications. Both the testability of those specifications and providing sufficient "hooks" for independent verification and validation contractors must be provided.

Two of the basic concepts in writing system (and software) requirements are to ensure that those requirements are complete, feasible, and testable, and to establish validation and test criteria. For the development of system requirements, fault-tolerant requirements must be specified, and any known conflicts and omissions in the requirements must be identified and corrected. For the development of software requirements, a requirements-level test plan should be developed. Later, during the development of the software design specification, a specification-level test plan should be written.

Finally comes the time for unit and integration testing. Here it is very important to outline testing concepts and goals. The tester must develop a test strategy, keeping in mind that the primary test objective is striving to uncover failures, not trying to "get by." The test plans developed during this phase must relate to the system requirements and specifications. The "build-a-little, test-a-little" philosophy can be put to practical use at this time, by carefully planning for test scheduling, reporting, and control. The tester must quickly learn how to avoid test bottlenecks and other problems--a configuration management control process can provide some real control here.

The test plans and test design and test case specification documents are important to accomplishing a good test program. Developed from the requirements documents and concurrent with the design process, these documents identify resource requirements, simulation requirements for inputs, analysis requirements for outputs, test case cross reference to system and software requirements, etc. Without a systematic planning at this, a thorough test program cannot be assured. Further, after completion of the test program, reconstruction of what was tested is difficult. Our experience is, however unfortunately, that this level of test planning documentation is not available. This non-availability also affects the ability of support agencies to maintain the software.

For unit testing itself, the developer must design test cases that include exercising critical software functions and boundary checking parameters. It is important to generate sufficient test data to perform these functions, and often times program "stubs" and test driven programs must be written before the "test-a-little" can be accomplished. Provisions must also be made for retesting after an error condition has been encountered.

In preparation for integrating the modules into a system, it is advisable to build from the top-down, and to build horizontally as much as possible rather than vertically, in order to allow crucial interfaces to be exercised often, and as much testing as possible down parallel paths.

During this phase, the test driven modules and program stubs are replaced with actual program modules, and stepwise integration and regression testing can be initiated. Special testing techniques must be employed to test real-time functions.

There are a number of specialized tools and techniques to assist the tester. These may vary depending on whether the system uses a stand-alone development system, or a host-target machine configuration, but in general they include the following: an automatic test data generator, environmental simulators, and interactive debugging tools.

Now the tester is ready for system verification and test. Keep in mind that the purpose and goals of system verification are to: (a) verify that the system performs as required, (b) revalidate any revised requirements, (c) measure performance under maximum load conditions, and (d) verify the completeness and correctness of the system documentation.

One of the more valuable tools available to the developer, albeit a costly one, is independent verification and validation, or IV&V. The Air Force has made it a policy that this structured process must be considered for use on all software intensive systems. Definitions and some details will be published in the revised AFR 800-14, but in short, IV&V has considerable value in high-risk applications, and there are now a number of companies experienced in this discipline.

Who will perform the system and acceptance testing? Hopefully, it will be an individual somewhat removed from the development work itself, someone like a quality control engineer. There should be scripts and documented techniques for the system test for the individual to use, and he should have available both unit and integration test results. The capability should exist both to simulate hardware failures, and to test any system/user interfaces. Even minor deviations from the procedures should be viewed with a jaundiced eye, and above all, the integrity of the configuration management scheme must be maintained.

There is no question that software failures are going to occur throughout the development cycle. Studies have shown that the most costly of these errors to correct are those that were created early and discovered late, such as errors resulting from implementing improper or misunderstood requirements. Particularly because of the inherent complexity of computer systems embedded in defense systems, "the often vague understanding of needs which typically characterizes defense system development during the early stages renders defense system software development particularly prone to requirements specification 'errors." (3).

OPERATIONAL TESTING (SOFTWARE)

Operational testing is that testing done in an operational environment, using representative operational and maintenance personnel. As a consequence, the software is evaluated, not as an entity unto itself, but as part of the system level test. The software is thus evaluated on a "by exception" basis, in situ. Here the focus is operational reliability (e.g., are operational requirements complete, are there any latent defects in the software?) Then the test scenarios are designed against operational effectiveness objectives.

OT&E brings to the system development cycle an independent view. The OT&E team, by observing system development activities throughout the development cycle, can provide independent advice on critical operational issues. This operational influence can best be exerted on the system development early in the cycle before significant resources have been committed to "metal-bending."

Goodenough (4) says, in discussing computer program quality,

> Correctness is not necessary for a program to be usable and useful. Nor is correctness sufficient. A correct program may satisfy a narrowly drawn specification and yet not be suitable for operational use because, in practice, inputs not satisfying the specification are presented to the program and the results of such incorrect usage are unacceptable to the user. If the program is correct with respect to an inadequate specification, its correctness is of little value.
>
> Consequently, although testing for correctness is the most common andbest understood testing goal, correctness is by no means the only important property of usable software--reliability, robustness, efficiency.... are also of significant importance. But these properties are less commonly the focus of testing activities.

OT&E provides the bridge between DT&E and operational use. DT&E activities focus on specification compliance. As Goodenough points out, this is likely not an adequate test of operational usability. The focus of the software OT&E should be, then, not on compliance with specifications, but rather on the characteristics of software which are incompatible with actual operational conditions. The intent is to determine the acceptability of the system to the user, not only from a mission effectiveness point of view, but from a supportability point of view. In this context "the term 'acceptable' implies that the user must determine what he considers to be a failure; this usually depends on the effect of the particular behavior of the system in question on the user's operations, costs, etc."

OT&E provides an opportunity to influence the operational characteristics of the software system. With access to program documentation, the OT&E team can independently assess the operational effect of specification (or other contractual) changes. Apparent adverse effects can be used as a basis for test design. Software OT&E can also provide a basis for suggesting parameters/locations within software for redesign or modification.

IV. OVERALL RELATIONSHIP BETWEEN DEVELOPMENT AND OPERATIONAL TESTING

Having examined the background against which both development and operational testing must take place, and the problems encountered, let us now summarize the overall relationship between DT&E and OT&E, in six major points.

First: All initial testing, whether DT&E or OT&E, evolves from requirements levied in program definition.

a. OT&E objectives are dynamic and reflect near "real time" performance requirements of the system.

b. DT&E requirements are nearly static due to contractual commitments to early defined requirements.

Second: Test assets for DT&E and OT&E should be similar.

a. Test assets for all testing are defined early in program development.

b. Changes in program development do not adequately get reflected in OT&E support requirements, i.e., failures during DT&E require additional testing and assets during OT&E.

c. Test assets are similar but the manpower and flavor of testing are significantly different between DT&E and OT&E.

c. Test assets are similar but the manpower and flavor of testing are significantly different between DT&E and OT&E.

d. Data reduction analysis of test data are similar for both test phases and need to be defined early during development, especially for operational usage.

Third: DT&E is a transitional lead-in to OT&E to ensure readiness for operational stressing.

a. This hand-shake concept falls short when DT&E and OT&E are combined, and the sharing of test assets (test articles, personnel, support equipment, funds, and time) is required.

b. Sharing of information and delaying OT&E until the system "passes" DT&E is an ideal situation rarely achieved.

c. Too much competition between the program office (to get the job done on time, within cost no matter what) and the OT&E organizations (ensure an operational and supportable system).

d. Requirements of OT&E often only highlight program office problems regardless of the actual causes (e.g., lack of funds, imposed congressional or Air Staff direction.)

Fourth: Emphasis on type of assets differs:

a. Development testing is more concerned with mission hardware and software to verify accomplishment of mission essential performance parameters.

b. Although system oriented, reality often forces development of mission essential subsystems at the expense of support systems.

c. Regardless of program direction, operational testing is concerned with total system capability and supportability.

d. Development testing can often be satisfied with simulated environment and prototype system (or components) to test satisfaction of requirements.

e. Operational testing is often required to use nonoperational environment but stresses realistic environment and representative system for test.

Fifth: Both OT&E and DT&E are subject to the same program shortfalls, but DT&E gets their requirements first.

a. Both subjected to unrealistic, inflexible schedules.

(1) Slips in DT&E often impose shortening of IOT&E schedule.

(2) Failures in DT&E cause shortages in OT&E test assets without contingency funding.

b. Program Office often plans optimistically for DT&E test assets and test schedule.

(1) Real life testing for most complex weapon systems has resulted in need for pessimistic, worse case planning.

(2) Problems caused in optimistic planning are passed on to OT&E.

Sixth: Management goals differ between DT&E and OT&E.

a. Program office/DT&E responsible testing organizations are concerned with meeting contractual schedules, whether such schedules are realistic or not.

b. OT&E agencies are concerned with meeting Air Force operational requirements schedules.

c. Both agencies are often up against unrealistic program schedules provided by DOD or Congress.

V. CONCLUSIONS AND OVERALL RECOMMENDATIONS

In summary, we have seen that there are inherent problems that affect both development and operational testing, particularly for software.

a. Program budgets are normally baselined and approved without inputs from proposed contractors or test agencies.

b. Contractors typically underestimate the level of work for weapon systems. Their objective seems to be for them to get their "foot in the door" (lowest bidder) and then have the option of increasing the price later.

c. Poor software configuration management practiced by the contractors.

d. Poor software design provided by the contractor--often caused by poor statement of system requirements.

e. High order (software) language standardization not progressing fast enough.

f. Inadequate time allowed for hardware and software integration.

g. IOCs are unrealistic. IOCs should be realistically stated in terms of a full concept of employment and then exercised to demonstrate performance to that concept (e.g., 30 days deployment exercise).

h. IOT&E started too early, before system is mature.

The duration of the acquisition process and the total testing time might be shortened by extending the period of source selection. In most source selections, many people who are not very familiar with the proposals are asked to review hundreds of pages of documentation from several contractors in just 1 or 2 weeks. If this period could be lengthened to one or two months (perhaps requiring some of the documentation to arrive before others) so the reviewers could take more time, they would most probably find the major differences among the proposals and provide better information with which to select a contractor. If the contractor with the better ideas is indeed chosen, fewer time consuming problems should arise in the development phase.

There should be more emphasis on establishing the software development capabilities of a potential contractor prior to contract award. This would include preaward surveys and, perhaps, establishing IG-like teams of software evaluators to assess contractor's management and software production capabilities.

There should be increased emphasis on successful accomplishment of design and system integration milestones before allowing progress to later milestones such as OT&E and IOT&E.

BIBLIOGRAPHY

1. Matthews, Allan R., "Personal Viewpoints," pg 22-23, ITEA Newsletter, April 1982, Volume II, Number 1.

2. Department of Defense Directive 5000.3, "Test and Evaluation," December 26, 1979.

3. Joint Logistics Commanders, Joint Policy Coordinating Group on Computer Resource Management, "Proposed Military Standard on Defense System Software Development (MIL-STD-SDS)" 15 Apr 82.

4. Goodenough, J. B. and C. McGowan, "Software Quality Assurance: Testing and Validation," IEEE Procedings, September 1980, p 1093.

ABOUT THE AUTHORS

Lieutenant Colonel Michael A. Blackledge is the Deputy Chief of the Software Evaluation Oivision, Directorate of Logistics, Air Force Test and Evaluation Center, Kirtland Air Force Base, New Mexico. He is currently responsible for assisting in directing some 20 computer systems analysts and engineers in the designing and planning of software operational test and evaluation, administering tests, analyzing data, and reporting results to the Air Staff. He is a graduate of the United States Naval Academy, and holds a master's degree from North Carolina State University in Mathematics. He was formerly the Chief of the Computer Support Division for Tactical Air Command's Studies and Analysis Agency, where he was responsible for large scale war gaming simulation of the air-land battle. In previous assignments, he provided computer modeling support for Oefense Communications Agency to the Joint Chiefs of Staff, course directed in the Mathematics Oepartment of the U. S. Air Force Academy, and supported high-altitude nuclear effects calculations at the Air Force Weapons Laboratory.

Major Walter G. Murch is the Chief of the Space/Strategic Software Evaluation Branch at the Air Force Test and Evaluation Center, Kirtland Air Force Base, New Mexico. He is currently responsible for activities and personnel in the evaluation of software during operational test and evaluation for Air Force systems. He holds a master's degree from Air Force Institute of Technology in Electrical Engineering and a masters in Business Administration from the University of Utah. He was formerly Chief of the Satellite Control System Engineering and Test Division in the NAVSTAR Global Positioning System system program office (SPO) where he was responsible for development, integration, and test for the GPS ground segment. In previous assignments, he was responsible for test design and analysis for inertial guidance systems, inertial navigation systems, and inertial components.

THE SOFTWARE TEST AND EVALUATION PROJECT:

A PROGRESS REPORT

Richard A. DeMillo
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia

R. J. Martin
Control Data Corporation -- Government Systems
Atlanta, Georgia

ABSTRACT

The Software Test and Evaluation Project was initiated by the Deputy Director for Test and Evaluation in 1981. The primary objective of the project is to develop new DoD guidance and policy for the test and evaluation of mission critical computer software. The information-gathering and analysis phases of the project are now complete. This article gives a summary of the status of the project, sketches of the state-of-the-art and current practice in software test and evaluation, and a summary of the preliminary findings and recommendations.

INTRODUCTION

The Software Test and Evaluation Project (STEP) was intitiated by the Deputy Director for Test and Evaluation in 1981. The primary objective of STEP is to develop new DoD guidance and policy for the test and evaluation of computer software for mission critical applications. A number of subsidiary goals have also been established for STEP. Principal subgoals include the stimulation of tool development, the support of guidelines development, and the identification of research issues and directions in the area of software testing.

STEP is conceived in four phases: information-gathering, analysis, assessment of feasibility, and policy development. The contributions represented by the papers in these proceedings are part of the information-gathering and analysis phases of STEP. The National Software Test and Evaluation Conference, conducted with the generous support of the National Security Industrial Association (NSIA) Software Group and co-sponsored by the Office of the Secretary of Defense, is intended to provide a national forum for discussing the state-of-the-art and current practices in the test and evaluation of military software.

In this article, we will present the the rationale for seeking improved DoD guidance in software test and evaluation, organization and current status of STEP, an overview of the state-of-the-art in software testing, an overview of current practice in software testing, and a sketch of our preliminary findings.

RATIONALE

The role of software in escalating the cost and driving down the reliability of military systems has been discussed extensively in recent years [1,2]. As a result, the relative imbalances in the testing of hardware and software have become increasingly visible to the defense acquisition community. In 1974, the Defense Science Task Force on Test and Evaluation observed: "Whereas the hardware development was...monitored, tested, and regularly evaluated, the software development was not."

Current estimates of increased software costs arising from incomplete testing help to illustrate the dimensions of the problem (see Figure 1). Averaged over the operational lifecycle of an embedded computer system, development costs comprise approximately 30% of the total costs. The remaining 70% of the lifecycle costs are absorbed in maintenance. Maintenance activities can include both system enhancements and the repair of errors. These are errors that might have been uncovered during by more complete testing during earlier phases. Costs in the development phase are distributed as follows: requirements and specification development, 20%, design and coding, 35%, test and integration, 45%. Thus, assuming that half of all maintenance costs are incurred in the repair of previously undetected errors, approximately one half of the operational lifecycle costs for embedded applications can be traced directly to testing activities; that is, either these costs are incurred by testing or are due to errors left undiscovered by testing.

A simple breakdown of costs does not reflect the impact which undetected errors have on total operational costs. As shown in Figure 2, the relative cost of repairing errors in software rises dramatically between requirements and specification phases and the maintenance phase. Of course, there are other implications of undetected errors in military systems. The mission critical nature of software in many modern systems means that software which fails during system operation can pose considerable risk to both the success of the mission and the safety of the personnel.

Primary DoD guidance for test and evaluation derives from DoD Directive 5000.3. This directive applies to both hardware and software components of military systems and sets forth the framework within which more specific military regulations and standards must operate. Three provisions of DoDD 5000.3 are particularly relevant to software testing. First, DoDD 5000.3 states that "Quantitative and demonstrable performance objectives and evaluation criteria shall be established for computer software during each system acquisition phase... Decisions to proceed from one phase of software development to the next will be based on quantitative demonstration of adequate software performance through appropriate test and evaluation." Second, DoDD 5000.3 requires that software be operationally tested using "typical operator personnel." Third, DoDD 5000.3 requires that operational test and evaluation (OT&E) agencies "participate in software planning and development to ensure consideration of the operational environment and early development of the operational test objectives."

For a variety of reasons existing guidance statements have not had the desired effect. Operational tests and test objectives are frequently stated in terms of overall system requirements with little or no attention to the software components. Although "typical operator personnel" may be essential for a thorough operational test, user involvement at earlier stages of development is widely thought to be desirable for effective design. While testing of hardware components may result in a database of quantitative test results against which reliability and risk models may be applied, software components are seldom accompanied by objective evidence of the effectiveness of the testing effort.

THE SOFTWARE TEST AND EVALUATION PROJECT

STEP consists of four phases intended to lead to improved DoD guidance for software test and evaluation (T&E). Phase I was an information gathering effort aimed at assessing the state-of-the-art and the state of current practice in software T&E. During Phase I an extensive survey of known techniques and tools for software testing was carried out. A summary of the state-of-the-art conclusions is presented below. The assessment of the state of current practice was accomplished by surveying DoD agencies, the military services, program offices, OT&E agencies, and Defense contractors. The survey methodology and a summary of the results is also presented.

An important activity in Phase I was a workshop held at the Defense Systems Management College (DSMC) in March, 1982 [3]. Attendees at this workshop included representatives of DOD agencies, the military services, OT&E agencies, DSMC, and NSIA. The goal of the workshop was to provide input to the STEP contractors on various aspects of software T&E at the start of the data gathering effort. Presentations given at the workshop included a summary of STEP goals, a presentation of the Embedded Computer Resources Initiative, a presentation of the Software Technology Initiative, and a summary of the Ada Initiative. Views and activities of DSMC, NSIA, DCA, and the Army, Navy, and Air Force OT&E agencies were also presented.

STEP Phase II consists of an analysis of the state-of-the-art and current practice in software T&E. In addition to evaluating the data gathered under Phase I, a panel of advisors was assembled to provide input from a cross section of the military, industrial, and university communities involved in software testing. Phase II will culminate with a set of recommendations being submitted to DDOT&E concerning the feasibility of formulating new policy for software T&E.

Phases III and IV of STEP are yet to be conducted. Phase III consists primarily of the assessment of whether new guidance can be formulated, while Phase IV is the actual development of policy statements.

STATE-OF-THE-ART OVERVIEW

Current research in software testing centers almost solely on testing for correctness, that is, on techniques that raise the users' confidence that the software functions in accordance with its specifications. "Testing" refers specifically to the activity of executing software on data (the test sets) designed to either reveal the presence of errors or insure their absence. Therefore, software testing is distinguished from other activities aimed at increasing software reliability (such as structured design techniques, formal program proving, and statistical reliability modelling).

Three aspects of extant research efforts in software testing are relevant for assessing the state-of-the-art: testing methodologies (i.e., methodologies for either generating test sets or determining the quality of previously generated test sets), testing tools (i.e., automated systems which implement one or more testing methodologies), and new hardware and software technologies which impact system reliability. In the subsections below, we will briefly outline the state-of-the-art in each of these three areas. A more detailed treatment of each of these topics can be found in the "STEP State-of-the-Art Overview" [4].

Test Methodologies. A test methodology consists of two (not always distinct) components. The first is a strategy which guides the overall testing effort, while the second is a testing technique which is applied within the framework of a test strategy.

Test Strategies. Module testing is the process of testing logical units of a program and integrating the individual module tests to evaluate the overall system. Main considerations in module testing are the design of test cases and the coordination of testing multiple modules. Test cases may be constructed from specifications or by analyzing the module code. Testing strategies corresponding to these approaches are called black-box and white-box strategies, respectively. There are two approaches to combining module analysis: incremental and nonincremental. Top-down and bottom-up testing are two incremental approaches. Thread testing is another strategy based on system requirements. Strategies have also been proposed for testing software throughout its development. Finally, several new strategies have been proposed based on an "evolutionary" view of the software lifecycle: systems are constructed as working subsystems corresponding to critical functions, and these subsystems are subjected to development and operational tests.

Testing Techniques. A variety of testing techniques have been proposed in the literature (see, e.g., the bibliography of [4]). These techniques can be classified as follows: static analysis, symbolic testing, program instrumentation, program mutation, input space partitioning, functional program testing, algebraic program testing, random testing, grammar-based testing, data-flow guided testing, and real-time testing.

Static Analysis. In static analysis, the requirements, design documents, and program code are analyzed without actually executing the code. Only limited analysis of programs containing dynamic data types and structures is possible using static analysis. Experimental evaluation of code inspections and walk-throughs has found these techniques to be very effective in detecting from 30% to 70% of the logic design and coding errors in typical programs.

Symbolic Testing. To test a program symbolically, input data and program variable values are given formal or "symbolic" values. The possible executions of a program are also characterized formally. The execution of the program is then simulated by a symbolic evaluator which interprets the formal representation of the program and data. The techniques for building expressions which descibe the state of the symbolic execution of a program lean heavily on techniques developed for proving program correctness. Studies describing the effectiveness of symbolic analysis for detecting errors indicate that it may be an effective technique for moderately large modules.

Program Instrumentation. Programs can be instrumented by statements or routines that do not affect the functional behavior of the program, but record properties of the executing program. Additional output statements, assertion statements, monitors, and history-collecting subroutines may be used to instrument programs. Experimental evaluations of instrumentation techniques indicate that experienced testers can decrease the debugging time for even complex programs using these techniques.

Program Mutation. Program mutation is a technique for the measurement of test data adequacy. Test adequacy refers to the ability of the data to insure that certain errors are not present in the program under test. In mutation testing, test data is applied to the program being tested and its "mutants" (i.e., programs that contain one or more likeley errors). If a program passes a mutation test, then either the program is correct or it contains an improbable error. Experimental evaluation of mutation testing indicates that the results of mutation testing are good predictors of operational reliability.

Input Space Partitioning. A path in a program consists of a possible flow of control. In path analysis techniques, the input space of a program is partitioned into path domains: those subsets of the program input domain that cause execution of the paths. Path analysis can detect computation, path, and missing path errors. Domain testing detects many path selection errors by considering test data on or near the boundaries of path domains. In partition anlysis, the specification of a program is partitioned into subspecifications. The subspecifications are then matched with domain partitions to increase the sensitivity of the test. All of these techniques have been shown theoretically and experimentally to be generators of high quality test data, although current technology limits their use to programs which have a small number of input variables.

Functional Testing. In functional testing, the specification of a program is viewed as an abstract description of its design. Function and data abstractions are used as guides to identify the abstract functions of a program and to generate the functional test data. Functional testing requires the specification of domains for each input and output variable of the program. Extremal and special values are the most important values in the domain of a variable. In a study of errors that occurred in a release of a major software package, functional testing was effective in detecting 38 out of 42 known errors.

Algebraic Testing. In algebraic testing, program correctness is viewed as an equivalence problem. Since the general equivalence problem is undecidable, programs to which this technique is applicable must fall in a restricted class of programs for which execution on a small test set is sufficient to infer equivalence. Applications of algebraic testing to array manipulation programs, polynomial evaluation programs, and other mathematical programs have appeared in the literature. Monte Carlo methods exist for algebraic testing procedures which make the technique tractable for many problems.

Random Testing. Random testing is essentially a black-box testing technique in which a program is tested by randomly sampling inputs. Depending on the sensitivity of the analysis desired, the sampling technique may be independent of the actual distribution of inputs or may attempt to accurately reflect the distribution of the operational environment. Random testing is useful in making operational estimates of software reliability and has some connection to problems arising in operational testing.

Grammar-based Testing. Formal specifications of some software systems can be given by state diagrams. By considering the state diagram to be a description of an automaton, classical machine identification experiments can be conducted to determine whether or not a program implementing the automaton does so correctly.

Data-Flow Guided Testing. Data flow analysis is a method for obtaining structural information about programs which has found wide applicability in compiler design and optimization. One result of data flow analysis is a set of dynamically meaningful relationships among program variables. Control flow information about the program is then used to construct test sets for the paths to be tested.

Real-time Testing. The characteristic phases of real-time software testing occur during development (on the development "host") and operational testing (on the operation "target"). Systematic techniques for testing real-time software during development, for the most part, do not make essential use of the fact that the software is real-time. Testing an integrated system on a development host requires an environment simulator and devices for controlling on-going processes. In testing real-time software on target machines, overall test objectives for the hardware/software system are used, and performance becomes a key observable factor in assessing the result of the tests. While the literature contains very few systematic techniques for real-time testing, studies of large-scale real-time software systems tests have been published, and some of these experiences may generalize to other applications.

Testing Tools. Testing tools may be classified by whether they carry out static or dynamic analysis of the program under test. Static analyzers are systems that manipulate source code to reveal global aspects of program logic, structural errors, syntactic errors, variations in coding style, and interface consistency. Static analyzers consist of front end language procesors, data bases, error analyzers, and report generators. Basic operations include data collection, error analysis, and error report generation. Existing static analyzers differ in terms of their scope of error analysis, the flexibility of user command languages, and the nature of error descriptions. Static analyzers have been used in many reported software development efforts. Dynamic analyzers, in addition to implementing many of the techniques described above, are used to generate test data, provide a convenient test environment, and compare program test output with expected output.

Symbolic Evaluators. Symbolic evaluators implement the symbolic evaluation testing technique. They provide the user with the ability to input loop and control point assertions and symbolic values for input variables. They also allow the user to monitor the symbolic execution of the program.

Test Data Generators. A test data generator is a tool which assists the user in the preparation of test sets. Three types of generators have appeared in the open literature: pathwise test generators, specification-based generators, and random generators. Pathwise test generators have four basic operations: program construction, path selection, symbolic execution, and test data generation. Specification-based generators provides the user with a language for constructing test case specifications; the system carries out the actual generation of test files from the tet specifications. Random test generators choose random values from the input domain according to statistical parameters set by the user.

Program Instrumentors. These systems gather execution data to reveal characteristics of a program's internal behavior and performance. In practice, instrumentation tools are the principal tools used to detect errors that cannot be detected by static analysis. Systems exist which provide coverage analysis, monitors and assertions, and detection of data flow anomalies. In addition, instrumentation subsystems can be found in several other types of testing tools.

Mutation Tools. An automatic mutation system is a test entry, execution, and data evaluation system that evaluates the quality of test data based on the results of program mutation. In addition to a mutation "score" that indicates the adequacy of the test data, a mutation system provides an interactive test environment and reporting and debugging operations which are useful for locating and removing errors.

Automatic Test Drivers. Automatic test drivers are software systems that simulate an environment for running module tests. They may provide standard notation for specifying test cases and automating the testing process. Some systems also compare the resulting output with the expected output and report discrepancies. Some test drivers operate on object modules, while others operate on source modules. Since the automation of the testing process is an integral part of most test tools, automatic test drivers appear in some form in most systems.

Comparators. A comparator is a system that compares two versions of data to identify differences. Comparators are used in the validation process to limit the scope of re-testing of revised software. The main differences among comparators lie in the form of the data and the flexibility in specifying tolerances for each comparison.

The report [4] contains a catalog of existing tools in each of these categories and a summary of their availability and support. Generally, however, it appears that testing tools which are available as supported, nonproprietary packages are rare. It is more common that testing tools are systems that are constructed and customized to a single software development project. Generalization, documentation, marketing, and support of such custom tools is capital intensive and is seldom carried out.

New Technology. Two aspects of new technological developments are relevant to software testing. First, there are new technologies that hold some hope for improving the programming process. New languages such as Ada, new views of the software lifecycle, prototyping, and reusable software all give software developers new tools and concepts to work with. Modern operating systems give programmers collections of tools which will aid in the testing effort. Standard architectures ease the transition from host environments to target environments. It has also become possible to "freeze" certain critical system components is custom hardware. While, the problems of determining correctness of design remains in transitions to hardware implementation, the static nature of hardware and the visibility of hardware interfaces may reduce the severity of many testing problems.

Second, new technology presents many new reliability problems. New applications such as distributed computing and communications rely on complex interactions of concurrent processes. These systems have thus far been as resistent to systematic testing techiques as older, real-time applications. Since many of these systems come equipped with stringent reliability requirements, new testing techniques are clearly needed. Customized hardware designs, in addition to providing benefits such as those mentioned above, also present new difficulties. As the density of functions that can be placed on a single chip increases, so does the complexity of the testing effort needed to determine that the designs are correctly implemented in the hardware. Existing hardware verification techniques do not appear to be adequate.

Summary of the State-of-the-Art. There exists a body of software testing technology which can be applied to increase the level and sensitivity of development testing. At present, there is little to guide software test groups in the choice of one technique over another, and choices will for the present be made on economic grounds. It seems obvious that using a systematic technique is superior to ad-hoc testing, but there is very little objective evidence to support this observation. The best approach is probably to choose a combination of techniques and tools which gives a level of test appropriate to the required reliability of the software and can be justified on the basis of overall system costs. Except for the few tools that are either in the public domain or available from tools vendors, testers will -- in the near term -- have to construct their own tools.

OVERVIEW OF CURRENT PRACTICES

In order to suggest improvements to the current practices in software test and evaluation for DoD applications, one must first know what those current practices are. Therefore, a survey of the state-of-the-practice was conducted.

Survey Methodology. The approach taken was to interview selected representatives of the military and industry on such subjects as military regulations and standards, reviews and inspections, testing techniques, tools, quality assurance, independent verification and validation, and risk assessment. To aid in the accomplishment of this effort, a set of data gathering guides were developed. Each guide was tailored with respect ot the function of the group being interviewed. These groups included HQ and Development Commands for the military services, Program Offices for selected programs, OT&E agencies, and Defense contractors. Although the guides were not administered as formal questionnaires, they did ensure that the same type of information was gathered during interviews with representatives of each functional group. In addition, the use of personal interviews rather than the mass mailing of questionnaires helped circumvent the problem of differing terminologies.

Survey Results. In the following subsections, we will describe the type of information requested of each of the functional groups during the interview process. Due to the amount of data gathered, it is not possible to present specific results in this article. Therefore, only general impressions of the state-of-the-practice in software T&E will be discussed. For a detailed presentation of the information gathered, see "STEP Current Practices Overview" [5].

HQ & Development Command Interviews. Interviews were conducted with representatives of the Headquarters and Development Commands for the Army, Navy, and Air Force. The primary purpose of these interviews was to determine what guidance the Headquarters receive from the Department of Defense with respect to software T&E, what guidance they pass on to the Development Commands, and how the Development Commands are assisting the individual project offices.

As was described earlier, the primary guidance given to the DoD components for software T&E is DoDD 5000.3. Each of the military services has implemented DoDD 5000.3 in regulations applicable to their specific circumstances. Those regulations of interest to us are, primarily, Army Regulation 70-10, the Navy TADSTANDS, and Air Force Regulations 80-14 and 800-14.

Military Standards also exist for use by contractors who are developing software for military applications. These include:

MIL-STD 1679 (NAVY)
- Weapons System Software Development

MIL-S 52779A
- Software Quality Assurance Program

MIL-STD 1521A (USAF)
- Technical Reviews and Audits

MIL-STD 490
- Specification Practices

MIL-STD 483 (USAF)
- Configuration Management Practices

For a summary of the contents of these Military Standards and other guidance documents, see [5].

In addition to the existing standards, the Joint Logistics Commanders have been directing efforts to produce tri-service standards. This has resulted, in part, in MIL-STD SDS on "Defense System Software Development". MIL-STD SDS establishes requirements with respect to software requirements analysis, design, code, testing, configuration management, quality programs, and project planning and control. It should be noted that although MIL-STD SDS is currently in the review process, some contractors are requesting waivers to use it as an alternative to other standards. The potential benefits of MIL-STD SDS are that it addresses the entire software life cycle, provides uniform terminology and definitions, and is for use by all of the military services.

Project Interviews. Interviews were conducted with representatives of specific project offices for major systems which are currently under development. During these interviews, information was gathered on project status and history, military regulations and standards invoked, reviews conducted, development test and evaluation, acceptance testing, quality assurance programs, independent verification and validation activities, operational test and evaluation, and risk assessment. One result of these interviews was the discovery of the complete faith which the military acquisition organizations place in their contractors. This is evidenced by the lack of formal procedures for tracking progress during the coding, module testing, and integration testing phases of the software development life cycle.

OT&E Agency Interviews. Each of the military services has an organization which has been given the mission to operationally test and evaluate new and modified systems. These OT&E Agencies are the Operational Test and Evaluation Agency (OTEA - Army), the Operational Test and Evaluaion Force (OPTEVFOR - Navy), and the Air Force Test and Evaluation Center (AFTEC). Since the testing which is performed by these organizations is operational testing of systems, software is usually singled out on an exception only basis. However, due to the special section in DoDD 5000.3 on Test and Evaluation of Computer Software, groups which specialize in software T&E have been formulated within each organization. These specialists, in some cases, are involved with the development of new systems from the time of conception. They attend the Computer Resource Working Group meetings, Preliminary and Critical Design Reviews, and may even witness acceptance testing. Another example of the OT&E agencies' increased interest in software is a set of handbooks which has been developed by the Software Evaluation Element of AFTEC for use when evaluating the operational effectiveness and suitability of software.

Defense Contractor Interviews. Interviews were conducted with twelve defense contractors. These contractors are involved in the development of applications software, the development of support software, and the independent verification and validation of military software systems.

Applications Software Developers. Six contractors were interviewed with respect to their efforts toward developing applications software for embedded or mission critical computer systems. The customers dealt with spanned the three military services and many other DoD components. The subjects discussed included military and internal standards; requirements, design, and code analysis techniques; the levels of testing performed; tools; quality assurance; independent verification and validation; and risk assessment. Most of the testing conducted exercises system functions with very little attention being paid to the coverage achieved. Few testing tools, other than simulators, and no metrics, were found in use within this population. In general, the methods used to determine whether or not a program is ready for the next phase of the developemnt process, whether that be integration or release, are both manual and subjective.

Support Software Developers. Two organizations which develop support software were also interviewed. Although the subject areas discussed were identical to those discussed with the applications software developers, the interviews conducted with these contractors centered upon the development and certification of compilers. The major difference between the testing of applications software and support software is the degree of automation used. In each case, a standard and extensive set of certification tests are run prior to each release. Very little human intervention is needed either when running these tests or when checking the results.

IV&V Organizations. Independent Verification and Validation (IV&V) is a risk reducing technique which is applied to many major programs under development today. Four industry contractors whose primary function is to conduct an independent evaluation of the software development efforts of another contractor were interviewed. Due to the high cost of IV&V, the activities described were usually only performed for a portion of any software system. The information gathered during these interview pertained to the military regulations and standards; the scope of the IV&V effort and the time of initial involvement; the relationship to the project office and development contractors; requirements, design, and code analysis techniques; independent testing; tools; metrics; and risk assessment. The most promising information which resulted from these interviews relates to metrics. One of the IV&V contractors is working toward applying the metrics framework described in RADC reports to a major program.

Summary of Current Practices. In general, the personnel involved in the development of military software are doing the best they can with the resources available to them. Unfortunately, those resources fall short of the resources needed to produce systems which meet the required operational reliability. There is a lack of qualified personnel in the acquisition organizations to track the progress of the Defense contractors. The testing tools which could help the Defense contractors ensure that the software systems they produce are of high quality are not available. And, of course, when the budgets are cut or the schedules slip, the activities which suffer are testing and quality assurance activities.

PRELIMINARY RESULTS

Two trends are evident in the studies conducted in Phases I and II. The first relates to the state-of-the-art. Even though research and development efforts in software testing are still quite immature, a number of testing methodologies exist which yield reasonably high-quality tests of programs. Further refinement of these techniques, the development of tools for their implementation, and the appearance of new techniques should make systematic testing of mission critical software a realistic goal of every development project. The second trend relates to the conduct of software testing in practice. There is a growing realization in the acquisition community that there is a need to monitor and control the software development process, and testing is an important part of that process. At the same time, software developers do not see any helpful guidance from project offices which deals in a specific way with software testing.

A number of weaknesses in software T&E as it is currently practiced were identified quite early in our study [3]:

1. Lack of T&E Planning. When there is no planning for T&E, or when planning does not occur early enough, there is a problem in recognizing the scope of the required testing effort. This problem is most apparent in development testing. DoD guidance in T&E addresses operational requirements, but planning for thorough tests of critical software components is rare.

2. Lack of T&E Resources. Testing is labor-intensive. At the development level, studies have indicated that systematic testing consumes as much resources as the original programming effort. In addition, the development of customized test environments and other tools may comprise a small development subproject. Without adequate resources these activities can be only neglected stepchildren of the project. During operational tests, personnel support is equally critical.

3. Lack of Testing Requirements. Test requirements are most frequently formulated in terms of overall system requirements. For example, specifications that set performance criteria are difficult to test prior to system integration.

4. T&E Shortcuts. There is a tendency to shortcut testing efforts due to budget and schedule pressures. This forces incomplete testing, testing to obsolete requirements, and inadequate management and documentation of the testing effort.

5. Unrealistic Deadlines. It follows that since T&E consumes resources (including time) deadlines must be sensitive to the scope of the testing effort. In practice, requirements force unrealistic deadlines on the testing phases.

6. Lack of Independent Test Teams. In general, developers have too much involvement in the testing effort. The transfer of development personnel into testing organizations as the software proceeds from the coding and unit testing phases to integration is common. Many errors are simply carried along in this manner.

7. Lack of Regression Testing Techniques. Retesting software which has been modified is too expensive. Minor changes in large systems that have not been designed with test requirements in mind, that require human operator involvement, or that have been poorly partitioned can demand retesting far out of proportion to the scope of change. Not only does this waste scarce testing resources, it indicates that system maintenance will also be costly.

8. Lack of T&E Tools. There has been a lack of investment in software tools for T&E. Tools which are created for a given project are seldom transferred to more general settings. As a result, many development efforts have a large "throw-away" component. In addition, T&E tools such as simulators, which can bridge the gap between development and operational testing, are rarely given the support needed to be useful components of the test environment.

9. Educational Problems. There is a widespread lack of sensitivity to the special problems of software testing. Managers with hardware or weapons systems backgrounds avoid treating software as a critical system component. By the same token, technical personnel with software backgrounds often do not view themselves as part of an engineering effort, assigning software problems a special status and isolating them from standard engineering approaches.

10. Lack of Quantitative Models. While hardware components of systems generally come equipped with physical models and objective data on which to base reliability and risk estimates, the evaluation of software is usually viewed as subjective. Objective measurements -- when they can be taken -- are seldom used in the decision making process, either because of resource constraints or due to the lack of validation of underlying models.

Major revision of DoDD 5000.3 and the attendant modifications to more specific regulations and standards will have a significant impact on these problems. However, the usefulness of new guidance in software T&E will be mediated by how rapidly the research, development and acquisition communities move toward state-of-the-art application of existing technology. One of the most significant needs is support for tool development. This may involve modifying contract funding patterns, and may initially increase project costs. However, there seems to be a consensus that testing cannot be justified on narrow economic grounds. Total lifecycle costs must be taken into account. Along the same lines, incentives mut be provided for improved testing throughout the development/integration portion of the lifecycle. This may require major revisions of the development process. For example, build-test-build approaches to software components that implement high risk functions may be developed. Regulations that address detailed unit and module testing requirements will also help.

New guidance and regulations must also be realistic. If developers and testers find themselves too constrained by regulations, they will not have the desired effect. It has been noted, for example, that not all software components are created equal: some implement critical functions and others do not [3]. To require the same level of testing and therefore the same resources for all components is probably not realistic.

Software developers and requirements writers must eventually strike an accord. On one hand development groups should recognize that neither requirements nor specifications are likely to remain static -- they must learn to cope with change. On the other hand, those who formulate requirements cannot assume that software is arbitrarily malleable: software changes may be as expensive and far-reaching as changes to any other system component. System retests and budget/schedule shortages are currently victims of the tension between requirements and development groups.

Finally, basic research is needed. There is no quantitative risk model for software. Software measurement techniques are still at an early stage of development so that objective data is still only a goal. Testing techniques, methodologies and tools need further development. The cost-quality tradeoffs for various techniques must be quantified if developers and testers are to make a choice from among the existing techniques.

CONCLUDING REMARKS

It is certainly feasible to formulate new DoD guidance for software T&E. We have already sketched the state-of-the-art and current practices in software T&E. New guidance must address the problems listed in the previous section, either directly or indirectly by encouraging new technology and acquisition procedures. With such encouragement, the technological "window" will move to provide more effective techniques for software T&E. New guidance should be general; development testers and operational test groups should not feel bound by mandated test procedures that fit neither their application nor their environment. The exact form that such guidance will take and its ultimate effect on the reliability of future military systems awaits further study.

REFERENCES

[1] Research Directions in Software Technology, P. Wegner, editor, MIT Press, 1979.

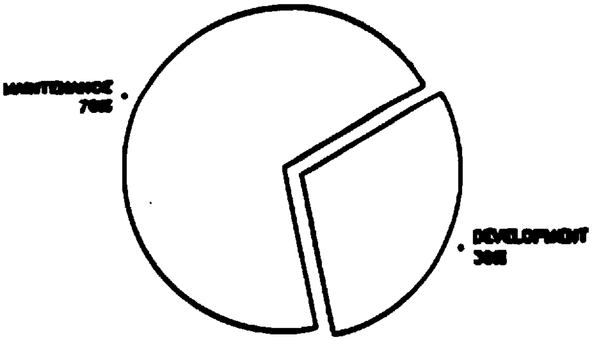
[2] "Candidate R&D Thrusts for the Software Technology Initiative," S. Redwine, E. Siegel, and G. Berglass, Department of Defense, 1981.

[3] "Transcript of Proceedings: Software Test and Evaluation Workshop," Defense Systems Management College, Fort Belvoir, VA, March, 1982 (Technical Report GIT-ICS-82/13, Georgia Institute of Technology, Atlanta, GA, 30332).

[4] "STEP State-of-the-Art Overview" (to appear 1983).

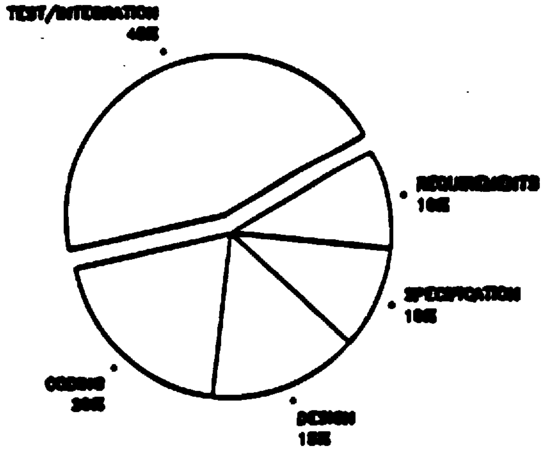
[5] "STEP Current Practices Overview" (to appear 1983).

EMBEDDED SOFTWARE COSTS
FIGURE 1(a)



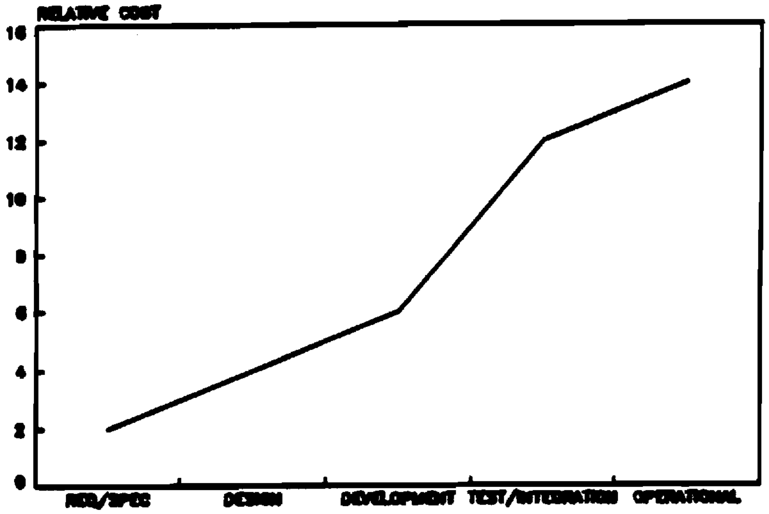
OPERATIONAL LIFECYCLE

EMBEDDED SOFTWARE COSTS
FIGURE 1(b)



DEVELOPMENT CYCLE

RELATIVE COST OF ERROR CORRECTION
FIGURE 2



ABOUT THE AUTHORS

RICHARD A. DEMILLO is Professor of Information and Computer Science at the Georgia Institute of Technology in Atlanta, Georgia. In addition to being the Principal Investigator for the STEP Phases I and II contracts, he teaches and conducts research in software engineering, computer security and theoretical computer science. Dr. DeMillo received his Ph.D from Georgia Tech in 1972. Prior to returning to Georgia Tech in 1976, he was on the faculty of the University of Wisconsin. He has also been associated with the Los Alamos National Laboratory and with a number of government and private organizations as a consultant.

R. J. MARTIN is a Research Scientist with Control Data Corporation, Government Systems, in Atlanta. She is currently Project Manager for the STEP Phases I and II subcontracts. Ms. Martin is completing the M.S. degree in Operations Research at the Georgia Institute of Technology. Prior to joining Control Data in 1981, she was affiliated with IBM Corporation in Houston and Atlanta. In addition to her interests in software T&E, she is currently directing an NSIA study for the Air Force Electronic Systems Division on C2 Software Development and Acquisition.

NSIA/OSD
NATIONAL CONFERENCE ON SOFTWARE TEST AND EVALUATION
PANEL DISCUSSIONS

Introduction

At the end of formal presentations for each of the three days of the conference, the speakers* were assembled as a panel to answer questions from the audience. The panels for each day were:

February 1, 1983: Dr. Richard A. DeMillo, Chairman

Dr. Edward Miller	Software Research Associates
Dr. Richard J. Lipton	Princeton University
Dr. James F. Leathrum	Clemson University
Dr. Leon Osterweil	University of Colorado
Dr. Leon Stucki	Boeing Computer Services
Dr. Victor R. Basili	University of Maryland
Mr. Ralph San Antonio	Dynamics Research Corporation
Dr. Martin Shooman	Polytechnic Institute of New York
Ms. Carolyn Gannon	General Research Corporation

February 2, 1983: Ms. R.J. Martin, Chairman

Mr. Marion F. Moon	Hughes Aircraft
Mr. Raymond J. Rubey	Softech
Dr. Peter Wegner	Brown University
Dr. Douglas Giese	TRW
Ms. Marilyn J. Stewart	Booz-Allen and Hamilton, Inc.
Captain William P. Nelson	USAF Electronic Systems Division
Mr. James Hess	US Army Materiel Development and Readiness Command
Major Edward E. Stevens	USAF Systems Command
Lt. Col. Michael A. Blackledge	USAF Test and Evaluation Center
Mr. Sam DiNitto	RADC

* In addition to the presentations given by the authors of the papers in the preceding pages, presentations were made by: Mr. James Hess (Test Procedures and Project Management); Mr. Sam DiNitto (Software Technology for Adaptable, Reliable Systems); Colonel J. Frank Campbell (Army Perspectives); Captain David Boslaugh (Navy Perspectives); and Colonel Edward Akerlund (Air Force Perspectives).

February 3, 1983: Dr. Richard A. DeMillo, Chairman

Mr. Donald R. Greenlee	Office of the Director, Defense Test and Evaluation
Colonel J. Frank Campbell	US Army Materiel Development and Readiness Command
Captain David Boslaugh	Navy Materiel Command
Colonel Edward Akerlund	Air Force Systems Command
Ms. R. J. Martin	Control Data

The following is an edited transcript of those panel discussions. Questioners were given the opportunity to identify themselves and their affiliations. Those who did not identify themselves appear in the transcript as anonymous questioners.

1 FEBRUARY - PANEL DISCUSSION

DR. RICHARD DEMILLO (GEORGIA INSTITUTE OF TECHNOLOGY): I have a question for Lee Osterweil. Your analogy to carpentry tools breaks down in the following sense: in order to build a house you have to drive nails into the wood, to build software you don't have to have the tools. Does that figure into your definition of what a tool is?

DR. LEON OSTERWEIL (UNIVERSITY OF COLORADO): I guess I would claim that to build software, you really do have to have some kind of tools; you need a compiler and things like that. To build a house you really don't need to have anything very much fancier than a screwdriver and a hammer. The more you have, the better the product looks at the end. I think that is the point. The quicker you can get it done, the better you can get it done. People built houses long before they had fancy collections of tools. People build software, today, even without fancy collections of tools. It is simply a question of being able to do it better when you have better tools and to do it more effectively.

MR. SAM REDWINE (MITRE): I have a lot of interests, but I'm going to address one particular problem because it came in front of me recently. I was advising some people about what they ought to put in an RFP, and I said you clearly ought to put in a requirement to collect a bunch of data, the kind that you know is good to collect. Then I looked around and tried to find, from DACS or elsewhere, a description that could be contractually referred to, and I found none. I wondered if anyone could address that problem.

DR. DEMILLO: Is that to anyone in particular, or to the panel?

MR. REDWINE: Anyone who can answer with a positive answer, but I don't have any preference.

DR. MARTIN SHOOMAN (POLYTECHNIC INSTITUTE OF NEW YORK): You're talking about specifications and how to submit that into DACS, what kind of data, is that what you're referring to?

MR. REDWINE: Well, that was my simple first approach, but when I talked to various people, it turns out, for example, they're right now in process between their old definitions and their new description of what sort of data they may want. That is some time away in the frame of when you look at the schedule on which the RFP is being prepared. Nor is it entirely clear to me that their definition is the one that we should reference. It is certainly the one that came first to my mind.

DR. SHOOMAN: I think that there has not been sufficient work done between the analysts who would want to use the data and the people who would collect it. There has not been specific movement afoot to collect data in a form so that it would fit model 1, 2 and 3, and so that it would satisfy the analytical questions of analysts A, B, and C and modeler X, Y, and Z. I'm sure that if you tried to do that you would find that you couldn't satisfy them all, but maybe you could satisfy 50-60% of that group. I think they have tried to do that, but to my knowledge, it hasn't been perfected yet. Maybe in the new effort it will be. Some of the studies that have been done with the data, in fact, have found that key data such as the number of test hours or number of failures during tests, which are needed for a lot of the models, were not recorded. Just the total number of errors were recorded, and no time sequence of when those errors occurred were recorded. It just said that over a period of 15 months we found 600 errors. Nobody said that during the first month we found this many errors, the second month we found this many errors, and we tested 25 hours during the first month, 50 hours during the second. That is the sort of data most needed. That was there for some models, but not for others. Perhaps, that's the newer data that's being recorded, but, I don't know.

MR. REDWINE: Let me ask the question differently to the entire panel. Let's say that you have the job of writing the paragraph in the RFP that is going to require builders of the software for DoD to report data. Not that you have the problem in the narrow sense, that it has to work in your model, but you have a broader expectation of what might eventually be done with it. What would you do?

DR. VICTOR R. BASILI (UNIVERSITY OF MARYLAND): I think you have to have a very specific idea of what you want to do with the data before you collect the data. That is what Marty is saying. You have to first establish what your goals are, that is why you want the data, Then you establish what your hypotheses or models are that you're going to measure. Then you have to look at what the ingredients for that particular set of models are and that is how you specify what you want. But, you have to choose those models beforehand.

MR. REDWINE: OK. I understand that, if I'm using the data, particularly in this development. But if I'm collecting the data for posterity, as well as having potential uses in this development, what do I do?

DR. BASILI: Well, you can collect it for posterity, but that may not be useful. I want to argue that that is not the way to do it, not just to collect data for posterity, but to have a goal from the very beginning. Otherwise, you will end up with the wrong information. Once you specify what your models are going to be or what your questions of interest and hypotheses are going to be, that drives the data collection process. If you haven't thought about that issue beforehand, sure enough there is going to be another model you want, and you will not have the data for it.

DR. DEMILLO: Anyone else?

MS. CAROLYN GANNON (GENERAL RESEARCH CORPORATION): Let me just add that that is one of the issues I was trying to get across when I said that the customer and the project manager, and the programmer, should get together at the project outset to determine what the goals of the data collection activity are. Posterity can be one of those goals, however, it can't be so general that you get "the world". When I gave the example of two missile projects: you're going to collect data now and pay the little bit of extra overhead on collecting the data now in hopes that on the next missile project there will be enough similarity that you can benefit by the kinds of errors that were recorded. You still have to narrow your scope.

DR. ED MILLER (SOFTWARE RESEARCH ASSOCIATES): I think the comment I would make is that even though you can specify what you want in the way of data, it is going to be difficult because of the people who are actually doing the work to record and collect the data. Then a second layer of difficulty occurs when you want them to release that to you. It kind of goes against tradition. It's always been the tradition that the developer of software is up to his own devices. So, that includes private data. One way of achieving this is perhaps to incentivize and contractualize, but certainly to make sure that you've made a good relationship with the person who you might ask to supply the data so that they will give it willingly and not create a conflict.

DR. DEMILLO: Vic, does your remark mean that historical data is suspect?

DR. BASILI: No, that's not what I meant. What I meant is that I'm going to specify what data I need to collect. I have to know beforehand why I want it and what model it's going to fit so it can be specified. What I am saying is that there is an infinite amount of data that I can collect. So if I'm going to collect for posterity, I would have to collect everything that happened. It's too expensive, and I can't do that. So, I have to single out what I want by setting some set of goals and base some of those goals on models or whatever it is and start to collect data driven by that specific purpose. I can't always second guess.

DR. DEMILLO: What about the data you are using now? Was that collected for a specific purpose?

DR. BASILI: It has always been driven by a set of goals. We have a paradigm: a set of goals, questions of interests, our hypotheses followed by a set of metrics, and we check that these are collected by the forms. Then, in fact, you had better go back and make sure you're interpreting that data in the context of the goals you established for it, because that colors how you collected that data.

QUESTIONER: How can you make it public?

DR. BASILI: I'm saying that you've got to establish a set of goals. My goal may be, for example, that I'm interested in evaluating the reliability in the model I'm choosing. It may be a very specific model, the mean time to failure, in which I need time between failures, say in computer time rather than in calendar time. So I've got to specify that I need the computer time, I need the errors, I need certain classes of errors. But that's based on that model, based on that goal which is that I want to be able to evaluate the reliability. I can't just say let's collect some numbers; I must say let's collect time. What time? It's got to be tied back to a very specific model of time.

DR. SHOOMAN: Let me just comment very briefly on the aspect of historical purposes. There is a set of data that was taken back during the 1950's for hardware reliability that plotted the reliability of a group of electronic circuits versus the number of years of experience of the designers. It showed that there was almost a direct proportionality between the more experienced designers and the more reliable circuits. Now, suppose it turns out that you want to go back and study that fact with regard to data that has been taken, with regard to software errors. Unless somebody said, when we first started collecting those errors, I want you to also tell me the number of years experience of the programmers who worked with these programs, there is no way of doing it. Alright, but, unless somebody has some suspicions that this is going to be worthwhile to study, that it is important, then nobody would bother reporting this data. So, it's relatively impossible to record all those relevent parameters. You have to start with a collection of objectives to satisfy a broad enough interest so that it is worthwhile, and work with that. Then as you go along, you may very well find that in 2-3 years, you need more data because people have learned more and people want more data on other things. This is an evolving concept. People have been collecting hardware reliability data since 1947 or 1948, and they still go on doing it. They have to get more data. There is a set of military handbooks on failure rate data, this started out as version a, b, c; they are up to d now. But, why do we keep doing this? Because they are learning more, the data keeps changing. The data collected now is far more sophisticated than the data collected 30 years ago. A lot of money and a lot of effort have been spent. Much, much more than has been spent for software. I think this is one of the reasons why we have made slow progress on software. There was a relatively large effort in the 1950's to collect failure-rate data. I don't think there has been anything like that in terms of scope and size to collect software data.

DR. DEMILLO: I think it is time to go onto the next question.

QUESTIONER: I have some questions for Dr. Stucki on the Argus. Is Argus available to the general public, does it automatically do dataflow diagrams for you, are the hierarchy structures linked to the dataflow diagrams, and what microprocessor does it run on?

DR. LEON STUCKI (BOEING COMPUTER SERVICES): The answer to the first one, "is it publically available?". It was done on internal money, so that probably answers that. Is it available? Everybody has a price at some point. OK. "Are the dataflow diagrams linked to each other?"..Right now, they are not. The linkage is between the dataflow diagram as an entry vehicle for entering information into the design data base. That link does in fact work between levels in the diagram. Right now, they are entered as separate diagrams. They can have commonality between them, and all of that information is linked in the database. But, we don't decompose the dataflow diagrams with the tools automatically, today. That is something that is planned. Do you have any other questions?

DR. DEMILLO: What was the machine?

DR. STUCKI: The machine, currently, it is running on an ONYX, which is based on a Z8000, but it is really not dependent on that particular machine. In fact, we have large portions of it working on a VAX right now.

QUESTIONER: What language is it written in?

DR. STUCKI: Most of it was originally written in PASCAL. We had too much trouble with portability. It's almost all written in C now, and the remaining modules that are currently in Pascal are scheduled to be converted to C.

QUESTIONER: One other question. How much did it cost to develop?

DR. STUCKI: I don't know. I've tried not to add the thing up totally. It's pretty expensive.

DR. DEMILLO: Next question, please.

QUESTIONER: I have two questions. First, to Dr. Shooman, in the references at the end of your paper, which is the book that you kept referring to?

DR. SHOOMAN: That is the one that's "Software Engineering Design, Reliability, Management", MacGraw Hill, 1983.

QUESTIONER: The second question is much more general. This is in terms of software quality assurance, generally. I think one of the building blocks is persuading creative software developers to write software in a disciplined manner. I think that is not easy. I think it poses cultural problems, introducing this kind of discipline, and none of the speakers have addressed this issue at all. I would like to hear some comments.

DR. OSTERWEIL: What I think we are dealing with here, as I said in my talk, is a basic industry. It has to be approached that way. I feel that there is a sort of Darwinism that's going to be at work here. I think of the early days, when flyers used to fly around in leather helmets and silk scarfs and take a lot of risks. Those people don't fly commercial airliners anymore. The analogs of those people in the software industry are slowly but surely going to be ground out. I see signs of this happening already. There are a lot of ways in which it is going to be accelerated, I suspect. I really do believe that as we put tools at people's disposal, whose job it is to relieve people of a lot of the tedium, we are going to find that people are going to be much more willing to be responsible and to submit to procedure because an awful lot of humdrum stuff is going to be taken off their backs by our computing systems. Slowly, we are going to find that responsible people are going to stay and be responsible. People who are less responsible will have a lot of push to have a machine do a lot of the tedious work for them, and the ones who won't submit to that sort of thing just won't be around for very long. We are building one of the cornerstones of society, I honestly believe, and all the forces are there for us to do it in a responsible way. All of the things presented here will in one way or another contribute to making it easier to do the right, responsible kind of job.

DR. MILLER: I just want to comment. I think that it is less difficult than you may think to get people to change their ways. I agree with Lee that it is going to take some time. Most of the time if you change the boundary conditions, the underlying, substrata of assumptions for the development of code, then people behave pretty much in the right structured discipline. The statistic that we often think of that illustrates this is that the cost per man-month today is about equal to the cost to buy and install a million instructions per second in hardware. So, if you teach programmers that the right thing to do is to burn machine cycles because that is cheaper than people cycles, they will build smaller modules. That takes some head shaping. It is a head shaping process and it takes some time to convince people of the overwhelmingly compelling advantage to structured techniques and disciplined techniques. But, it does work out, I believe.

DR. STUCKI: I don't know if you noticed or not, but one of the guiding philosophies that we espoused was reasonably well stated in one of the slides that I threw up real quick, and you probably didn't get a chance to read. But, my philosophy is if you are doing a good job on designing and building tools, it ought to be, from the user's perception, easier to get his/her job done using the tools or the tool system, than any other way. If you achieve that, you create a usable tool. If you haven't, you have an unusable tool. I don't think they have to be mutually exclusive. I think that if you define your methodologies and tools properly and implement them in such a way, that the average person will realize that "hey, I really got this thing done faster this way", then, that's really where it's at. That is where we are heading.

MS. GANNON: I think we were really fortunate to have the speaker at lunch time that we did, because education is definitely an area where software engineering practices can start at an earlier age. I think that it is very important that they be taught in the universities. However, I think the panelists who have spoken on your question so far have made the situation sound a little bit better under control than it really is. This morning, Dr. Lipton talked about rapid prototyping. In the last couple of years, we have all gotten used to the software development waterfall chart lifecycle engineering approach to developing and testing software. Yet, he was bringing up actually quite a different approach, which I think possibly is just as viable and may be even more relevant to developing large systems than the usual waterfall chart with no feedback loops. So, I don't think that, in spite of the tools that we have developed over the years, we have even hit upon the best development methodologies yet.

DR. SHOOMAN: Let me just comment very briefly on this too. In terms of education, there is a major amount of education to be done in middle and top management in companies. You don't get into middle and top management in most companies unless you are over 40. If you are over 40, you never took any software courses in school, by and large. You probably got most of your experience doing hardware development. Unless you're an exceptionally inquisitive person with a wide viewpoint, you probably didn't learn much about software development. So, here you are in charge of a large hardware/software complex, and you understand the hardware because you've done it before. But you don't understand the software. And now, you got burned in the early 1960's when people wasted all your money, and didn't deliver a product. You learned that what you must do is ask them how much of the code has been coded and tested. If it is 50%, and they spent 50% of the money, it's OK. If they spent 40% of the money, they are heroes. If they spent 60%, they are bad boys. Now, you go ahead and do a top-down development for a manager like that, and you spend 40-50% of the money, and he asks you how much of the code has been written and tested. You say about 5%, I wrote the control structure, I tested it. He's not going to even listen to the rest of what you say. He's just going to be thinking of, who do I replace this clown with?

So, before you do something like this, develop, and use any new tools or new development philosophy, you better be pretty sure that management understands what you are doing and understands that when they ask you questions, they have to ask you different questions. If they had asked you in a top-down development, at what level in the design process are you, have you designed the interfaces, have you designed this, how long do you think it will take you when you're finally ready to code ... those are questions you can answer. So, in the same way, using any tools, unless management understands, you're going to be in great difficulty. Because even if you get the programmers to do it, management doesn't understand. They know that when management tells them not to do something, they don't do it, even if they think it's good.

DR. STUCKI: I can't let it go just quite like that. There are a couple of things that I think are totally overlooked here, and with all due respect, I disagree with the last statement somewhat. I think that, the over 40 part is probably OK, and the background is probably OK, but I think there is one thing that is really important ... we need to tell management and make sure that management understands that there is more to software than just counting the lines of code. That's why, when I said the phrase "computer-aided design and manufacturing for software", I made a big point of the fact that I'm producing a lot more than code, I'm producing documentation and a lot of other things. So, when I go in and give my pitch and say I spent 50% of the money and here is 5% of the code, I can also say here is 85% of the design documentation, by golly, and here is 75% of the test plan, because I've already thought about the test plan ahead of time, and stuff like this, then I think it is a new ball game. I just had to get that much in.

QUESTIONER: I would like to address an issue to the panel at large. One observation is that there are many levels of software testing, and I think perhaps we talked about software testing at one edge of the spectrum. But, particularly with embedded microcomputers, etc., the distinction between software testing and system testing blurs somewhat, particularly when you begin to do acceptance testing, like flight testing, sea trials, etc. And, the comment that comes to mind, I'm not sure that it's been thoroughly addressed, is that testing is hard work. I think that was one of the first points made at the conference, and it seems to me a very key issue of that hard work is trying to come up with very detailed test specs and test cases that really relate back to the original system requirements or B5 specs, Part 1 specs, whatever, as well as the more detailed lower level specs. It seems to me that only in this way could the customer and the company management be reasonably assured that when you're through testing, the software really does what it's supposed to do. I think that as of yet this hasn't been addressed very heavily.

DR. DEMILLO: Comments from the panel?

DR. MILLER: I really couldn't agree more. In fact, I'm not apologizing for the software engineering community, but there's a certain sense of this, maybe my colleagues here will nail me for it, but there is a certain sense of solving the problem that's rather easy to solve rather than those that are harder to solve. So, while we can analyze individual program structures and figure out test cases in a fairly simple manner, there being no general structural theory of systems and there being no mathematically tractable body of techniques for handling system level behavior, we kind of look at that and say, that isn't quite as interesting, so we don't work on that one. There is a kind of avoidance of by far the most critical issue facing someone who is purchasing a system. That is how to answer in a simple yes or no manner; "Is this acceptable or not?". I think, however, that may be an oversimplification. I see a lot of systems which are built according to good techniques, and then there's this formal acceptance test. My suspicion is that the formal acceptance test is a piece of window dressing that probably ought to be eliminated in favor of a more detailed check-out phase or several months of investigative analysis of the program behavior. I don't know, maybe that will get the discussion going.

DR. SHOOMAN: I could suggest a different kind of acceptance test, which I propose. That is that the customer develop a set of N test cases, say 110, whatever number it is, including some stressful ones. Let's say he does this testing in three phases. The first 10 cases test major features of the program. You run these first 10, and the developer has these first 10. The only purpose of that is to make sure he brought the right disk or right reel of tape with him. You know that something is not strange, he tested all those and expects those to work. Then you take another 100 that he doesn't know about. You run those, and perhaps 95 of those 100, nothing is perfect, so you can't expect all of them to work, if 95 out of those 100 work, it passes. Then you get him to fix up those other 5 cases, and to take care of those bugs is a minor manner. If it doesn't pass, then perhaps he pays a penalty equal to the cost to make up a new 100 test cases. You give him the first 100 and say go back and do your homework again, and come back and we will have the new 100 test cases for you. This is perhaps a way of testing the software. But of course, this only is useful if you can make up a representative set of test cases, and if you know how big that number is, and if you know whether 90% is a passing grade or 80%. If you put a ridiculous figure on that, then nobody passes. You do yourself no good, you do the developer no good.

DR. MILLER: Your concept is absolutely correct, but the practical reality is that there are very few formal acceptance tests which don't succeed, and the reason is that the numbers of tests involved are small enough, and the people who are running the tests have run them in advance, and everything is successful. Yet systems still are accepted with enormous error content.

MR. RALPH SAN ANTONIO (DYNAMICS RESEARCH CORPORATION): I wanted to point out that we do have a presentation tomorrow by Dr. Giese from TRW who will be looking at the impact of new hardware technology on this testing process. Also, another thing. I agree with your overall statement about the general void, particularly in the area of integrated hardware/software testing. In fact, another presentation on the docket for tomorrow is one by Marilyn Stewart, which will be addressing some of the current problems in the acquisition framework and how some of the new policies and procedures that are being promulgated will overcome those problems. One area which I don't feel is adequately addressed in the new forthcoming policy is the area of integrated hardware/software testing. You mixed a couple of things in with your question, because you were talking about hardware/software issues at one point, and then later spoke of tying it back to a software requirements specification, and, subsequently, a higher level specification. If we talk about the software requirements specification, you've allocated the requirements to the software. You know you are dealing with software. But when you truly go back to the higher level systems specification, then I think that is where we do have some real voids in the system, right now.

DR. DEMILLO: Let's move along to the next question.

MR. KENNETH MOORE (AT&T): Earlier today, Dr. Lipton was talking about the rapid prototyping, and it was mentioned that the prototype was the seed or the beginning of the software project that is now grown from that prototype. What I would like to find out from the panel is some comment on the suggestion that the use of a prototype be solely to validate the user requirement for a system and to aid in the development of the metrics to be used in the test phase, and then to end the prototype phase at that point and begin the full system development from those requirements, instead of building a prototype and then incrementally adjusting it.

DR. OSTERWEIL: You are describing what I sometimes euphemistically call a software life spiral. The main trouble with most life cycle models, as a number of people have observed, is that they don't have any cycles in them. Everything goes from requirements to a design to code, and then to a complete operational system. What you are suggesting, in fact, is that in the beginning we go through this procedure very quickly and produce something that actually runs and is intended to be the back-to-the-beginning, namely, the requirements process. And that we then take this as being a more detailed requirement specification and we run through the cycle in more detail, therefore more slowly, produce another prototype at the end, which then feeds back into the beginning, so my spiral spirals out and gets larger and slower. There are a number of people I have spoken to that claim they have never write finished systems, they just write a succession of prototypes, and that supports the point of view that maybe there is something to be said for this. My own perspective, namely the tools perspective, says that this is probably fine. What we should do is simply facilitate the process by enabling the succession of stages to go more easily, by means of more tool support. I don't think there is anything much wrong with the paradigm at all.

DR. STUCKI: I think you pointed out something rather interesting and I think we reinforce the way most people do, in fact, take for granted that you are going to use the prototype for the next system. This is almost implied by what Lee was saying. I think sometimes it would be nice to associate with a prototype, like in Mission Impossible, the idea that it will self destruct in so many minutes after it's done with whatever you want to use the prototype for. I think we, for some reason, have a very hard time throwing the prototype away, and it's probably a mistake on our part that we don't do that more often.

DR. BASILI: One comment, and that is back to the goal driven business. There are different ways of talking about a prototype. One is one that you create and throw away, another one is one that I start taking a subset of the problem and iterate until I build the whole system. Why I would pick one approach over another one, and I don't mean there are just two approaches, depends on what I want to do. For example, if what I am interested in is getting an overall statement from the user's point of view of the full system, I might pick a prototype that I can throw away, i.e., build the whole system to throw away. If I am interested in getting some experience with development or low level experience with a user, then I might build a subsystem. I feel I know how I could do it right. So, again, it is goal generated. There are a lot of different options, I have to decide what I care about the most and why I'm building that prototype. One option of building a full prototype is that I also have a simulation that I can use, that I know what my test results should be, I have my oracle. So, when I finally do my testing, at least the results should correspond with the earlier process.

MR. SAN ANTONIO: When I was listening to you formulate your question, it appeared that what you had recommended was something similar to the classical way we acquire or theoretically acquire systems in software now. That is you go into a phase, sometimes it is referred to as a validation phase, where you develop prototypes and validate the concept and, yes, this is in fact what we want to build. Then, you throw that away and go into a later phase where you go out and build the prototype production models. In that sort of system, whereby you say, OK once we finish this we are going to scrap it and start over, I think the user tends to get the feeling that he's got one shot in the barrel after you've thrown it away. Therefore, he wants to get all his requirements packed in that one development activity. What Dick was recommending this morning was a slight variation. That is, you start out with the premise that you are going to essentially keep modifying that system, and, lo and behold, you may in fact throw large portions of it away, but, as a concept, you're not saying I'm going to finish it and throw it away because that conjures up all kinds of ghosts in people's minds.

DR. DEMILLO: Next question, please.

MR. NELSON ALLAN (GENERAL ELECTRIC COMPANY): I have a couple of questions. Software quality requirements are very high in some flight critical control systems in aircraft. You may want it be extremely improbable that an error could occur that could cause a significant event, possibly an aircraft crashing, which sometimes comes out with something like 1 in 10 to the ninth probability of one thing in 10^9 opportunities. I wonder if the panel feels that the models that were discussed, particularly by Dr. Shooman on reliability, are valid to this range of probability of error; if it is not available now, when could we expect it would be in the future? And secondly, Dr. Stucki, when will the aircraft companies be willing to accept software for flight critical systems with known, expected errors in them?

DR. SHOOMAN: Let's take the system that we, as the general public know best, which is the Space Shuttle, which is again a flight critical system. If the re-entry control system fails, the astronauts are stranded in space. Given the catastrophic situation which may occur, what was done? Five computers were on board, with four on-line, the fifth was stored somewhere from what I read, I don't know if it was in the glove compartment or underneath the seat. It was not on-line. The only reason I can figure out it was not on-line was they were afraid that a power surge might wipe out four of the computers and they would have this one that they could energize and put up. The computers were in a redundant checking arrangement, using a voting scheme, they were compared. One of the four had a different program. It had redundant software. Now, I guess I would interpret your question as, could one analyze a configuration like this in terms of software reliability. The answer is yes. I'm not sure how accurate the model would be, because people don't write redundant software, very often two different algorithms, implemented as programs by two different groups to do the same job so that, externally, they should give the same answers, yet have different algorithms so that presumably, at the same time, wouldn't have the same software. Could we analyze this with the kinds of models I've talked about? Yes, presumably. How well would it turn out? I don't know, I've never tried it. Do I think it would be accurate enough to ensure the very, very high reliabilities? To be honest, I would have to spend a month or so trying it out, studying it, and then maybe I could give you an intelligent answer to that. On the surface, there is no reason why it could not be applied to such a problem. How would the details work out? You see one critical problem in this case, you are going to only get very, very high reliability in software if you have either tremendously low error content, which probably would not apply to these models, or if you have redundant software. In the case of redundant software, what you are most worried about is common load failures, in other words, something that causes both software programs to fail. If a portion of the algorithm is the same in both cases, and it has an error, your wonderful redundant software didn't help you one bit. One would have to look for those. The answer is I don't know. It would be very interesting to try.

DR. DEMILLO: The second question is for Leon.

DR. STUCKI: Let me rephase it to make sure I understand. Basically, you asked the very interesting question, "when will the airplane company be willing to accept software", right?

MR. ALLAN: With known errors in flight critical applications.

DR. STUCKI: OK. I'm not sure about the known errors part, but let me comment on the following. You probably realize through the advertisements that Boeing has produced two new airplanes that have just recently been certified. Don't quote me, I'm not a Boeing spokesman on this because I don't know the exact details, but I understand that there are somewhere between 60 and 80 microprocessors on the new airplanes. But, they were certified. Most of the functions that Boeing has chosen to have automated assistance with have manual backup procedures and techniques. It is different from, say the Space Shuttle, which I was told, and this is the way I understand it now, there is no way on earth to land that thing manually. There is no manual system for landing the Space Shuttle. That is not true with the commercial airplanes. If all of the computers failed, they can still land. In fact, they were certified without the flight critical software systems being certifed. It's not to say that it's an easy problem. The flight critical functions that were software controlled have not yet been certified and constitute, in my opinion, a rather interesting dilemma. They have certainly caused high level management in Boeing to sit up and take heed of what's going on. One thing that may be interesting. Some people won't like to hear this, but when I first came to Boeing in the mid-70's, I started hearing about what they were planning to do with respect to this. At that time, I asked them the question, "well, have you defined some software standards and standard languages to be used?", and the answer was "no". I said that was going to be a real cause of concern, and now the people, I think, agree with that. Many of the systems are coded in different languages and all sorts of interesting problems occur because of that.

DR. RICHARD J. LIPTON (PRINCETON UNIVERSITY): To add two cents to your question, I think it is a very good point. I think most of the tools in testing have the basic assumption that all tests, rather all faults or errors, essentially have the same cost or the same value of importance, which is clearly false. I think what we need are testing techniques and tools that will allow you to be very selective and say, well, I have certain properties of my system which are mission critical. Faults must not be there or I must have some great confidence that they are absent, but other faults will be less critical and I will live with some reasonable range. So, we need to have methods that can weigh faults in a very non-uniform way. That's the only way to probably get the kinds of reliabilities that you want in a cost effective manner.

DR. DEMILLO: I would like to ask the question of what the questioner had in mind when he said, "accept the software?"

MR. ALLAN: Well, it's kind of hard to get people to accept software that has an error in it in the anticipation that if the error were to occur at the wrong time in the flight the airplane would crash. Nobody, either the airlines or the FAA would normally accept software like that. You have to demonstrate that it's error free; I think everybody in the panel agrees that it is very hard to do that. So, the prediction model that Dr. Shooman was presenting was one approach to show a very low probability of errors. We have to talk about extremely low probabilities; I guess the model he mentioned is not quite adequate for that few a number of errors in the software.

DR. STUCKI: One quick point. I believe that if I'm not mistaken, most of the FAA procedures are predicated around the probability of some event occurring being extremely low. I don't believe they are predicated upon the impossibility of any particular event.

DR. SHOOMAN: I don't think anybody can logically or philosophically talk about hardware which never fails, therefore, you can't talk about software which never fails. It's only the terms. I was under the impression that the FAA and the airlines industry did not use the term failure, they used the term non-scheduled maintenance. So, in other words, we would talk about the number of non-scheduled maintenances to the hardware being very low, we talked about the number of non-scheduled maintenances to the software being very low, and that's the way to approach it. Whenever somebody says "zero" failures, when we know that philosophically it can't happen, that, to me, is like saying perpetual motion. You don't make progress in mechanical engineering by looking for perpetual motion machines. You make progress by looking for low friction bearings that have friction levels of 10 to the minus something. The same way, I don't think you make any progress by talking about no software failures.

DR. DEMILLO: I for one have to take an airplane home. Next question, please.

MR. SAM BERNARD (GENERAL DYNAMICS): The first question ... as far as testing goes, which people in an organization should write the test procedures to do the test? It's been stated by several authors that your best and most creative people who are software writers should be pulled off to write the tests for the software. Does the panel have any opinion on that.

DR. DEMILLO: The panel may want to respond, but tomorrow we will have speakers on managing the testing process. Those speakers may also want to respond to this question.

MS. GANNON: Could I just take a shot at what my opinions are? You mentioned test procedures, but I think you would like to not limit it just to the procedures but actually whole test plan, because the procedures, of course, just tell you how to carry out whatever the test criteria are. I feel that the customer or the user should have to shoulder the burden of specifying the acceptance criteria. If the person cannot state the criteria by which he is going to judge the product, it's hopeless. You'll never agree that the product meets the specification, so I think the customer should come up with the acceptance criteria or at least be guided into it by the developer, but, agree that he will take those as the acceptance criteria. Then, based upon this now written down set of acceptance criteria, you can develop a test plan. It should have a section of functional tests. If you use a specification language you may have a test for each part that then is developed and is specified by the language. It should also require structural types of tests which then come out of the way the software is designed. So, I think it is a cooperative effort that should start with the user, who is the customer, coming up with acceptance criteria, and then a quality assurance group within the developing organization coming up with a test plan. The QA group has to know both what the product functions are going to be and also what the software design looks like so they can make sure the design is carried out.

DR. STUCKI: I would like to add one thing. I think that what she said is right, but I would broaden the definition of what she said is the customer to include any agent of the customer who is involved in specifying the requirements for the system. For each and every requirement for that system, that person who documents that requirement has the responsibility, in my opinion, to include with that requirement, the measuring stick against which that requirement is going to be assessed. If the customer knows that, great. If the customer doesn't, and defers to a technical assistant of some sort, then that technical assistant has as a part of his/her task to include the yardstick against which the requirement is going to be judged. This, together with possibly an independent view from an independent quality assurance perspective is what I would think.

DR. MILLER: I was going to add that if you interpret your question in a slightly different way as far as testing goes, who should write the test, meaning who's good at writing tests, I could add from our experience that people who are essentially mathematically skilled seem to make the best testers of programs because they have the best analytical skills, the most precise and careful discipline qualities, and trained forms of thinking. This does not mean that programmers are not good at finding errors in programs, it's just that it's not the primary indicator, analytical skill is.

MR. BERNARD: Second question. I noticed that you all advocate a top-down design and testing approach. When should a bottom-up approach be used and when should stub programs be used to fill in when the code that's necessary for a given module isn't developed as yet?

DR. OSTERWEIL: I really think you're asking basically the same question as far as I'm concerned. The answer is that there is a need for both kinds of things to go on. I think that when you talk about bottom-up testing, you're basically talking about acceptance testing, basically talking about having the user or purchaser exercise the thing and see whether it seems to work or not. That doesn't rule out the necessity, quite the opposite, for doing the other sort of thing, mainly looking to see that the thing which has been built is structurally sound. There is a definite advantage to using the so-called life cycle approach, and many people have objections to doing this, but the life cycle approach does indicate that the development of a software program goes through phases. Some of those phases, in my opinion, are simply not "customer land" phases. You get down inside detailed design, then you are really looking at something which has to be thought of as a top-down activity. It may not be carried out as a top-down activity, but it has to be thought of, when it is complete, as being a top-down activity, and it should be subject to top-down analysis. At that point, you are looking at testing which is done by some internal organization, which is good at doing structural analysis because they are good software engineers. The bottom-up testing is the sort of thing the customer does at the end to see that the product is really is doing what the customer wanted it to do. I think at different points in the life cycle, you do different things.

MR. BERNARD: Bottom-up testing should only be done by the customer is what you are saying?

DR. OSTERWEIL: I don't think it has to be done only by the customer, but it certainly has to be driven by the customer, as was observed earlier on. The customer knows what it is that he/she wants, so that the customer is in the best position to see that's what's being delivered.

DR. JAMES F. LEATHRUM (CLEMSON UNIVERSITY): The question of appropriate design discipline is something I wanted to comment on a minute ago, and this is another opening to add a comment. I have had some concern about the top-down stepwise refinement sort of discipline applied rigidly because it fails to identify the commonality that one might achieve in designs, and thus limit the number of modules maintained in which, that commonality can be identified. A different light on that subject came to my attention when I was discussing this conference with a colleague and I asked him how he designed for the purpose of testability or with testing in mind. He said he purposely tried to combine modules that were designed such that they were used for multiple purposes. This seems to be the opposite of the usual notion of cohesion for achieving a single purpose in a module. He pointed out that the ability to find the defect in a module may be a function of the amount of traffic through the module. I could use an analogy, if I wanted to test this carpet I would put it out there in the main hall, have multiple uses for the carpet in order to determine its servicability. I purposely gave credit for that thought to somebody else, but I didn't want to take the credit, but I won't name him, I don't want to blame him either.

MR. GEORGE NEEMAN (US ARMY MATERIEL DEVELOPMENT AND READINESS COMMAND): I guess this is for you, Lee. Your analogy of the screwdriver and hammer, and I take it that's your assessment of where we are right now, about how many years, or tens of years, or hundreds of years do you think it will take before we get to the full toolbox, where we know what we are doing and we know how to require a contractor or tell the contractor what we want and expect a quality product.

DR. OSTERWEIL: I think we are a little farther than just having a hammer and screwdriver. I think those are the ones we are familiar with. I think there are an awful lot of other things out there that are potentially very good, very useful tools. The question of when they get to the point of what I would call totally good, complete and accepted tools, there are basically two forces that have to come into play. First, there has to be an awful lot of investment into bringing these things to the point where they are handed to a lot of people who can evaluate them and decide how good they are. The other one works in the opposite direction. I think that if everybody in the panel here magically today got tools which we all magically agreed were perfect, fit together perfectly, and we all decided that that was great, by the time these tools impacted the whole world and were finally accepted by the people all the way in the back of the room, I think many years would pass. I believe that in the software engineering community, good ideas eventually catch on. They have to be developed. There has to be capital put into the development of them, but once that takes place and they are put out there, there's a grapevine phenomenon. People pass the word from one to the other to the other, and eventually something catches on because it is good. I think UNIX is an example of that. It wasn't forced on the world, but a lot of influential people decided that it was a really helpful thing, and they got their work done better because of it, and the word spread. After a while, everybody was using it. I think that is an indication of how long it takes a good idea, once it's been adequately capitalized and put out there for people to use. It really catches on. But, I would say that with enough push and enough capital investment, we are still looking at on the order of several years, maybe a decade or two, before we finally get to where it is we have to get with tools.

DR. STUCKI: Let me just make a couple of quick comments. First of all, I don't think there is such a thing as a perfect toolset that you can just slap on any RFP or whatever you want to do. Look in the carpentry profession, I know a little about woodworking. When I want to build or make a certain piece of wood or perform a certain project, I have radial arm saws, I have tables saws, I have mitre box saws, I have hand saws. You get three or four carpenters in here and tell them to work on the same project, they may not use the same tools. So, I'm not sure we are ever going to use exactly the same toolset, and I'm not sure there is such a thing as a perfect toolset that we are really striving toward. That doesn't mean we can't invest in some other kind of screwy saw that's better than all of them sometime in the future anyway.

The other thing is, I think with Lee, that there are a lot of good tools out there. At least there are a lot of good tool ideas out there, and for various reasons, maybe because of the resources required to use them initially when the hardware was expensive, when the tools first came out, there are a lot of reasons for why some of the tools weren't used in their current packaging. It doesn't mean you can't take some of the good ideas they had with slightly different packaging, maybe put some of them on micros, maybe put a better user interface on them, maybe do a little bit of juggling around. It really has not too much to do with the basic depth of the tool. I think there is a lot of technology out there, in fact, most of the new tool systems you see, including a lot of the things we are trying to do, they're not really grandiose new inventions. They are really better packaging on many existing ideas, and novel ways of combining existing ideas that other people have talked about before.

MR. NEEMAN: A follow-up question on that. In the meantime, how does the government try to require the contractor to use a particular tool? For example, if I hear from word of mouth, that the Boeing tool is just the greatest thing since apple pie, and I want to put it in my competitive contract or even a sole source contract for some other contractor. How can the government try to overcome the reluctance of first, the government itself, the program manager saying, this tool is going to slow down the program, require more cost, schedule, etc., and second of the winning contractor saying that we are not experienced with this tool, we're reluctant to use it, therefore, we are going to price it high and hope that you don't accept it. Maybe that's just a fact of life at this point, but have any of you found a way around that or a way that could help the government require some of those?

DR. STUCKI: I have mixed feelings on commenting on this. A couple of years ago, the government wanted to encourage contractors to use more "modern programming practices", such as structured programming, etc. When the RFP's went out with those terms in them, there weren't any respondees to those RFP's that I know of that didn't agree to do that. They may have interpreted some of the phrases differently, which they did very often, but, they did agree to do them. I think the customer in some sense is in the driver's seat, the customer is always right. There is a little bit of that philosophy that if the customer is willing to wield the big axe, and the big staff, he can do it.

DR. OSTERWEIL: I think in a very real sense, it's kind of a non-problem. The best comparison I can make is the higher level languages. There was, at one time, a lively argument about whether you should code operating systems, for example, in machine code or whether you should use high level languages. You really don't hear that controversy anymore. Most systems software is written in higher level languages. It is simply because people have found that it is more cost effective to do that. I don't believe I ever saw anyone prove it. I don't believe I ever saw anybody with a definitive set of statistics that showed it was absolutely, positively it was more effective to write system software in high level langauges. It's just simply that people went out and did it and discovered that was a good thing to do and became comfortable with it and they just simply did it that way.

I think the same thing holds with what you're trying to talk about. I don't believe any customers ram any use of tools down any contractor's throat. The customer that tries that on a contractor who can't use the tools effectively is going to regret it. Eventually, contractors will come to the point where they have tools they are comfortable with and they're effective in employing. They will bid based on their sure confidence in their use of tools, and they will be able to produce software more effectively, and it is simply going to happen. I don't believe it is useful to try to ram it down anybody's throat. I don't believe that it ever will be adjudicated in a meeting like this. People just finally get the work done better that way, and that is the end of that.

MR. NEEMAN: If I can just say one more comment. Then what you are really saying is the government ought to continue the way they have for the past number of years in putting the general phrases in the RFP that say: "please respond in a manner that would be technically acceptable", or "use a methodology that would gain these overall concepts", rather than require specific tools, even though, up till now, we realize that we've done that, and we're still getting poor quality products?

DR. STUCKI: I'll address his question. It seems to me that the prudent thing you could do is say we would request the prospective contractor to propose for us to demonstrate in his response to this RFP the methodology, that he would use and the tools that he would use. In particular, we would like him to address issues. You could make some suggestions, like will he include analysis of the specifications, all of PSL or something equivalent to that. You could make some sort of suggestions that would encourage them to either think seriously about using the tool or come up with a good reason why they've got a better way to do it without the tool. Maybe they can come up with a better tool, who's to stop them from doing that? You don't really want to discourage them. In fact, the last thing that I'd like to see in a RFP is something that said thou shalt do it with PSL. I'd like to see instead something that says thou shalt do it with a formal methodology, such as PSL or something equally as good, where the burden is put on you to at least look at the thing and take some kind of a stance pro or con, and justify why you are doing it.

MR. SAN ANTONIO: It does to some extent relate to the point on MIL-STD 1679 because you were dealing with standards, but you posed a couple of interesting questions. First of all, if you do not own something, if something is proprietary, it belongs to a single contractor, then you obviously can't go out in a specification. If you truly believe that it is something that you want, then you're going to have to invest the monies up front to procure it and then to make it available to everyone. We've seen that. We see that now, with the development of Ada and the Ada Programming Support Environment. One program that immediately came to mind when you were talking about it was a program of the Air Force's Aeronautical Systems Division. They essentially capitalized contractors to use a certain tool to enhance and automate the manufacturing of aircraft. I don't know what the statistics are on that program in terms of what the actual dollar savings were, but I know there was a great deal of consideration to investment strategies and how they should invest in that technology and the expected benefits to be derived from that. Relating to the question on 1679, it is very difficult to write all of the criteria in the standard when you go out and prescribe a general requirement to be satisfied, and then come back and have sufficient criteria to measure whether or not someone is adequately going to respond to that. We are still trying to do it though. As new technology comes along, sometimes that is your only alternative, because you don't own the tools but you want to see something like one of those tools used on your program. It's a hard nut to crack and your real control mechanism then is in the evaluation of the contractor's proposal, but then you're betting on the fact that the way you weight the contractor on the specific item will determine the outcome of the contract award. In the case of software, that typically doesn't happen.

DR. DEMILLO: Leon Stucki mentioned that when people were talking about structured programming, trying to decide what it was and who should use it, and the government was thinking about requiring structured programming in RFP's, there was a great deal of interest in automatic structuring of programs. This sort of gave you the idea that there were people out there who were thinking: "If this is a requirement that we have to satisfy, we'll satisfy it, but you won't get the product that you want out of it." That is a the danger of requiring a tool, or requiring something specific, of that nature, in an RFP. In doing some of the data gathering for STEP, we came across people who told us: "Sure, you can write a requirement like that into a contract, and we will do our best to get around it, to formally satisfy the requirement, but to get around it, if we don't like it." Any other questions?

MR. MICHAEL MERRITT (BELL LABS): My question is addressed to two of the panelists, Ralph San Antonio and Vic Basili. I detected what I saw as a contradiction in two of your slides. I believe I'm quoting from Ralph's slide that, subjective evaluation undermines control. Then there was Victor's discussion about the value of reading requirements documents and code, seemingly a proponent of, such subjective evaluation techniques. I was wondering if I have, in fact, identified a disagreement? Do you have a comment?

DR. BASILI: Looking for errors in a requirements document isn't a subjective evaluation of anything. It's a start, and certainly is: "I have found an error or I haven't". It's a fact, and I can demonstrate that there is an inconsistency or ambiguity or something of that kind. I don't know if that's the subjective issue, but let me go on and say I happen to believe that some subjective data isn't bad.

MR. SAN ANTONIO: I agree with you that subjective data is of value, but the point there was, if you're faced with telling someone that they didn't pass the exam, that they've just failed, it becomes a very sticky situation as to exactly what is required and whether or not they met the objective or met the requirement at that particular point in a contract. I am speaking very specifically about acquisition programs where you are going back to a contractor and saying I'm rejecting this data item because it doesn't contain a certain prescribed set of information or you did not satisfactorily complete this milestone. At that point in time, you have to be very definitive, and my point is that if you can be quantitative you might be convincing. But, if you are not quantitative, and it's just my gut feel or I didn't get a warm feeling from the presentation, then you have really lost a great deal of control.

DR. BASILI: Let me come back to the subjective data. There are a lot of uses for that. For example, it is very hard to give an objective evaluation of something like the use of a methodology. I might want to quantify that and give you a rating of 1-5, and then have to justify why I gave you a 3, if you didn't use in the right manner and here are the reasons. Those are things you would like to know about, and we just don't have any quantification mechanisms.

DR. STUCKI: Let me put one plug in . For those who have heard various of us talk about various tools and so forth, there is something coming up in the immediate area here in the summer that a few of you might be interested in, the "Soft Fair". It's going to be a software engineering tools fair in Crystal City in the end of July. You might want to look around for that. That'll be showing the state of the art tools, many of which are not publically available, from research labs and universities, from companies and so forth, but it's a technology interchange to actually show you the what the state of the art of tools is. For the person who asked Lee how long it might be, you might judge for yourself after you've partaken something, an activity such as that. It's sponsored by IEEE, ACM, National Bureau of Standards, Ada Joint Program Committee, IFIP, and a couple of others.

MS. R. J. MARTIN (CONTROL DATA): First of all, I would like to ask Ray Rubey about the economics of testing theory, that he discussed this morning. I would like to know how useful it is in practice. Do you see the main use of the theory to be the conclusions you draw from it or has anyone actually tried to go through the calculations? If so, how detailed are the calculations, do you calculate dollars only, how do you calculate loss of life, and so on?

MR. RAYMOND J. RUBEY (SOFTECH): I guess the easiest thing is to convert everything into dollars. We don't get so gruesome as to calculate loss of life as far as lost function costs. I see at least two purposes to the model. One is you suggest doing some modeling, you like to try to put some numbers in and see how much this error costs, how much this application's lost function cost might be and how much testing then might be justified. Really, that's more of a research type of thing. The most practical thing is to start to define some terms so we can start to talk, in particular with management, the developers, the customers, about what things are important, and why you should test them out. One problem we have always had in testing is we talk about different things. Part of the purpose is to define some terms and ideas.

MS. MARTIN: Has anyone tried to actually do the cost functions?

MR. RUBEY: I'm doing some of it. I don't really have any results yet. Just constructing models, what if this, then what about this, what is the testing process amount in man-months, translated into dollars what kind of errors would we have to get out for that amount of testing.

MS. MARTIN: One thing I would like to note has to do with something Marilyn Stewart was talking about. The terms and definitions for MIL-STD-SDS are currently being reviewed by EIA. Someone from EIA contacted me last week relative to that review. So, if anyone here would like to be involved in the review process of the terms and definitions, feel free to contact me. I can tell you who to contact at EIA.

MR. JAMES HEIL (ITT): One of the points made in one of the talks was the advantage over the life cycle of software providing some degree of flexibility. I guess in the last 18 months or so, in industry, we've noticed the use of reprogrammability concepts. For example, where certain system parameters, so-called user data bases, etc., and in some cases, even software templates are incorporated into reprogrammable devices such as EA ROMS, etc., with the implications that in operational use, somehow, based on changes in the environment or new algorithms, whatever, the operational user can actually modify the software and ergo, the system performance. Now, this has some rather interesting advantages in capability, but it is also frightening from a testing point of view. This is obviously an element of injection of, in some sense, new technology and obviously impacts software testing in the broad sense. I wonder if anybody would like to comment on the opportunities and threats involved in that emerging process.

MR. SAM DINITTO (RADC): I guess I didn't realize they did that much. As far as the users being able to reprogram, we feel there is a certain class of problems that will allow this sort of thing. Obviously, the testing is of great concern. Obviously, we don't want to give that power to the guy out in the field who is just supposed to be checking the connections. I think before the Air Force, or the DoD, takes out something like that, obviously the testing technology and guidelines are going to have to be concrete. I would hope that any modification like that would be a fielded system modification, not just an isolated case. Although, I will say, in the EW area, we've come a long way in that the people out there maintaining those systems, maintaining their software, having to change it because of a different threat scenario, have done a pretty good job. They can turn it around in a few hours now, based on modifying some tables. Those people know what they are doing. I hope that the DoD will not be so foolish as to give everybody that capability.

MR. RUBEY: Responding to the same question, I hope with an optimistic viewpoint. Based on some of the experiences of templates, they just add a little more complexity to testing. Not an enormous amount more. But one of the ways to consider it is if you're changing the value of a parameter in there, it's just another input that can vary, as if the user is supplying the input or, the environment in which the software was used was supplying an input. So, if you know that in advance when it is going to be tested, a parameter or a set of parameters can be varied, and you make certain that during testing that you test the limits of this variation as if it were a normal program input variable. In some systems it becomes advantageous to build in the self checking as part of the system so as you use the data or you exercise the system, there is code in there to re-run the test or do data validity checks on the system. This in itself is just like a built-in test on the software.

MS. CARAL GIAMMO: I have a comment on the question you had. It doesn't matter how careful you are in that deployment when someone messes up and the last exercise is to send out a software correction by a message, by a number of other means, and send out a validation package to one of the major sites, they install it, the validation package works the first day of the exercise, the proper software is operating, then the system crashes, and one of the operators on one of the night shifts reloads one of the old versions of the software. We went five days into an exercise, and that system crashed, because they didn't have the proper latest correction to the software. So, I don't know how you control it. My question, I don't think the people on the panel are the proper persons to answer the question, but I'll ask it anyway. They have all brought up a very similar problem which has to do with the fact that, in the life cycle, one of the reasons the government is in so much trouble is that we don't have enough people to manage the whole cycle and properly monitor what is going on during the acquisition cycle. To solve this, we have gone out and developed things like IV&V contractors, where you are going out and buying an IV&V contractor because you can't do it yourself in-house. Now, here you've got a $10 million project, you've got a 5 person office, one of whom is the secretary. You're going out now and trying to sell the idea that you need to spend another $5 million or another $2 million for an IV&V contractor, and that you need 2 more people to monitor the work of the IV&V contractor to make sure they are really doing the work. How do you do it? How does DoD start to, or what's going on within DoD to try to, solve the people problems that I hear people talking about?

CAPTAIN WILLIAM P. NELSON (USAF ELECTRONICS SYSTEMS DIVISION): Being a foolish young captain, I'll answer part of that. It turns out that we do have that kind of concern in Electronics Systems Division, and I'm sure the other product divisions do. In fact, the presentation I gave on software quality assurance is one of our answers to that solution. There is a section in "52779A" that says the contractor is responsible for only delivering software that works. We believe that. So, we feel it's the government's job to monitor the contractor, using his own software quality assurance organization. That's one of the reasons we are trying to push it and why we are going out into contractor plants doing software quality assurance reviews. We don't advertise it that widely, but as a short term solution, that's one of the few things we can do. We can't get more people. We don't have any control over that at our level, but we can try to get the contactor to do the right things so we can watch him. Hopefully, by watching the software quality assurance organization, it limits the amount of effort we have to spend. The other thing we can try to do is that we're trying to get the program offices more effectively used, things like Defense Contract Administration Services and the Air Force Contract Management Division. Those people are not in program offices, but they're the people in DoD with the in-plant responsibility for monitoring the contractor. They should be doing more to keep us out of trouble. That's basically the kind of solutions we've been going after in ESD.

MR. DINITTO: Within the Department of Defense, and within the Air Force, we have an effort right now to establish a speciality code for embedded computer systems,. I know the Navy, for all of DoD, is undertaking a study to establish a special series on the civilian side called software engineers. The problem again that we have is that we cannot compete dollar-wise. What we do hope to do is to come up with some other training programs. In fact, in a meeting we had a couple of weeks ago within the Air Force where we were looking at the initiative, looking at those high-task areas, a very good point was brought up. People said, "why do we say we want a software engineer?" It's just like a software problem. One problem. Why don't we start thinking about some special specialty codes, dealing specifically with testing, dealing with some of the project management, dealing with some of those other aspects of the life cycle or different facets of software, rather than saying any software person can fit in anywhere. The initiative, as I said, has a heavy emphasis on the human resources area, in the training period. We see the problem not just within the Department of Defense, we see it an awful lot in industry, the lack of qualified people. At the university we don't have to speak to that, industry is hiring them away too.

DR. DOUGLAS GIESE (TRW): I don't think you can really get to the end of the system and then test. I think you have to follow this stuff all the way through. The first time you start writing the requirements, is it testable, how can I test it? As you develop the system, just to make sure that you are implementing what you're supposed to be implementing, verifying it as you descend down from the requirements to the specs to the design, that the process is consistent and each level implements the level above, and you have to do simulations or analysis to show that you're following the path. You can't just shoot the arrow and hope it hits the target and go down at the end and measure what your error was and try to feedback. You've got to follow it's path all the way to the target. I think that is what the contractors can do.

MS. GIAMMO: I believe in all that stuff. My question is how do I convince Congress to give me five more people to do that?

DR. GIESE: That's not my problem.

MS. MARILYN STEWART (BOOZ-ALLEN AND HAMILTON, INC.): About the only thing I can point you to is a study done by RADC on the cost effectiveness of IV&V. The bottom line of that study was that if you started early, IV&V would pay for itself in life cycle cost savings in errors not committed, not built into the software. I can refer you to that document, which is entitled "Analysis of IV&V Data" and is available through DACS and standard DoD documentation services. I could refer you to that. There is no simple answer other than to start planning for it as early as you can. If you've got your funding profile already defined and find out you're in trouble, and now you need $5 million extra for IV&V, you're going to have a lot harder time justifying it than if you put it into the POM, or whatever your original funding documents are, at the very outset of your program. All I can say is, just program for it early.

LT. COL. MICHAEL A. BLACKLEDGE (USAF TEST AND EVALUATION CENTER): There is half the answer on the budget part, the other half on the people part. What we've advocated in the Air Force is to use a hybrid team for IV&V anyway. You get some of those people that are going to end up maintaining that software, and they are part of your IV&V team, along with the contractor. If you can get enough of them, perhaps you don't need a contractor if they're well enough qualified. But, you line up some of those people, and you don't need to take your own slots if you can take them out of the maintaining people. That way, they also get trained on the software they are going to be taking over anyway.

MR. JAMES HESS (US ARMY MATERIEL DEVELOPMENT AND READINESS COMMAND): Let me address for a minute a couple of the steps we are taking in the area of training. I can't help you get the people either. One of the things DoD has done in the past several years is to implement a course at Defense Systems Management College, entitled Management of Software Acquisition, a very good course, a short course to bring some people in. Unfortunately, or maybe fortunately, people in the services and industry recognize the value of the course. The last time I talked with Ken Nidiffer, he was booked up for the next two years. So, we are taking steps. People recognize the need and are trying to avail themselves of it.

MR. DENNIS GACKE (SPERRY UNIVAC): There is a lot of movement afoot in the defense to consider firmware as software, and software as software, i.e, treating it alike. Therefore, test requirements apply equally to firmware as well as all kinds of software. That's the difference between the different types of software as well as firmware. The thing that we get into in the development of a computer itself that consists of emulation firmware, diagnostic firmware, operator panel firmware, power tolerance all that kind of stuff, each one of those is just a little bit different, and has to be, in my mind, treated differently and tested differently. There doesn't seem to be too much recognition of the fact that there are different types of software in different departments. That's the first question to be addressed. Secondly, we have gotten involved in the independent in-house testing, down at the unit code level. The unit code was arbitrarily set at 150 lines of micro code. There is expressed a lot of difficulty in doing the testing of that, from the standpoint of how do you verify that it meets design requirements, how does it meet its performance requirements. Most of the testing, as it turns out, is more of a structural nature where you just verify that the code executes without error. Maybe somebody has some insight there.

MS. STEWART: Well, let me say this. Unit testing is usually conducted as structural testing. It usually is at that level, i.e., testing to design. So, that is the normal intent. Unless you've done something special with your methodology to build requirements traceability into the process and back-up at the unit testing level to requirements, that's about how you would expect it to come out. You would expect unit testing to accomplish exactly what it is accomplishing. On the firmware question, I think all we can say is, within the DoD at large, that's still out for study. It's been a known problem for some time. The existing policy drafts just treat firmware as software, unless a waiver is granted. The kinds of consideration that you're raising are granted on a program by program basis. So, we don't have a simple solution to that problem at this point.

CAPT. NELSON: On the software versus firmware, maybe I shouldn't admit to it, but there is a white paper written by Dr. Sylvester at ASD that talks about hardware intensive versus software intensive firmware, and the fact that you should treat them differently. Most of us tried to kill that paper, and we effectively did so. I know for a fact that we just reviewed a RFP at ESD, where we allowed the term software versus hardware intensive firmware, because the program office did an excellent job of defining them, did a very good job of lining out the types of requirements they wanted, and they addressed some of the concerns you have, the fact that there are different ways to test it, and it has different maintainability requirements and such. So, that is kind of echoing there that when you have a program by program basis, such things are allowed to happen, if it's reasonable and well-defined. There was a Monterey conference, the Joint Logistics Command Workshop, Monterey conference, and Panel B, specifically addressed that whole issue. There again, the thing's filed in DTIC, so if you just use Joint Logistics Command, you can run it down from there.

MR. RUBEY: Let me address the second part of your question and share some experiences. Very often, people say we are having trouble testing or we're having trouble defining some reasonable tests. If you question them a little bit, you'll find out the reason they're having trouble in testing is because they really don't know what the requirements are. This particularly becomes significant at the module or unit level or some low level, because the module's been defined without laying out what the requirements are. I suspect that nobody knows what the requirements are that they are supposed to test that the unit or module satisfies. Once you know what the requirements are that the unit is supposed to do, then it becomes very easy to do something other than structural tests. When you do not know what the requirements are, then you test the structure and elevate the question to a higher level.

QUESTIONER: I want to comment on testing the modules against the requirements. A lot of times you'll have several modules that go together to satisfy one requirement, so that makes it rather difficult. I think that most of the trouble comes from very ill-defined high level requirements. From our experience, we've experimented with automated unit test tools and have found them to be extremely successful on scientific types of code where you have a very specific input and a very specific output that is governed by some type of equation, that works very well. But, for other applications, we've pretty much had to tailor each test to each specific application if at all possible.

MS. MARTIN: I would like to expand on the question that was just asked in reference to firmware versus software. I think that another way to state that question is, if you just think about software, are all errors created equal? The testing requirements or standards don't really seem to make a distinction between how testing should be done in terms of the criticality of failure. Is any work being done in those areas?

MS. STEWART: A standard V&V approach is to do criticality analysis of all of the software requirements, and that's based on the premise that all requirements and all errors are not created equal. Obviously, when you are dealing with mission critical weapon systems, this just tends to fall out. The kinds of functions that do mission data recording for post-mission analysis are not equal to those that protect the guy in the airplane from being fired on. It's just a fairly simple fall-out of the system mission. The work that's being done in this area is to base testing on a criticality analysis, which sorts out which errors are more important than which other kinds of errors, and testing against requirements, using that as the foundation.

LT. COL. BLACKLEDGE: MIL-STD-1679 gives a definition of different severity classes of errors, and even goes so far as to say what the passing criteria are. You can have zero of class one of the most severe errors and so on. So, there is something, I don't know what SDS has in it.

MS. MARTIN: In MIL-STD-1679, when it describes the testing that must be done for a system, such as module testing, subsystem testing, and so on, is there anything in there that says you don't have to be quite so thorough here if it's a non-critical module?

LT. COL. BLACKLEDGE: I don't think it defines modules as critical. It just goes through the errors. It does say you can have a lot more non-critical errors than you can have critical errors. It makes a differentiation that way. It does not go into a module characterization.

MR. STEVE HABER (SANDERS ASSOCIATES): The last two days we have heard a lot about life cycle costs and development. We've heard it from a variety of perspectives and the panel members presently in front of us represent a very good cross-section, in my opinion, of different viewpoints of the life cycle. Now, I would like to know, based upon the various opinions and experiences and viewpoints, are we moving towards really implementing life cycle costs or are we still in development and acquisition perspectives regarding the acquisition life cycle costs? We talked about having to do more up front, but I still see schedule slips, squeezes of schedules, everyone talked about various squeezes of schedules on the unrealistic approach. I wonder if anyone would like to comment on is the trend really changing or is it just verbiage?

MR. RUBEY: What I see in life cycle costs... I think the DoD community is very serious about life cycle costs. I think that's the selling point on many approaches, many systems, and behind their emphasis in pushing the higher level language. I think where there is not much interest, not much emphasis in life cycle costs, is with the contractors and developers, and they're only going to do very many things in life cycle costs reductions as they get pushed by the DoD. The contractor doesn't care once he's got it accepted. His part of the life cycle is over. Right? That characteristic will never change. It will be the DoD that pushes life cycle costs improvement.

COL. HAL FALK (WRIGHT PATTERSON AFB): Although perhaps part of my concerns have been allied to some of the comments that have been made, I would like to read this to IV&V. I understand there has been an analysis on the usefulness of IV&V and perhaps that was based on how we developed software in the past or on systems that were developed in the past. We used the approaches that have occurred in the past, and I would hope that we have learned to better manage and develop software, and we've heard some of those ideas expressed here today. IV&V costs a lot and, perhaps I'll be shot by most of the people in here if I talk it down, but from a cost standpoint, I would like to talk about other approaches. Are there other alternatives, using, enforcing good development practices, tools, methodology, quality assurance that we've heard about today, and is there some approach, except in perhaps the most critical software, where we don't have to have IV&V but can use a single contractor to provide us good and error free software?

LT. COL. BLACKLEDGE: Before I let Marilyn answer that. The ideal situation, the ideal solution is to have the people in the program management office experienced enough in software so that they can monitor the contractor, so that they are their own IV&V team. As you saw by that 13%, nobody raised their hand, that's not found. You don't find that kind of software expertise in a program office. Why? Because they're in short demand right now, they're spread out otherwise. Failing that, the next best thing is to draw in expertise, as I mentioned on Caral Giammo's question, to draw in expertise from the people that are going to have to maintain that software. That would be, if not as good, at least it would be something that would give you some training for those people. They may not be IV&V experts, but they're going to be the type of people who are going to have that software dumped on them someday. Those would be cost effective, if you could do that. Failing that, you go to the IV&V contractor.

MS. STEWART: Well, I guess as spokesperson for IV&V, let me say this. There have been a lot of statistical studies on where the costly errors are in software and where does IV&V detect errors? The kind of things you are talking about, with automated aids, can attack errors in the coding process. They can reduce the amount of errors that are made in the first place. However, the bad news is that the most expensive errors and the most difficult to detect are the conceptual errors, errors in the requirements definition in the first place, and in the design, the development of an architecture that implements that. We have a long way to go before our technology makes that process more error proof. We have some techniques that are good for adding rigor to the process, but we aren't going to see errors of those kind, conceptual errors, not introduced for a long time.

COL. FALK: Just one thing to add. Is there anyone here in the room or on the panel that can cite an example of a fairly significant software development that was accomplished successfully without IV&V?

MS. STEWART: Not me.

COL. FALK: Is there a Boeing representative in the crowd?

CAPT. NELSON: I can't swear this is true, but what is claimed by Electronics Systems Division is that the PAVE PAWS effort was a very successful effort that did not have an IV&V contractor. It has been sold as one of the most successful acquisitions at ESD. I want to make one comment on the IV&V question in general. There are other alternatives. In fact, we see them. At ESD, some programs go out and they basically hire two contractors up through PDR or up through CDR, and they try to get the bugs out that way rather than hire an IV&V contractor. Another option is to hire an engineering management support contractor, which is what a lot of people really need, and call it IV&V anyway. The key there is that those all cost money too. The whole point is you're short people, which is probably why you went to an IV&V contractor, because you couldn't do it yourself. No matter how you solve that problem, it's going to cost money. Just pick the best solution, and it might be IV&V.

DR. GIESE: I'm not sure how much IV&V was done on the site defense program, that TRW delivered. I know we had set up quite a bit of internal controls within the company. Just speaking from that, I know we had fairly exhaustive control boards and as we got closer to the final delivery, any changes had to go through a number of different review processes, internal to the company, which is basically our own internal QA. That was a fairly large project that was delivered on time.

COL. FALK: I just wanted to add a few more comments on V&V. One of the experiences on the PAVE PAWS system was that ESD spent an awful lot of time at the various reviews, going over the software. A very large percentage of the time was spent specifically on the software. So, they really wrung out a lot of the software problems, and I know the sizes. For example, the tactical software C5 was about 2,000 pages. It was wrung through thoroughly, and similiarly, I think, the B5 was approaching 800 pages just for that tactical software CPCI. It was a very, very intensive thorough up-front effort, and certainly by most standards, was a very successful program. On a slightly different note, relative to the V&V effort. It would be an interesting question if you assumed that all of the review points, including the TRR, are really complied with, plus the spirit of the software QA program is enforced where each contractor really has a very thorough software QA program. All of these things together suggest, if an internal software QA effort may be run 6-8%, and you have to ask them, is it cost effective. In addition to that, if there is some assurance that there is a very good internal procedure, what is the incremental benefit of having a complete V&V effort, and would it be worth the incremental cost? I'm sure there are two schools of thought on that subject, so I will get out of range quickly.

MS. STEWART: The main thrust of IV&V is towards detecting technical errors, whereas the main thrust of SQA is typically to see that good development procedures are set up in the first place and followed, that kind of thing. Now, if those good development procedures do get set up, they certain enhance IV&V, but they typically do not include the detailed engineering assessments that fall out of IV&V. So, that's the question you've got to ask yourself. That's got to come from somewhere. It sounds like on PAVE PAWS, it came out of the project office itself. In other areas where the project office doesn't have that kind of manpower, they turn to an independent contractor, or they may go to life cycle support agent. There are many ways of doing it, but you've got to have that kind of engineering assessment.

MR. RUBEY: One comment, reflecting here, is, has anybody thought of a major project that was successful without IV&V, and maybe we are able to come up with one. Can anyone think of a project that failed without IV&V? We probably don't have to dredge our minds very deep to come up with projects that failed. I am an advocate of IV&V. I don't think it's a magic procedure, by any means. It could be done by lots of people, including the contractors themselves, if they established an independent group and staffed it properly with the same kind of people. The problem with the development contractor doing it is that he puts all of his good people on the development phase, and weakens the IV&V end. The advantage of getting somebody independent in there, whether it's an independent contractor, independent government agency, which probably the government agency is the best, is that agency will put their best people on it. They won't put their worst, that's their job and their responsibility. Since I've been in IV&V, I've challenged my friends in development. You people have a golden opportunity, you can put us out of business doing IV&V if you don't leave anything around for us to find. If your product is perfect, then the next time around the customer, the Air Force or Navy or Army won't need any IV&V. Sometime, probably when we do get much better in development, IV&V will disappear. It's a crutch we're carrying with us because we're just barely learning to walk.

CAPT. NELSON: I've got to make one comment on the SQA/IV&V question. That question gets asked about every other time I give the briefing to a program manager, because I tell them 5-10% of the development costs for software quality assurance, and 10-40% for IV&V, and then we peel them off the roof. I tend to agree with the answer Marilyn gave because that is basically the program we take. They are, IV&V and SQA are, basically complimentary processes. IV&V is concerned with the technical product we feel, and SQA with the development process. So they complement each other. The other thing is, I won't mention the system, but on one of the reviews we just finished after I wrote the paper, not only has there already been one IV&V contractor, they're hiring a phase two IV&V contractor. We just did a SQA review and the program manager wants us to go back again and do another one. The program manager has decided software is his critical element, and he wants SQA and he wants IV&V, and he's going to pay for them both. He's going to pay us money to go visit the contractor to make sure everything is happening. So, it's a question of where your risks are and whether or not you really think it's worth spending the money. Some program managers think it is.

MR. RUBEY: One final comment. The easiest person to market IV&V itself is somebody who has been on another program that has used IV&V. The hardest person to market IV&V is somebody who hasn't used it before. So, I think that sets sort of a selling point, we have a lot of repeat customers.

MS. GIAMMO: Marilyn talked about the study at RADC, which I know about, but no one is mentioning the study at the software engineering lab over at NASA-Goddard, where they used IV&V, and not only was it costly, slowed down the project, the software came in at worse quality than other projects where they didn't use IV&V. That one is available. I'm sorry that Vic Basili wasn't here to talk about that, but there's another side to IV&V. NASA-Goddard has a software engineering lab, and the head of that is Price McGary, and Vic Basili at the University of Maryland are members of the consortium. They, in computer science and software engineering labs , did a study on some space flight software at Goddard.

CAPT. NELSON: I just want to make a comment on that. IV&V is just another contract. If it's not managed properly, it will fail. There are cases of IV&V contracts that failed. So, there's a question of, just because you're going to do IV&V, you still have the problem of selecting a qualified contractor and having the people to manage it. It's not a panacea, but it's a tool that can work. How can you tell if an IV&V contractor is doing his job well? The same way you tell if a prime contractor is doing the job well. You've got to monitor him, you have to look at his product, and you have to evaluate him just as you do the prime.

MR. RUBEY: From a lot of experience in IV&V, I would say, if he doesn't come up with any mistakes, he's not doing a very good job. You're not taking much of a risk if you criticize him for not finding many mistakes. One refinement on that, what I usually say is one of the best ways to evaluate an IV&V contractor is what I call the "cry wolf" ratio. The IV&V contractor will be giving you reports and reporting errors. If 95% of the time he says this is wrong, and it turns out that the development contractor scratches his head and murmurs and cries, and says "I'll fix it", OK. But, if half the time, he says this is wrong, the development contractor says "wait a minute, this turkey doesn't understand what he's doing, then you've got a pretty bad IV&V contractor. You have to have a high percentage of hits, and he should be firing off the guns fairly often too.

QUESTIONER: Next question, back to the resources. Because of the limited resources in government, do you government representatives, feel, since we have much more today than yesterday, is the government going to more specific requirements now on RFPs, and if so, do we have the resources in the government to evaluate deliverables under that, or are we going more towards what the industry people recommended yesterday that we put in the RFP to use a tool like this one, or one as good as that one, and then don't we take more resources to evaluate a deliverable under that type system?

CAPT. NELSON: I guess I'll take a shot at that. One of things I usually try to avoid is source selection. I think we are going to more specific requirements in RFP's, because we're getting to a point where we know what the requirements are. The key question is where they fit in the RFP, they may be there by reference. There may a reference to MIL-STD-SDS, whatever it ends up being. That's still a requirement in the RFP. Right now, I mentioned earlier today that we have a 7 page paragraph that we put into the A spec, we also have about an 11 page statement of task path for software development that we try to get into every contract with software development. We put very specific requirements in there, as specific as we can get, all based on mistakes we have made in the past. I feel that MIL-STD-SDS and some of these other things, as they evolve, may not be part of the basic RFP package, but they will be there by reference, and it will be more specific because we've learned a little bit along the way.

QUESTIONER: But do you have the personnel to allow you to evaluate that 7 page statement of work? Or whatever gets delivered based on that 7 page statement of work? You've told me basically that you have a small office, and it takes you between 1 and 2 months to do a snapshot review of a program. Obviously, the Air Force has more than 6 programs. Even if you only do it on a yearly basis, you're telling me that some of them want you to come back next year. So, you're saying that you're looking at 5 or 6 or maybe even 10 programs, and the rest you're not really worrying about.

CAPT. NELSON: I think the point that should be made is that the technical staff aren't the only people that have computer written on them at ESD. There's approximately currently 150 computer resources which are distributed to program offices based on a workload forecasting model, that says here's how much work computer resources are going to do. It says this program needs three, this program needs four. I think this number may not be accurate, you never heard me say it, but I think the current manning level is around 70%. Everybody gets to be equally undermanned. That's true Air Force-wide. There are efforts going on way above my level to try to alleviate that problem. We're doing the best we can in the meantime. If a program has software, there are people who carry the label software assigned to that program to the maximum capability of ESD. We do try to have the people there to evaluate those issues, plus in theory they can come to the technical staff and ask us for assistance.

QUESTIONER: Another follow-up for the Captain. You mentioned DCAS, and you're trying to use them as much as possible. There are many people that believe DCAS is having enough trouble trying to monitor hardware contracts, and that since they are generally GS 9's or 11's, and you're obviously having problems keeping software people that are 12's or 13's, what do you really expect from DCAS? Do you think they can do anything?

CAPT. NELSON: One of the points I make in the paper is that we found that DCAS and CMD have the same trouble getting software quality individuals that contractors in ESD has. It's a real problem. They are beginning to get people that are qualified. The key issue here is that a DCAS or AFPRO is only going to do what a program office asks them to do. If we don't task them to support software quality assurance, they won't support it. If we do task them and we keep repeating it, they'll eventually hire the people and train them. In fact, the Contract Management Division at Kirtland has a training program underway which we have been supporting with traveling instructors to train their people in software and software quality assurance. It's a problem. There's not enough qualification now, but is being addressed at many levels within DoD to try to alleviate this somewhat.

MR. HESS: Let me address the first part of your question. We talked about the standards part of what we require on our contracts, on the other hand, we are also looking at requiring specific tools. The principle tool that we are now looking at requiring is the Ada language itself, if you want to consider that a tool. I see that happening more as the Army and other Services get more experience with requiring specific tools to be applied to their projects.

MR. CARL FISHER (JLR, CORPORATION): Captain, first of all, I totally agree with your attitude about DCAS or AFPRO, I've not seen one yet that has those qualifications, but your awareness is gratifying. The real question I have is a human problem. It deals with the relationship that a project office gets with its contractor, and the strong need to succeed. You won't make General if you're on a project that doesn't work well, so you don't tell anybody about it. If the IV&V contractor works for you, you don't tell anybody about that, you work it so that the report is somehow sanitized. We already have a good many wickets that we have to go through. We have reliability audits, we have PDRs, CDRs, but after you've been in a program for a while, the group, as a whole, from project office, acquisition manager, down to subcontractor, gets to have a siege mentality of, "we have to protect ourselves from all of those guys out there who are trying to find out what's wrong." I would like the panel to comment on that problem, that I perceive to be very real. I don't know that it has a solution, but more wickets doesn't necessary solve that problem.

MR. RUBEY: Let me comment on the siege mentality, or, I don't want to hear any bad news. From an IV&V attitude, the IV&V contractor has to be providing problems and information to the Air Force or DoD agency. That's how they should be judged. They want the program to succeed, the project office wants this project to succeed, the developer wants it to succeed, and the IV&V agency wants it to succeed. But, the IV&V agency's contribution towards making the project succeed, and in some sense the project office's contribution toward making the project succeed, is by identifying problems very early in the process, as was mentioned before. Their job is to find errors early when there is time to do something about. The IV&V agency has failed, the project office has failed, if all the money has been spent and time is gone, then you say "hey, we've got a problem". What you have to do is find the problems early, and that's the way you make projects a success. I've never seen a project that didn't have lots of problems. The difference between a successful one and a failed project is how fast the problems are found and corrected. If you find them fast and correct them early, you can make the project a success. If you don't know them, if you hide your head in the sand, it will be a disaster. Eventually, any project office has to prove performance. The guy's at AFTEC are going to get their shot in.

MR. FISHER: That's true. I think what I'm hearing is if you have a good, well-managed project, it's going to work out alright. You'll go through all the wickets and ask all the right questions. But, we all know there are lot of projects for one reason or another that don't work out right, and they still go through all the wickets, one way or another. That's the concern. I'm not objecting to IV&V at all. It's what do we do about this other problem that does exist.

MR. RUBEY: I agree with you. You can get through all the wickets if the wickets are rubber stamped, mixing of metaphors. In fact, if I can go back to the previous question, I think one of the reasons the Air Force very often, or the DoD, needs high staffing in project offices is because they're doing much of the management that should be contractor's responsibility. The only reason they get involved in making the wickets high and hard to get through is because internally the contractor hasn't done very much to get through the wickets. Sometimes, the only reason any substance is put into a design review or test requirement review is at the insistence of the project office and the management within the development organization could care less about it. I think we need a lot of people in project offices that make those wickets high, so that getting over the wickets means something. If we are going to fall flat on our face, let's fall flat on our face at the early wickets.

QUESTIONER: I would like to make a few comments about IV&V. We do have large software development projects in my company that are successful without IV&V. We also have disasters on our hands that are very heavily involved in IV&V. We have incompetent software managers, and we have very competent software managers, and there's a direct correlation between which is which and when we use IV&V. With good managers who attract good engineers, and do a good job, the customer does not need IV&V. The customer knows very quickly when he has the other situation and begins talking in terms of IV&V. The point is, let's be honest about it. It has to do with people. We're dealing with people problems. One of the things I like about the software initiative is that it begins to deal with competency in software engineering, and how you develop software. Until we get that, we're going to need IV&V for the incompetently run project, and we don't need it for others. We're going to need to upgrade our profession until we are all comfortable that we know how to do this job and do it well, and provide the kind of products that do the job.

MR. RUBEY: Let me respond to that. I don't think we use IV&V or a high level of monitoring by a project office just when there is incompetence in the developing contractors. I believe there is at least one other factor, and that's a factor of how important the system is that is being developed to the government agency that's buying the system. For example, I've had experience in two systems, the Minuteman and the B1. Both of those have had a long, long history of IV&V, and I would not criticize in any way, shape or form the development group. Very well managed development group, good software development activity. But, both of those were very, very important systems. As was said in the beginning, they were not going to have a B1 sitting out there, flunk its flight test, not have a successful flight test, so we will take out this extra insurance policy on IV&V

although the software development was a super group. Same thing is true for Minuteman. It's a cheap insurance policy, given the cost for it in respect to the system. Get back to the paper. Think of the lost function cost. What good is all the billions spent on a Minuteman if the software doesn't work? If I have to spend a few hundred million validating it, that's probably a cheap price. Same thing on the B1. There's another factor besides the quality of the people doing it. How important is it that it works correctly?

QUESTIONER: Just two comments. First of all, I think it might be a very interesting element of research to identify several eminently successful projects that either did have IV&V or not, and try to determine what the anatomy, what was the underlying reason for the success of those projects either way, and try to plagarize the things that went right and use them in future projects. A second comment related to some of Captain Nelson's points. I think that the assessment of a contractor's past performance and his current plan as perhaps witnessed with a CPDP that comes along with a proposal; does he really know how to manage a software intensive project; what is his track record in the past? I think it would be a very useful thing to give more emphasis, so that you don't end up awarding a contract to the lowest bidder. If you take a less qualified bidder, and buy a cheap insurance policy with a V&V contract, I'm not sure if you really end up ahead in a global sense. One comment from the Colonel from ASD is that a project office also has to be very concerned with the quality of an IV&V contractor. Properly done, if he's doing his job, it can be an asset. I think some of the prime contractors here can also cite cases where an IV&V person has deflected key management resources from the real problems into addressing literally pounds of nitty-gritty, comma kind of points. So, I think the Colonel's point is very well taken that a good IV&V program is a real asset. One that is poorly managed can actually have a negative effect for the DoD projects.

CAPT. NELSON: For a while we weren't allowed to use past performance. I think we can do that again. Although we're not doing a lot in that area right now, I think it's worth noting that Mr. Phil Babel of ASD, who is the computer resources focal point there, has been working on a rather thick list of questions for pre-award surveys which would basically, hopefully give us a tool to do that. There was, in fact, a letter signed out by Lt. General Bond about a year ago that said you should try to do pre-award surveys for software, and maybe Steve can help me out on this. If I'm not mistaken, the new AFSC supplement to 800-14 that just got published also says you should think about doing pre-award surveys. So, we've got the policy, now all we need is that implementation part that always seems to be lacking.

MS. STEWART: I have one comment on V&V as an insurance policy. I can state from first hand experience that it is no fun, and not very effective, to take a poor contractor and try to keep them in line by applying a lot of V&V. It really is a management headache, both for the V&V'er and for the project manager. I don't even like to hear that suggested as a way out. An approach for picking a low price bidder, you're not really sure, but then you buy V&V as an insurance policy. It's going to be headaches all the way around. There's occasionally just more errors than you have the resources to detect. It's just really not cost effective.

MR. RUBEY: I agree 100%. I think the ground level that is urged on program offices is that IV&V can only tell you when there is a problem, they can't straighten the thing out. So, if it's an incompetent development contractor, all I can tell you is that you're on a sinking ship, so abandon ship. They can't go around and start repairing the ship. The development contractor is sailing the ship and keeping it off the rocks. In regard to your first comment, there isn't much literature, much discussion of the effectiveness of IV&V, a recent report was one of the few mentioned. It might be good sometime to have a panel discussion having a representative of, say, three projects, and for each one of those projects, have a person that had a prominent role in the IV&V, a prominent role in the development contractor, and a prominent role in the customer or program office and see how their IV&V worked, how their working arrangements were, whether it was effective. Very often, after an IV&V agency has worked with a development contractor for a while, a fairly appreciative bond develops both ways. The development contractor likes the IV&V contractor.

DR. PETER WEGNER (BROWN UNIVERSITY): I imagine that in the future, when we have to make a decision between bailing out sick projects by a lot of IV&V, or throwing it away and starting anew, you might increasingly go in favor of throwing it away. This goes back to something that Richard Lipton said yesterday about rapid prototyping. The whole idea that of the producing things that we should consider throwing away and starting over will be something that will be made easier by the new, better access to computing facilities and richer tools that we have.

MR. DONALD R. GREENLEE (OFFICE OF THE DIRECTOR, DEFENSE TEST AND EVALUATION): Tom Burley likes to tell the old one about the drunk who exceeded his normal standards of revelry one night, staggered to his car, drove home very uncertainly, carefully parked it on his lawn, staggered out of his car, fumbled for his key, put the key in the lock, and managed after a great deal of difficulty to get the door open, fell flat on his face on the floor. Who would be standing there but his wife, who looked down and said "well, what've you got to say for yourself?". Summoning all the dignity he could, he said, "well, I don't have any prepared remarks, but I'll be glad to address questions from the floor."

DR. DEMILLO: First question.

MR. MIKE KRESS, (SUNDSTRAND DATA CONTROL, INC.): There's a very comprehensive article in this month's issue of "High Technology Magazine" on Ada, written by James Fawcett, who is the publisher of "Defense Electronics Magazine". One of the interesting comments that I noticed was that "despite the push for Ada, some programming gurus strike a cautious note, observing that Ada is extremely complex and requires highly skilled programmers to write effective programs". And, then he quotes from a Mr. Michael Ryer, who is head of the Ada Program Office at Intermetrics, saying that "Ada will be completely successful or a total failure. We will know by 1985, when the first Army contracts mandating Ada will be nearing completion". I was wondering if perhaps Colonel Campbell could address whether that seems to be a valid perception of the complexity of Ada, and if so, has that been considered in the decision to embrace Ada? And, are we really looking at this serious a problem that we would probably have to send our programmers back to school for a week or a month to learn to program in Ada?

COL. J. FRANK CAMPBELL (U.S. ARMY MATERIEL DEVELOPMENT AND READINESS COMMAND): A couple of comments. Number one, you might be wondering why the Army is pushing so hard for Ada. Going back to '79, TACPOL was our standard language at the time, and had widespread use in four systems. We looked around and saw what was happening, and it really didn't make sense for us to invest a lot of money in that particular language at that time. Ada was on the horizon, and we put our money there and have been pushing hard. We've sort of got an advantage over the other services, they have their standard languages. Incidentally, we use those in many of our systems. We had that wonderful opportunity of proliferation that we had to do something about, so we did go for Ada. Two things I would say. Number one, part of the complexity that people are dealing with in Ada exists in our current systems. Yet there is no programming language that can deal with this complexity, it is done primarily with assembly language. Maybe there are other ways to ensure that this is dealt with, using Pascal, C, and that sort of thing. You could put those two together and do essentially the same thing. So, I think that the complexity is there, and to have a language that handles that complexity, indeed, it is going to be complex, there's no two ways about it. The other thing, I will challenge you a little bit, is that from a testing standpoint, the little bit I know about engineering, the thing that you use to test with is, in general, more complex than the thing you are testing, just by nature. Admittedly, Ada has a lot of features in it that would help us during the life cycle. It has a lot of features in it to correct or to find the errors that are normally done by programmers. The complexity probably deals in two or three or four percent of the language. My perception is that if we get to the point where we have to block out some of those features in order to make Ada really usable, we will have given it a good shot. Admittedly, Ada is complex, but it's complexity is directed to put the things in the compiler to take the load off the programmer to do a lot of checking.

MR. KRESS: Will the familiarity to use Ada require formal training for programmers, who will have to go back to school for a week or can they pick this up reasonably in the self-taught mode?

COL. CAMPBELL: There, again, it depends on where you are coming from. We've got some efforts going on evaluating that. Who needs to know what? You don't train a systems programmer in college, he learns a little bit about JCL, etc., but the guy that actually designs that is an experienced guy. I am, by no means, an expert in programming. The little bit that I've had in Fortran and Pascal, there are very few concepts that I can't grasp very quickly. Whether or not I could put them to good use and write good code, that will take some experience.

MR. MOORE: Earlier, Capt. Boslaugh talked about testing the requirements and the design of the software. The first day of the conference, Mr. San Antonio gave us an overview of how to, quantifiably measure the design specification once it was mechanized. My observation is that once we mechanize a design specification, it becomes software, open to all the errors that are possible in software. But, don't we also have a problem insofar as the proliferation of these character-string, generic, design aids, that we are really just glossing over? We've been talking about four different ones in the course of this conference. Is there any work being done to perhaps standardize these design aids?

CAPT. DAVID BOSLAUGH (NAVY MATERIAL COMMAND): If I may, I would like to take a crack at that. You're absolutely right. Over the last few years in preparation for trying to design an all up software engineering environment and to design a software factory, we surveyed the tools. I will tell you one thing. We've sure got plenty of tools. We've got tools that address every part of the software life cycle, and we've got tools for everything. The trouble is that they're in bits and pieces. They're written in many different kinds of languages, many are unique to different instruction sets and unique to processes. One of the first things we want to do in the conceptualizing of an overall environment is exactly that, is cut down the tools to a few good ones, first of all, and also make the tools compatible so the output from one tool flows logically into the next one. Also, if we can do it right, get as much automated production of documentation from those tools, as we can. But, you're right, the next part of the software monster that's going to eat us up is going to be an incredible proliferation of tools that don't fit and match each other.

QUESTIONER: Providing we get that solved. The next problem would be, is there any work being done in perhaps establishing a stronger link between the metrics that Mr. San Antonio talked about for measuring the design specification and the metrics used in the evaluation of the actual executable software? If that takes place, would it be possible also to perhaps measure a system specification with a view of that as being a prototype?

COL. EDWARD AKERLUND, (AIR FORCE SYSTEMS COMMAND): Are you asking, is a specification a prototype?

QUESTIONER: Provided the metrics are in hand to the point where their is consistency to the metrics used to judge the completeness of the design specification, matching that to the actual design specification for the executable software, wouldn't we be able to perhaps flush out problems more right at the design area as opposed to waiting until we have executable software?

COL. AKERLUND: Are you talking about going through the program design language and specifications?

QUESTIONER: Yes. PSL's or whatever.

COL. AKERLUND: And you would associate metrics with the PDL?

QUESTIONER: Yes.

COL. AKERLUND: I don't know of any work that tracks through all of that. I'm not completely familiar with all our laboratories' activities either.

DR. DEMILLO: I can respond to that. Yes. In general, there are people who are interested in executable specifications, testable specifications, and look at specification as a programming task. That's not the growth area in testing right now, but, there is some interest.

QUESTIONER: Just a question considering life cycle cost implications. Has there been any effort to look at tools to provide documentation, automatic generation of documentation, not only for the developer, but also over the entire life cycle. I think one of the big problems is that the documentation many times ends up as a last minute job, written by programmers, who perhaps did not develop the original code, etc., etc., and of course, in many cases, frankly it gets worse, it goes downhill from there, over the 10-20 years of the useful life of the system. I'm just wondering if there are any attempts to include some guidance relative to documentation tools to ensure that the documentation supports the life cycle use of the system?

CAPT. BOSLAUGH: I mentioned our hopes for a complete comprehensive software engineering environment which would cover the entire life cycle, and as one of our strong desires, we would like automated documentation production. One of the reasons we would like it is that we think that is a good way to get contractors and other agencies to use the standard environment, to drop out the documentation in the form of the contractual data item definitions that the government asks for. Now, that's our desire. I don't know how much work is going on in those lines. I've heard of bits and pieces of technology that's slanting toward automated documentation production. I think there is probably an awful lot of work to do. We very definitely want this kind of capability in an environment, and as part of the products of the DoD Software Initiative, we hope to be able to specify that in a lot greater detail.

COL. CAMPBELL: Let me just add one thing to that. The same thing applies in a lot of other areas in terms of documentation. The ILS package, the TecData package for hardware, runs, in general, that depends on size, complexity and a lot of other things, a million and a half or two per copy. That's one of the areas that we've seen our reduction of kinds of computers helping tremendously. The circumstance is that a guy out of my office works weekends right now, and otherwise, on a team that's looking at the way to do that with the new video disc technology and other technology. Part of our problem is to come to some agreement as to the environment of the data base. As it now stands, with some 40 different software environments, that we would have across that system, it's uneconomical to try to do something like that, and very definitely that is why we're trying to get to a common environment. We know it's not going to stay constant, it's going to be changing, etc., and has to be improved on each system. But, that's one of the things that we hope will come.

COL. AKERLUND: A quick comment on documentation. Many times I've been asked about why the government wants so much documentation on software. What I've been trying to communicate to industry, is that the documentation we want is the same documentation that the software developer needs. That is, if the designer leaves in the middle of the design, or if the coder leaves in the middle of the code, can the next person pick up without sizable delay in completing the software package and testing? If it's documented to the point that the person who's working on it can leave, and another person can pick up using the documents that are available, to complete the task, then the documents are at a level that we need to support that system. Today where complex systems are created by many different agencies, keeping that document current and related to the systems, is most difficult and probably one of our most important tasks.

MR. RUBEY: Let me address a question to Ronnie and other people in STEP, and that's the recommendation to develop tools, which has come through very strong in the last couple of days. Maybe we're pushing forward, I would suggest, as a skeptic, a little too fast for tools. We're not very good in testing techniques and methodologies. We don't really know how to do the process manually very well; our fault is in imagination and thinking of better ways to test the software. What are your feelings of why you think just being able to do things automatically is going to solve that problem? We can't do it manually. We go back to our carpenter analogy, it is as if we're going into the Sears tool department, we're looking around, but we've never been in carpentry before, and we buy the fanciest and most elaborate radial arm saw. I'm afraid some of these expensive tools, like that radial arm saw, are going to sit there unused or we'll start hacking up pieces of wood. Rather than build a nice software house, we'll have a lot of firewood with the tools.

DR. DEMILLO: I think that's true. I think there's a chance that a fancy tool will be unused. As Lee Osterweil said, the tool's that are out there that he sees don't qualify as tools by his definition, because they aren't usable enough, they're not friendly. That is certainly a danger, but, I don't think the issue is do it manually versus do it automatically. There are things that you can not do manually. Where there are things that you do manually that have only subjective results, and manual testing is often a code word for debugging. I think there is a difference between testing and debugging. For the kinds of software systems we're looking at, the test cases have to be large, the test harnesses have to be complex, the data that has to be recorded through various testing phases is voluminous and has to be documented. I don't think there is a choice between doing it manually and doing it automatically.

MR. RUBEY: The question I have is do we know enough to even do it manually, given enormous labor? Then we might think of defining a tool. My scepticism is that if we're not smart enough to even be able to do it manually, then a tool isn't going to help us if we don't know what we should be doing. We're given a powerful tool, and we apply the tool, regardless of what the need is.

DR. DEMILLO: I have a comment, but I'll let Ronnie or Don respond first.

MS. MARTIN: First of all, your analogy made me come with another one. One of the things that was discussed is that we are at a level of a hammer and a screwdriver. If you give me a piece of wood and a hammer and a screwdriver, and I want to cut that wood up, it's going to be real messy. It's not going to do a very nice job. That's kind of the same thing as the manual testing. There are some things, as Rich said, you can't do manually. There are certainly deficiencies in all areas, in terms of methodologies and whether or not we really know what we're doing anyway. As we said in one of the recommendations, there needs to be an analysis, an objective analysis, of the various tools and methodologies to determine which ones are better than the others. We're not saying we have all the answers there. But, at the same time, we do need some kind of tools that are available to the general public, instead of proprietary tools that are redeveloped by each organization over and over again for their specific projects. That's a very large waste in terms of money and time, and everything else. Some tools are definitely needed. But, there does need to be research in terms of exactly which tools and which methodologies are the most effective. Does that answer your question?

MR. RUBEY: I think that answers my question. I think the effective thing in your report would be a specific recommendation of a particular tool you think would be effective rather than, because of the limited time we have had just talking generically. The best thing to say is we need hammers, saws, screwdrivers, and with this we can build a good house, rather than saying, let's go to Sears and buy a big box of tools and hope there will be some in the box that will eventually prove useful.

DR. DEMILLO: There's really no way of making that kind of comparison right now. The kind of data you would need to be able to select out a specific set of tools and recommend these is not there. It's a very dangerous thing to do in the second place. When you do that, you tend to focus the attention on those tools, and that's a kissing cousin of standardization anyway. It would tend to freeze the technology. The technology is not that advanced right now, but it's better than none. You can expect it to get better. That's not a reason for not using the tools.

MR. NEEMAN: In light of the software idea first, OSD has been saying, and many of the speakers have expressed concern about. What steps have OSD and the services taken to acquire additional resources, both personnel and high grades to provide this emphasis and what success has been met with by OSD and so on? And as an example, what happens when a new PM is chartered? Do we say 50% of the cost of the new system will be software, therefore, 50% of the engineering services in the PM, the personnel must then be software qualified or work on the software areas? Is anybody working on those type of techniques?

COL. CAMPBELL: I guess the three of us were in a meeting earlier this week and talked about those same sorts of things, if I recall. There is a real problem there, and I had that in my chart software is manpower. I find it not true just in the military, but I find it true elsewhere in industry. That guy just cannot take on more because he doesn't have the talent or the capability there to do it now. I think it's a generic problem. The approach that we've got in the Army, again with the little diagram of the post-deployment software support center? We do have some resources scheduled to come in there, and we will build some knowledge bases, there. I don't call them centers of expertise or anything like that. At any rate, there will be a knowledge base there eventually to transfer. It's a real problem, and I don't know where the people are coming from.

CAPT. BOSLAUGH: You really like to rub salt in the wounds, don't you? You're talking about a very real problem, a source of constant frustration, especially during a time when even though the defense budget, they keep telling us it's going up. There's a very tight lid on manpower of any sort, especially a lid on the high grades. That coupled with a nationwide, what seems to be a growing shortage of software smart people, really makes our job fun. There are indications of at least some people at high places that are starting to get sensitive to this. I'll give you an example. The people who got the DoD Software Initiative in motion were at least perceptive enough to realize that it wouldn't go anywhere unless some additional ceiling points were identified with that Initiative, and actually given to the military departments to manage it. That is a sign of enlightenment, it's not a lot of ceiling points, it's just a handful, but at least, it's a sign that some very senior people are starting to realize that we don't have enough people nor the right kinds of people to really grapple with the software monster. I wish I could say that more is really being done, but, it isn't.

MR. NEEMAN: Let me add a part to that. I realize everybody is saying we need more people, and everybody tends to throw up their hands and say, we can't do anything about that. We have more and more studies, and we get a few people here and there, and they take a few from somebody else, and say, well, now it's your software even though you were yesterday a chemical engineer, or something. I just don't see a concerted effort either from OSD or even all the services saying we have to do this. Maybe we have to get people that are trained in trying to get more resources, which as far as I know, we don't have any of those that are working on these types of problems. Maybe we do, and I'm not trying to say it's not a difficult problem, but I just don't see a concerted effort. It seems like everybody is saying, we can't do much about that.

COL. CAMPBELL: Let me just add a couple of things. Just in the instance of our headquarters, at least they recognized the fact and they put 17-20 people together in a group to try to do something about it, and that was done last year. I talked to a guy in the Air Force that is on one of the other groups that we are on. He said he started with something like 200 people eight years ago, and now it's more like 1700. Those things will happen, but they're not going to happen overnight. I know in terms of even trying to put my office together, I couldn't have handled 50 people in one year and gotten them organized. It just takes time to do that. So, I refer to it as a social process that we will all become a little bit smarter over time. The harder we work at it, the quicker we will get there, but it's not going to happen overnight, that's my personal opinion.

DR. DEMILLO: I would just like to add that that's not a problem that's unique to the military. Manpower shortage in computer science, data processing, cuts across academic sectors, military, industry and, I think there is, if not a concerted effort, there are at least pockets of concern for that. The draft Software Technology Initiative included a significant upgrade of support to graduate computer science programs in universities, which I think addresses the problem directly.

CAPT. BOSLAUGH: One way we see out is, first of all, we're always going to have to depend on a certain cadre of very skilled master software crafters. Also, we can do a lot in the way of allowing everybody to be their own programmer, and a big thrust in environments is to build very user friendly environments in which the main qualification you will have to have to write a program is to know what you want your system to do. The environment will then help you write your program. I think the programming is going to have to move down and turn everybody into their own programmer. That's one of the ways we know we have to go to solve the problem.

COL. AKERLUND: One more comment on personnel from the Air Force perspective. We've have trouble trying to identify the people who have the skills. They are out there, but they're not classified. We've taken a new approach. I'll tell you about it. It's not happening yet, but, for the Systems Command, the command who's responsible for acquiring the system, we have built a software survival course. The intent is to have everyone that comes into Systems Command attend this course first before they enter the program office. It's actually a two phase course, there is an acquisitions part of it, and then there's the software survival course, which I view somewhat as a laboratory for the acquisition course. We're planning on a pilot in March and expect it to be going full speed in October. It is an attempt at bringing folks who are coming into the command, second lieutenants and the young civilians coming aboard, to get them not only knowledgeable with the acquisition process, but knowledgeable about how to acquire software, how to plan for it, how to make sure it is appropriately managed while we are acquiring it.

COL. FALK: Maybe my question is in a couple of phases. We've heard about the Software Initiative, and $200 some million being applied to that, whatever that is, and I'm part of it, over the next few years. Do we have a fund to accomplish the tool development and other initiatives that this report is going to report out on, your T&E report, is going to report on? Or, will you join the Software Initiative and put your wants on the list of things that has developed or will be developing over the next years?

MR. GREENLEE: I'm sorry Sam DiNitto is not here to address that because I can't talk to the Software Technology Initiative at large as well as others here can. It's my understanding that the funding that will be spent by the STI will cover major research needs. Before that, and I think it was well commented, that we need to determine that there is a requirement for further tools. That's been surfaced as a potential recommendation of this group. I believe that we owe ourselves a very careful evaluation as to whether we need more tools or simply need to make available the ones that exist. The details of the STI, in my understanding, are yet to be fleshed out. I believe that Sam commented, there is a workshop later this month, there's an executive committee meeting at the end of the month, and I'll stand ready to be corrected, as far as to where the bucks go and to what areas, I don't believe it's been specified yet. With reference to earlier questions though, I would remind us all that development of personnel was one of the specific objectives of that STI.

COL. CAMPBELL: He didn't say anything about another pot of money.

COL. FALK: That was the point I wanted to clear up. This question was prompted by a question phrased in a little different way, by a two star Air Force General to Dr. Martin. What else have you got going in the DoD, after she had told the executive committee about the software initiative or after Col. Druffel had briefed the executive committee. She reported that one of the things was this project, and we reported to General Welch that we were coming over to find out what this was all about.

MR. GREENLEE: OK. I talked to General Welch at that meeting afterward, and explained to him directly that there was not in the offing any funding out of the T&E community to support tool development. We construe our efforts as distinct from, but hopefully not uncoordinated with, the STI at large. That in my understanding, will be the source of funds for any research initiatives, including software tool development, should that be deemed a requirement. That is a good question.

DR. DEMILLO: None of the presentations this morning mentioned networking distributed computing systems of processes that are loosely coupled and the software that runs them and controls them. There is a whole bunch of reliability problems that come up in those settings. Is there an awareness of that in the services?

COL. CAMPBELL: I probably didn't do a good job of describing, but the one chart that I showed, at the various nodes, in fact I didn't have time to describe it. The one thing that we see is that in that five sided thing that I had up there on a diagram perhaps 50 if not more, somewhere between 40-75% of that kind of software could be common between those nodes. We don't see a reason then to have different processors for similar kinds of things, developing that software on different ISA's. In addition to the things that I showed up there today, we do have two projects right now, in fact, there were reviews this week, for standard operating systems for the military computer family. That's in the initial definition stage, writing the requirements for it, and as the Nebula machines become available, etc., there will be a build phase to arrive at something to handle that. But, that's one of the reasons we feel we need a standard environment and a standard operating system available in as many places as we can. We'll never get one that does everything in every kind of system.

COL. AKERLUND: One other comment. Systems of systems, that can be used for a loosely connected set of word processors, between buildings in a network, or could describe very complex communications systems, including massive antennas, satellites, communications processors, or even weapon systems associated with it. Someone needs to be really thinking about the testing of that system, are the pieces part of the performance or is the whole the performance? System control. That's a very important part. We anticipate seeing a lot of systems control being passed through communications protocol, other parts of the system understand what the other parts are doing. How do you test for that? What is considered reliable and what isn't? Really important thing to think about.

COL. CAMPBELL: One of the problems you run into, particularly in the command and control environment, I remember in 1979 we had an old system a lot of people have heard bad things about. I never will forget going up to OSD, and the guy wanted us to structure a task down at Ft. Hood in a period of six months that would take that command and control system, put it into three different configurations, and run exercises, if you will, to determine which alternative was the best one. If anybody has ever tried to set up an office and automate it and you get your SOP's and everybody knows what the software is doing and what the input and what the outputs are; and then pull another group together, write another SOP for office operation, and do that again, and then do it a third time in six months, and compare the results, I defy you to do it. I defy you to even tell me what the measures of performance are, how you measure it, we have a long way to go in that category.

MR. EDWARD G. JACQUES (NAVAL SEA SYSTEMS COMMAND): Several of the problems you have been discussing are relating to more than just software tools in the generic sense. I think you are getting into an area of simulation involving many systems. In a case like this, the system cannot really be tested at the contractor's plant for obvious reasons of funding and scheduling to develop such a facility. Can any one comment, in general, on what the services are doing to have a service laboratory responsible for a particular product area and testing it at that laboratory where a transition is made at one point in the development?

CAPT. BOSLAUGH: I'll mention one which you are probably involved in, and that's the Naval Ocean Systems Center in San Diego. The prototype aircraft carrier, operational test site and the other test sites for combat direction systems. NAVELEX, Naval Electronics Systems Command, is going to be building their first C^3 software factory in the next few years, which will have a massive simulation capability and the ability to bring in real live command and control type communications inputs. There are not enough of these centers being planned and designed yet, though, is my own personal feeling.

COL. AKERLUND: In my experience, we end up at the nth hour doing simulation devices because we realize that systems can't be physically put together or you can't always obtain all the pieces. It becomes a very difficult problem. Even the systems we are seeing today are larger than laboratory environments, they really go across many different disciplines, it becomes a very difficult problem. And, it needs to have the frontend planning, that people here have been talking about, very important. Some of our laboratories are working in very specific areas, but when you get these massive complex systems, including weapons systems, communications, and other things, the test of that system is most difficult.

MR. JACQUES: I agree with you, but I don't think the PM can really take that initiative. Because of planning requirements, I think it takes some higher level initiative. I'm more familiar with the NOSC from the torpedo simulation work that they've been doing for quite a number of years. It's a very successful tool used by many contractors. But, I don't always see this in other areas. It also puts the government in a position where they are able to control the testing of the final product.

MR. GREENLEE: There are basically two ways that the land based test sites get established and funded. One is as you indicate, by the program manager who needs such a facility to bring out his system. There's also an institutional fund administered within each service for the general improvement and modernization of test ranges and test sites, including operation and maintenance. Normally, to speak in rough terms, a small test site is deemed to be the responsibility of the benefiting program manager. A large test site, such as the TRITAC facility at Ft. Huachuca, particularly if there's more than one benefiting service, is typically funded institutionally by either a lead service or split funding among the benefiting services. So, it's not always a case of, if the PM can't afford it, we don't have it. There is a mechanism which is monitored at OSD level. The money is fenced, and protected. Hopefully, major needs are met in that way without taking it out of the pocket of the program manager.

DR. DEMILLO: That closes our final panel. I'd like to thank you all for your participation over the last two and a half days. I'd also like to thank our speakers and the sponsoring agencies, NSIA and OSD.

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

VOLUME 6

TACTICAL COMPUTER SYSTEM APPLICABILITY STUDY

BY

JAMES F. LEATHRUM

SUBMITTED TO
THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION
AND
THE OFFICE OF NAVAL RESEARCH
ONR CONTRACT NUMBER N00014-79-C-0231

SCHOOL OF INFORMATION AND COMPUTER SCIENCE
GEORGIA INSTITUTE OF TECHNOLOGY
ATLANTA, GEORGIA 30332

OSD/DDT&E
SOFTWARE TEST AND EVALUATION PROJECT

PHASES I AND II
FINAL REPORT

Volume 6
Tactical Computer System Applicability Study

by

J. F. Leathrum

Department of Electrical and Computer Engineering
Clemson University
Clemson, SC 29631
(803) 656-3190

SUBMITTED BY
GEORGIA INSTITUTE OF TECHNOLOGY

TO

THE OFFICE OF THE SECRETARY OF DEFENSE
DIRECTOR DEFENSE TEST AND EVALUATION

AND

THE OFFICE OF NAVAL RESEARCH

FOR

ONR CONTRACT NO. N00014-79-C-0231
Subcontract 2G36661

FOREWORD

This volume is one of a set of reports on Software Test and Evaluation prepared by the Georgia Institute of Technology for The Office of the Secretary of Defense/Director Defense Test and Evaluation under Office of Naval Research Contract N00014-79-C-0231.

Comments should be directed to: Director Defense Test and Evaluation (Strategic, Naval, and C^3I Systems), OSD/OUSDRE, The Pentagon, Washington, D.C. 20301.

Volumes in this set include:

Volume 1: Final Report and Recommendations
Volume 2: Software Test and Evaluation:
State-of-the-Art Overview
Volume 3: Software Test and Evaluation:
Current Defense Practices Overview
Volume 4: Transcript of STEP Workshop, March 1982
Volume 5: Report of Expert Panel on Software Test and Evaluation
Volume 6: Tactical Computer System Applicability Study

Contents

PAGE

Abstract 1

1. Modelling the Tactical Computer System (TCS) 2

1.1 Introduction 2

1.2 Modelling the TCS's Digital Input/Output Module (DIOM) 4

1.3 Structure of the Simulation Facility 7

1.4 Device Interactions with the Simulation Environment 11

1.5 Simulatin Results 12

1.6 Multi-bus Models 15

2. Analysis of Message Traffic 19

2.1 Projected Message Sizes and Frequency 20

2.2 Hardware Loading 26

3. Memory Resource Contention 27

3.1 A View of the TCS Memory Allocation Scheme 27

3.2 Testing the Allocation Scheme 30

4. References 32

Abstract

As part of the on-going development of the Tactical Computer System for the U. S. Army, this study develops a modelling methodology for communications networks. The models are based upon the paradigm that all the critical components; i.e. buses, communications controllers, memories, etc., are processors. The models are composed of multi-task abstractions of processor networks.

The impact of the work reported here is best viewed in the context of a period which began with no quantitative tools to use in the applicability evaluation of the Tactical Computer Systems and which culminated in a decision to re-design the systems software. Although a complete model was never implemented, the increasingly quantitative insights obtained from modelling raised questions which soon indicated that the system was quite over-designed in its raw message handling capability. The lack of flexibility in the memory management software ultimately proved to be a critical shortcoming which was uncovered in field tests. This report describes the tools developed to bring the message handling capacity for the system into focus.

1. Modelling the Tactical Computer System (TCS)

1.1 Introduction

Computer systems modelling, or systems modelling in general, is an activity characterized by difficult and unprecedented choices of levels of abstraction. The problem of generation and validation of appropriate models may be addressed along several dimensions:

1. Appropriate representation of detail,

2. Analytic tractability,

3. Tools of composition of complex models from simple, primitive ones.

Because of the presumably deterministic behavior of computer systems, one is naturally drawn to modelling them at the extreme of minute detail. Such models are often as difficult to build and maintain as the system itself and, thus, offer little assistance in engineering judgements about performance. Amongst the most successful classes of models of computer systems have been those derived from queuing theory. The various components of a computer system are viewed as service centers for jobs passing through the system. Queuing would be permitted at each component such as device controller, bus, memory, processor, etc. These models often suffer from a combination of too high a level of abstraction as well as mathematical intractability (CS81, La81). The alternative of an "operational model" represents an even higher level of abstraction with an attendant gain in tractability (DB78, ZSEG82). If one is to maintain some flexibility in the level of detail and tractability, simulation presents a possible alternative provided the process of constructing a simulation can be automated and simplified. Simulation languages have been under development for several decades now, and they have proven to be very useful tools provided the systems composition process adheres to rules which are often determined by limitation of the language's host. Most notably, the user of most extant simulation languages finds himself imposing artificial and irrelevant sequencing and procedureness upon the system being modelled.

Given the need for a generic modelling facility of the sort that would address some of the performance issues related to the TCS, and given that the TCS, itself, is a generic design which is almost certain to undergo radical change before being fully applied, it seems appropriate to address some of the problems which arise in composing simulations. It is not proposed that the approach taken here should be exclusive of more analytic and more approximate methods, but only that readily composed simulations are an important part of the mix of useful tools.

The paradigm adopted for the simulation facility developed herein is that "everything is a processor". Here, a processor is characterized by clearly identifiable interpretation cycle, some resources dedicated to communications with other "processors", and a queue of active processes. In addition to processors as we know them, we will model device controllers, buses, human beings, etc. as all being processors. In addition to the obvious composibility of such simple one-of-a-type primitive parts, this view of computer systems is further sustained by the empirical observations that computer related hardware tends to undergo a "wheel of reincarnation". That is, as the need for a new device is realized, it is first built to satisfy the stated need. As the device's capability is enhanced, it tends to evolve toward being a processor. At some point this ultimate processor may spin-off new devices which, in turn, are re-incarnated as processors.

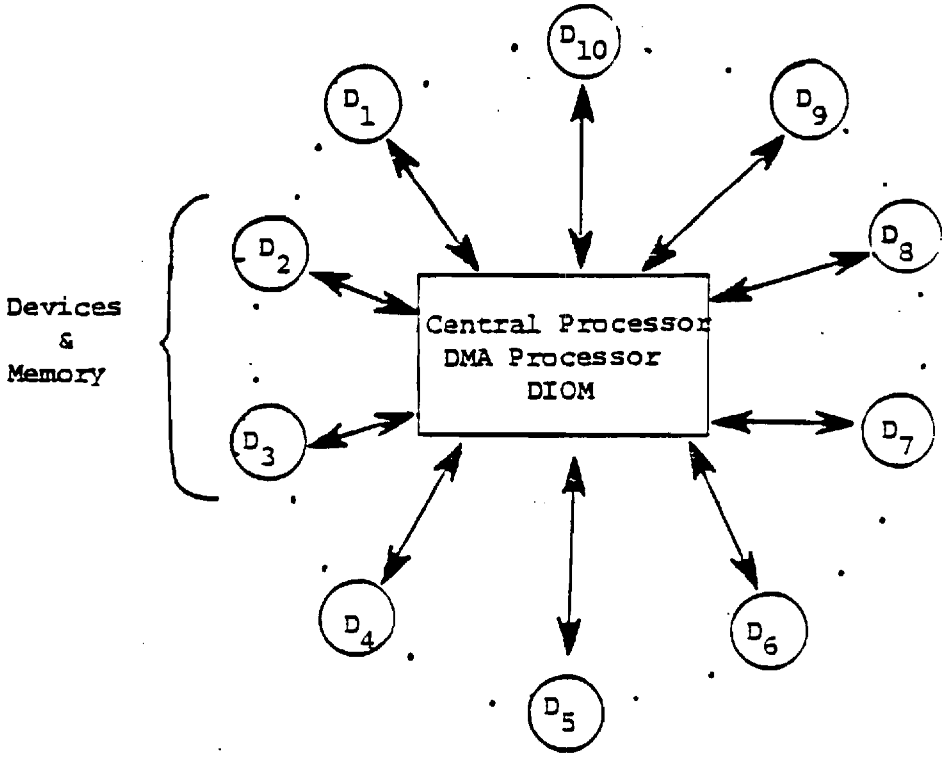
For the purpose of modelling and simulation, it is convenient to capture the re-incarnation at the point of being a processor. Less developed components can usually be approximated by choice of parameters in the processor. (i.e. a bus processor with a maximum queue size of 1 is a good approximation to a UNIBUS.) There are numerous examples of the re-incarnation of processors to wit: disk controllers, multi-plexers, terminals, and, most recently, buses.

Thus, our model of a computer system will be a collection of inter-connected processors. To capture the reality of any one instance of a system, the individual processors will be parameterized as a means of limiting the generality of their behavior.

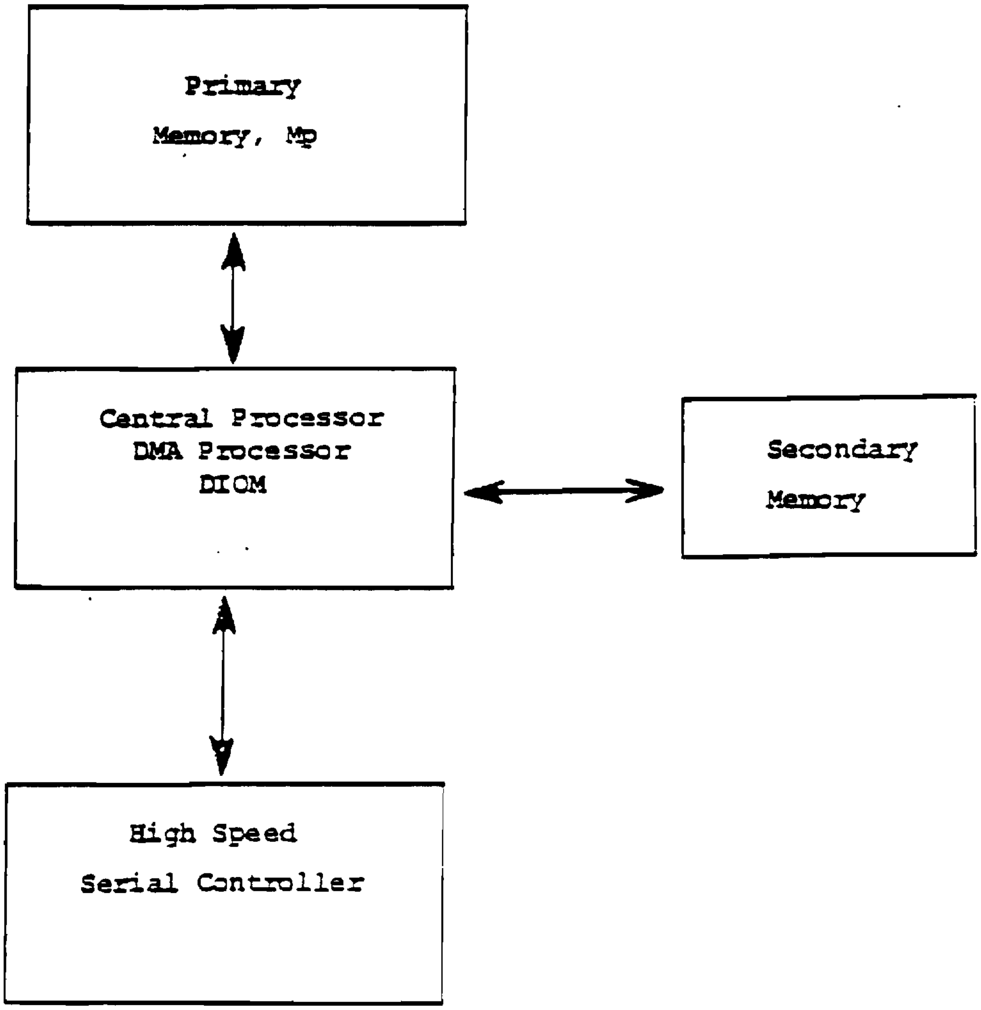
1.2 Modelling the TCS's Digital Input/Output Module (DIOM)

In an earlier analysis of the TCS, P. Enslow (En81) concluded that there was some reason the question whether the Digital Input/Output Module (DIOM) of the TCS could handle the projected message load. This question motivated the choice of this component as the focus of the first attempt to apply the modelling strategy outlined in the previous section. The DIOM is viewed as a processor with a number of devices (i.e. processors) connected to it and attempting to pass data along the

DIOM. Functionally the DIOM is very similar to a bus, but some distinction must be made in the TCS between the processor-memory bus and the DIOM. For the message handling applications, the central processor, P , is driven entirely by interrputs from the devices. Thus, for all practical purposes, the P is the DIOM processor. The configuration may be thought of as:



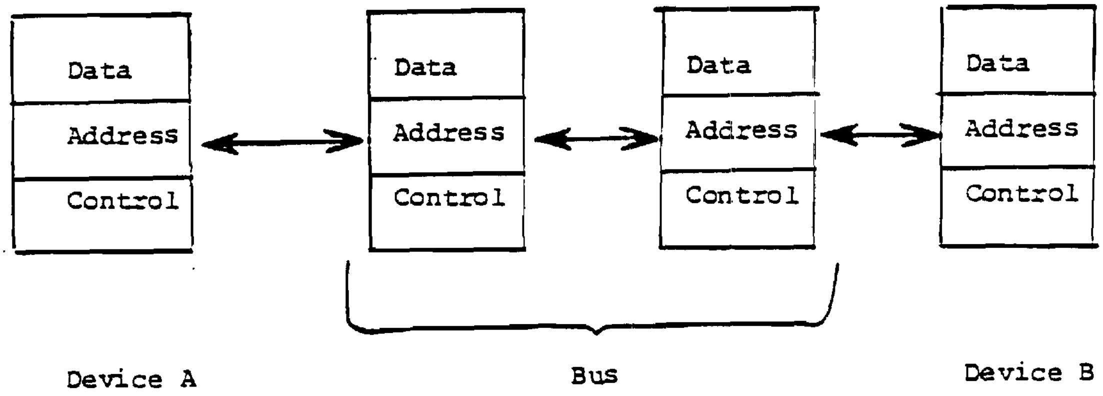
The particular subset of immediate concern is:



The question to be addressed by simulation is whether such a system can sustain the expected data rates between the devices.

1.3 Structure of the Simulation Facility

The simulation facility proposed in the previous section has been implemented as a prototype of the approach wherein the components are universely viewed as processor. For the purposes of the prototype, the processors are classified either as the bus which serves as the master processor or devices which are characterized by having interfaces with an external environment. With respect to a single message, the interaction may be viewed as:



Thus the interfaces are defined by:

```
          interface =>data              : MESSAGE
                       Addr             : address_RECORD
                       Contr            : control_record

     address_record =>destination       : DEVICE
                       location         : INTEGER

     control_record =>comm_type         : S_OR_R
                       write_complete   : BOOLEAN
                       interrupt        : BOOLEAN
                       int_enable       : BOOLEAN
                       dma_count        : INTEGER
                       priority         : INTEGER

           S_OR_R::=>                     SEND
                                         |RECEIVE
```

The bus, in turn, has associated with it a queue of pending data transfers

```
                  bus_queue  => seq_of_queue_elements

              queue_element  => interface
```

(Note: The "Diana" formalism is used above to describe the essentials of the internal structures (GW81)) The bus processor handles the transfer of the interface structures from the source device to the destination device. The mechanism for the transfer may be viewed as:

```
                +-------------------------+
                |                         |
                |                         |
   bus cycle:   | Update Clock            |
                |                         |
                |                         |
                +-------------------------+
                |                         |
                |                         |
                | Device Poll*            |
                |                         |
                |                         |
                +-------------------------+
                |                         |
                |                         |
                | Interrupt Service*      |
                |                         |
                |                         |
                +-------------------------+
                |                         |
                |                         |
                | Data Transfer           |
                |                         |
                |                         |
                +-------------------------+
```

(Here, vertical ordering will indicate rough sequencing, * will indicate iteration, and o will indicate selection) The update of the clock in the bus will provide sychonization with an external clock if necessary. The device poll module will handle interaction with external data sources or sinks.

Device Poll:

Update Clock
External Poll
Interrupt Service
Set Bus Interrupt

Interrupt Service:

Move data to the bus
Queue up the interface

Data Transfer:

Unqueue Interface
Move data across the bus
Interrupt the destination device

The bus cycle and device poll components are purposely shown to be very similar in structures. They are, in fact, implemented as parallel tasks, and the control_records serve as semaphores which synchronize their respective cycles. Thus, each device and the bus are instances of processes. The distinction between a device and a bus is a matter of convenience in that the device must interact with the simulation environment, but the bus interacts only with devices and other buses.

1.4 Device Interactions with the Simulation Environment

Each device in the simulation system must routinely interact with the outside world. This interaction is standardized by specifying the languages which describe the events arriving at the device and the language generated by devices responses. The arrival of events is described by:

```
device process ::=  events
         event ::=  time  control_part  data_part.
  control part ::=  auto  No. of events
                    Mean Interarrival time
                   |single
     data_part ::=  In DMA Count Data
                     Destination Location
                   |Out
                   |In  Data--Dma case
```

The responses are characterized by:

```
      Message ::=  Message received on Device Location
                  |Request received on Device Location
```

With respect to the arrival of events, the device tasks are viewed as interpretors of the language described by the "device process" grammar. Before a "message" can be generated by the device, that same device must encounter a "out" data part in its event sequence.

1.5 Simulation Results

As a demonstration of the utility of the simulation facility, the identification of bottlenecks in the DIOM was chosen as a test case. Using the paramaters suggested by Enslow (En81), the TCS input/output module was modelled as a high speed controller, a primary memory, a secondary memory, and the bus. The designed date handling rates are:

Bus: 16 MBPS

High Speed Controller: 2 MBPS

The primary memory is presumed to be as fast as the bus on the average, and, for this exercise, the secondary memory is presumed to be running at one half the rate of the bus on the average. Since the TCS processor is interrupt driven, the interrupt handling rate is also an important parameter and is taken to be

10^4 interrupts/sec.

Using a message structure for the high speed controller of:

(a) 400 words per message
3 interrupts per message
(uniformly distributed over the message)
16 bits per word

(b) 64 words per message
1 interrupt per message
16 bits per word

(Note: Case (a) was used by Enslow (En81). Case (b) is a more recent understanding of the message structure.)

It is apparent that the interrupt handling rate is the limiting factor. The smallest unit of time to be considered is the interrupt service time. Each device will incur interarrival times of:

(1) Memory: 1 bus cycle/interrupt

(2) Secondary Memory: 2 bus cycles/interrupt

(3) High Speed Controller:

Case a: 10.67 bus cycles/interrupt.

Case b: 5.12 bus cycles/interrupt.

Where a bus cycle is taken to be the minimum interrupt service time of 10^{-4} sec.

Each of the above cases is considered in the context of 300 messages (with acknowledgement) being passed from the high speed controller to the primary memory and 150 messages passing from the secondary controller to the primary memory. All devices are assumed to have the same priority. The bus queue is allowed to be either unlimited in size or of unit size. The latter bus queue is appropriate for the TCS.

(a) Maximum Queue Size = 2 (Note: This configuration allowed interleaved DMA transfers)

	Mean	Mean
	Interarrival	Event*
	Time	Backlog
	(Bus Cycles)	(No. of Events)
High Speed Controller	10.67	.92
Secondary Memory	2.00	4.68
High Speed Controller	5.12	.95
Secondary Memory	2.00	4.47

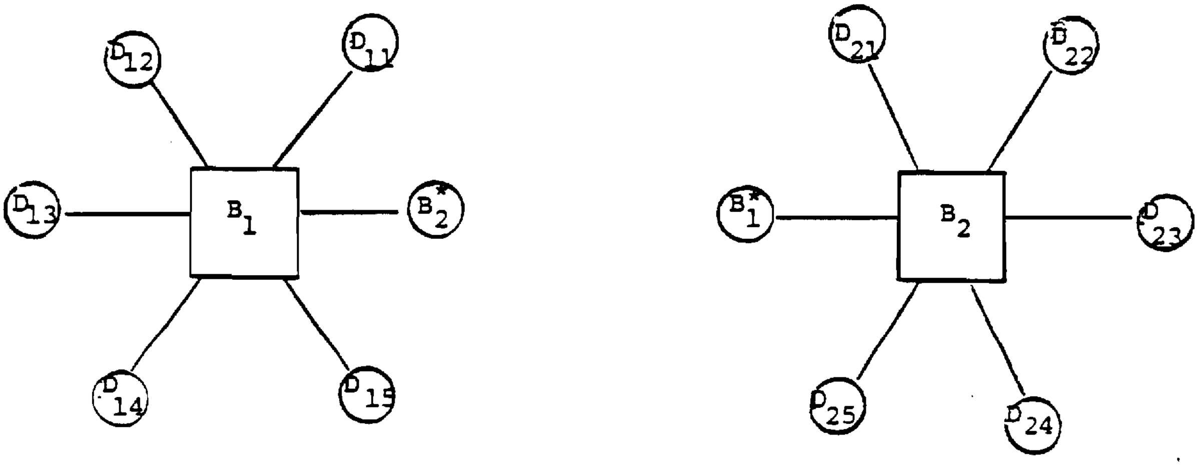
(b) Maximum Queue Size = 1 (Note: In all cases the secondary memory obtained first access to the bus)

High Speed Controller	10.67	2.92
Secondary Memory	2.00	.32
High Speed Controller	5.12	10.30
Secondary Memory	2.00	.32

(*The mean event backlog is the mean queue size of events associated with the device. It includes the event which is in process by the bus.)

1.6 Multi-Bus Models

The simulation facility described in Section 1.3 has been extended to include multi-bus architectures. The technique used to achieve this has been to allow the various busses to treat each other as devices as in:

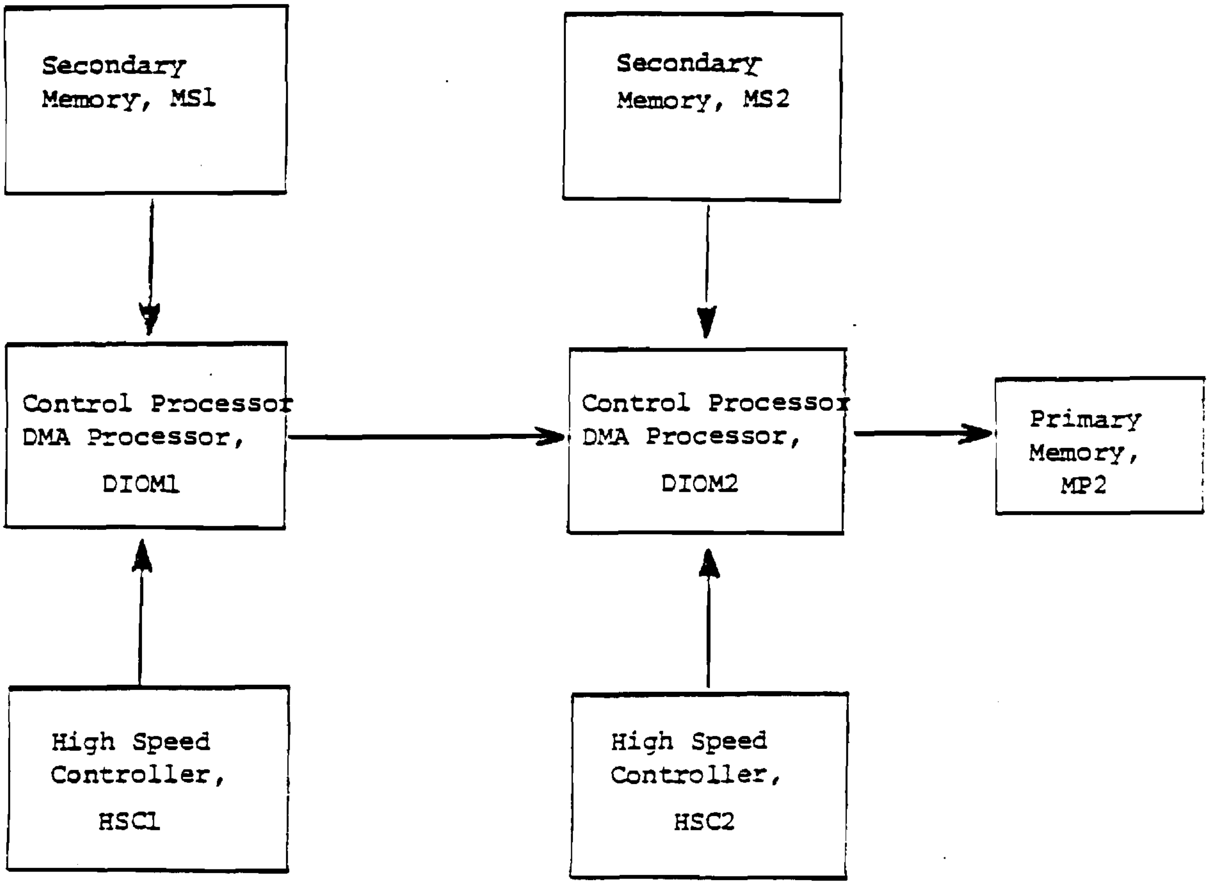


The Dij are devices in the conventional sense and are realized as individual tasks being driven by an event sequence.

The interface nodes, B_2^* and B_1^*, are treated locally as devices in that there is an applicable event sequence. However, the data is not provided by the data record of the event, but, instead, arises from the bus-to-bus connection. Thus the B_1^*, B_2^* pair are not sources or sinks for data, but most act as transfer devices only.

The synchronization of the B_1^*, B_2^* was accomplished by requiring that the sender be treated as a receiving device by its host bus. That is, if data is flowing from B_1 to B_2, then B_2^* is a receiving device with respect to B_1. The receiving device (in this example, B_1^*) is required to be in a DMA node such that it expects auto-generation of data. When the event of B_1^*, interrupting B_2 occurs, it has the effect of not only initiating the transfer across B_2 , but it also frees B_2^*, for subsequent transfers across B_1 . The effect is a single message buffer at the bus-to-bus interface. The allocation of delays to the various devices follows the same rules as with the single bus access.

As an example of the type of configuration which can be simulated, consider two buses connected according to:


Secondary Memory, MS1
Secondary Memory, MS2
Control Processor DMA Processor, DIOM1
Control Processor DMA Processor, DIOM2
Primary Memory, MP2
High Speed Controller, HSC1
High Speed Controller, HSC2

Using this configurations, message events were created using the following interarrival times:

Device	Mean Interarrival Time, Host Bus Cycles
MS1	20.00
HSC1	15.12
MS2	10.00
HSC2	5.12
MP2	1.00

The two buses may be operated with respect to independent clocks, but for the purpose of this example, the buses run at the same speed (i.e. 10^4 messages per second). The buses, DIOM1 and DIOM2, establish event sequences for the sending and receiving of messages respectively. The event sequences were random and occurred at mean interarrival times of

Sends by DIOM1	:	10.0 host bus cycles
Receives by DIOM2	:	15.0 host bus cycles

The results of the simulation for the case where each bus has a maximum queue size of 1 are shown in Table 1.1.

Table 1.1

Message Load and Mean Event Backlogs

Device	Total Delays	No. of Messages	Mean Event Backlog
MS1	6865	50	6.870
HSC1	4991	150	2.200
MS2	200	150	.133
HSC2	1038	300	.676
MP2	650	650	1.000
Sends by DIOM1	1221	200	.610
Receives by DIOM2	143	200	0.048

2. Analysis of Message Traffic

In order to assess the likelihood of resource contention problems associated with the message flow intensity, AMSAA prepared some estimates of the message sizes and arrival rates (AM82). This information included:

Inclosure	Description
1	Block Diagram of TCS.
2	Block Diagram of DIOM.
3	DIOM Input/Output Controller Block Diagram and Function Description.
4	DIOM's Bus Signal Assignments.
5	Physical Performance Characteristics of TCS to be modeled.
6	Specific Hardware Configuration of TCS to be modeled.
7	Message traffic that can be simulated to represent a TCS under "expected load".
8	OTHER ENVIRONMENTAL SIMULATION CONDITIONS.
9	Display/Keyboard Analyst traffic load.
10	OPTADS TCS Communication Design Memorandum.

Of particular interest to the modelling activity inclosures 7, 8, and 9 which establshed the expected message load on the TCS.

The immediate concern is whether this expected load approaches the capacity of either the high speed controller, the CPU, or the DIOM.

These critical capacities are:

High Speed Controller: 8000 to 32000 bps (Inclosure 6)

CPU: 10^4 interrupts per second.

DIOM: 16 mbps

2.1 Projected Message Sizes and Frequencies

The traffic loads were analyzed in terms of worse case conditions, i.e. the largest messages being handled at the highest expected rates. Likewise, each character is presumed to consume a whole word. The message loads have been converted to units of primitive messages per second where

Primitive Message: 64 words of data
64 words of address (Inclosure 8)
16 bits per word.

User Message: Integral number of primitive messages

Acknowledgements: One primitive message per user message

Interrupt Service Rate: Primitive messages per sec.
(i.e. 10^4 primitive messages per sec)

Note that Inclosure 8 states that one word of address must be transmitted for every word of data. This seems to defeat the purpose of the DMA controller, but the additional load was included for completeness.

The expected message traffic intensities were reduced to those shown in Tables 2.1 through 2.3.

Table 2.1

High Speed Controller's

Expected Message Load

	In-Coming	Out-Going
Channel	Primitive messages/sec	Primitive messages/sec
1	.0261	.0244
2,3	.0239	.0239
4,5,6,7	.178	.101
8,9,10,11,12,15,16	.182	.239
13	.0284	.0213
14	.0414	.0371
Net Load	0.4805	0.4447

Table 2.2

Data Logging: Expected Message Load

Source Channel	Primitive Messages/Sec.*
1	.0243
2,3	.0218
4,5,6,7	.169
8,9,10,11,12,15,16	.164
13	.0278
14	.0399
Net Load	0.4468

*No acknowledgement is transferred

Table 2.3

Floppy Disk Output:

Expected Message Load*

Source Channel	Primitive Message/Sec
1	.00289
2,3	.00347
4,5,6,7	0
8,9,10,11,12,15,16	.0651
13	.00868
14	0
Net Load	0.0801

*This load represents one-third of the SITREP load received, (AM82, Inclosure 8)

Table 2.4

Display/Keyboard:

Expected Message Load

Characters/Sec	Inter-arrival time, min	Primitive Messages/Sec
500	4	.0326
1100	6	.0477
1600	15	.0278
650	3	.0564
Net Load		0.1645

Table 2.5

Summary of Message Loads

Device	Primitive Messages/Sec.	Load, Bits Per Sec.
High Speed Controller	0.9252	1895
DIOM Bus	1.616	3311

2.2 Hardware Loading

Even under conditions of the largest expected messages arriving at the highest expected rate, it is apparent that the critical hardware resources remain under utilized, i.e.

Device	Load, bits/sec	Capacity, bits/sec	Per Cent of Capacity
High Speed Controller	1895	8000 to 32000	2.37 to 5.9
DIOM Bus	3311	16 X 10^6	.021

Thus, it is apparent that the projected message loads described in (AM82, inclosures 1 through 10) do not come close to reaching the capacity of the TCS in handling message traffic.

3. Memory Resource Contention

The purpose of this section is to establish a framework for the analysis of the utilization of random access memory by the TCS. The system is designed to allow the inclusion of up to 512K words of memory, but the partitioning of this memory is a potential source of contention between the various active tasks.

3.1 A View of the TCS Memory Allocation Scheme

In focussing upon the WORAM as a possible source of resource allocation problems, it is helpful to distinguish between a physical view:

```
         MEMORY::= SEQUENCE_OF_PAGES
           PAGE::= SEQUENCE OF WORDS
Where    |MEMORY|< 1024 pages (implementation dependent)
           |Page|= 1024 words
```

and a logical view:

```
     USER_MEMORY::= SEQ_OF_PAGES
     |USER_MEMORY|< 64 Pages
      No. of Users< 8
```

The addressing in each view reduces to

Physical Addressing:	10 bit page no.
	10 bit word no.
Logical Addressing:	3 bit user no. (CPU controlled)
	6 bit page no.
	10 bit word no.

The CPU has two modes for assigning user numbers

(a) Executive		(b) Executive	
DMA		DMA	
User		User	Program area
User		User	Data area
.		User	Program area
.		User	Program area
.		.	
.		.	
.		.	

The translation from logical to physical addresses is handled by a map vector:

map_vector::= SEQUENCE_OF_PAGE_DESC

PAGE_DESC=> Physical_page_no: 10 bits
Protection_part: 6 bits.

The physical page numbers in the PAGE_DESC may be shared amongst several descriptors thus allowing sharing of program and/or data areas amongst the users.

It is useful to reflect upon some more-or-less obvious implications of the design. First, in a fully implemented design, (i.e. |Memory| = 1024 pages), there is no possibility of complete allocation since the user has only a 19 bit address. Thus, the effective maximum memory size is 512K as indicated by the full logical address. Secondly, the extent of contention for memory is going to depend upon the sophistication of both the operating system and the applications programmar. There is an apparent bias toward dividing-up the memory at configuration time. The executive gets the first 65K and the DMA controller gets at least 1K words per communications channel. There seems to be an assumption that the executive will get the 64K on the CPU chassis, and the DMA and user would divide the rest. Given the speed differential between the CPU memory and the WORAM memory, the allocation between the executive and the users could be redesigned in the interest of overall speed. For instance, the generating system could spawn "users" for one-time actions or it could use data areas allocated in the user space for infrequently accessed items. Likewise, the user-to-user boundaries in memory could be restructured to allow sharing of memory.

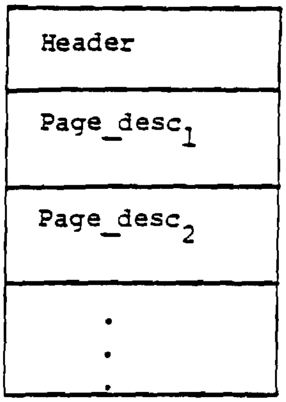
If memory is cheap, why should one worry about the sophistication of the allocation scheme? The total capacity is not the only issue here. The ability to work around bad memory boards, and bad pages may be of critical importance in the field maintenance of the system. If the whole system is relying upon the executive residing in a simple critical resource, then perhaps one should examine the possibility of configuring the system without that resource.

3.2 Testing the Allocation Scheme

With respect to probes on the memory allocator, one would assume that there is an allocation bit vector someplace in the CPU or RMU which maintains the allocation status of each page. This would be 32, 16 bit words where each bit would indicate whether the corresponding page is in use. The bit vector would appear as

1101111001..............

and the quantity of interest is the total number of pages allocated over time (i.e. the number of ones in the bit vector). An equally interesting (and perhaps more available quantity) is the total size of each map_vector over time. The total allocated pages is bounded from below by the sum of the map vector sizes. A typical format for the map vector might be



where the header would contain the size information needed. Another possibility would be to place a software probe in the allocator and de-allocator routines of the executive to keep a running count, by user and total, of the pages in use.

REFERENCES

(AM82) AMSAA, Personal Communications (from P. Ward), 1982.

(CS81) Chandy, K, Sauer, C.H. Computer Systems Performance Modeling, Prentice-Hall, (1981).

(DB78) Denning, P. J., Buzen, J. P., *The Operational Analysis of Queuing Network Models,* Computing Surveys, Volume 10, (1978), p.225.

(En81) Enslow, P. H., *The U. S. Army Tactical Computer System: A Study of its General Purpose Applicability* (1981).

(GW81) Goos, G., Wolf, W. A., *Diana Reference Manual,* (1981).

(La81) Latouche, G., *Algorithmic Analysis of a Multi-programming Multiprocessor Computer System,* January ACM, Vol.28, (1981), p.662.

(ZSEG82) Zahorjon, J., Sevick, K. C., Eager, D. L., Galler, B., *Balanced Job Bound Analysis of Queuing Networks,* CACM, Vol. 25, (1982),p.134.