STAR
(1+12)

SOFTWARE ENGINEERING LABORATORY SEL-83-008

SEL-83-008

(NASA-TM-85445) PROCEEDINGS OF THE EIGHTH ANNUAL SOFTWARE ENGINEERING WORKSHOP (NASA) 326 p HC A15/MF A01 CSCL 09B

N84-23137 THRU N84-23149

Unclas

G3/61 19076

PROCEEDINGS OF THE EIGHTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

NOVEMBER 1983

NASA

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

PROCEEDINGS

OF

EIGHTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Organized by:

Software Engineering Laboratory
GSFC

November 30, 1983

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 582.1
NASA/GSFC
Greenbelt, Maryland 20771

EIGHTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

ABOUT THE WORKSHOP

The Eighth Annual Software Engineering Workshop was held on November 3, 1983 at NASA/Goddard Space Flight Center in Greenbelt, MD. Once again, the attendance approached 250 persons representing 5 universities, 23 agencies of the federal government and 44 private companies.

The four major topics of discussion included: 1. The NASA Software Engineering Laboratory, 2. Software Testing, 3. Human Factors in Software Engineering and 4. Software Quality Assessment. As in the past years, there were 12 position papers presented (3 for each topic) followed by questions and very heavy participation by the general audience.

The workshop is organized by the Software Engineering Laboratory (SEL), whose members represent the NASA/GSFC, University of Maryland, and Computer Sciences Corporation (CSC). The meeting has been an annual event for the past 8 years (1976 to 1983), and there are plans to continue this event as long as it is felt they are productive.

This record of the meeting is generated by the SEL and is printed and distributed by the Goddard Space Flight Center. All persons who are registered on the mail list of the SEL receive a copy at no charge.

Additional information about the workshop or about the SEL may be obtained by contacting:

Mr. Frank E. McGarry
NASA/GSFC
Code 582
Greenbelt, MD 20771

301-344-6846

AGENDA

EIGHTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 3 AUDITORIUM
NOVEMBER 30, 1983

8:00 a.m.	Registration – 'Sign In' Coffee Donuts	
8:45 a.m.	INTRODUCTORY REMARKS	J. J. Quann, Deputy Director (NASA/GSFC)
9:00 a.m.	Session No. 1	Topic: Current Research in the Software Engineering Laboratory (SEL) Discussant: F. E. McGarry (NASA/GSFC)
	"Evaluating Software Engineering Technologies in the SEL"	D. Card (CSC)
	"Dynamic Metrics for Software Management"	V. Basili (University of MD)
	"Characteristics of a Rapid Prototyping Experiment"	M. Zelkowitz (University of MD)
10:30 a.m.	BREAK	
11:00 a.m.	Session No. 2	Topic: Testing Software Discussant: J. Page (CSC)
	"Structural Coverage of Functional Testing"	J. Ramsey (University of MD)
	"A Methodology for Detecting Errors"	A. Goel (Syracuse University)
	"Testing and Error Analysis of a Real-Time Controller"	C. Savolaine (Bell Labs)
12:30 p.m.	LUNCH	

1:30 p.m.	Session No. 3	Topic: Human Factors
		Discussant: V. Basili (University of MD)
	"Transformations of Software Design and Code May Lead to Reduced Errors"	E. Connelly (PMA, Inc.)
	"You Can Observe a Lot by Just Watching How Designers Design"	E. Soloway (Yale)
	"Evaluating Multiple Coordinated Windows for Programmer Workstations"	C. Grantham (University of MD)
3:00 p.m.	BREAK	
3:30 p.m.	Session No. 4	Topic: Quality Assessment
		Discussant: W. Agresti (CSC)
	"Cleanroom Certification Model"	P. Currit (IBM)
	"Projecting Manpower to Attain Quality"	K. Rone (IBM)
	"An Approach to Software Baseline Generation"	J. Romeu (IITRI)
5:00 p.m.	ADJOURN	

SUMMARY OF THE SESSIONS: EIGHTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Prepared for the

NASA/GSFC

EIGHTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

by

Thomas A. Babst

COMPUTER SCIENCES CORPORATION

and

THE GODDARD SPACE FLIGHT CENTER

SOFTWARE ENGINEERING LABORATORY

INTRODUCTORY REMARKS

John J. Quann, Deputy Director, Goddard Space Flight Center (GSFC), made the opening remarks at GSFC's Eighth Annual Software Engineering Workshop. He stressed the importance of Software Engineering Laboratory (SEL) activities to GSFC and pointed out the effect of this work on the Spacelab project and its relevance to future projects such as the Space Telescope and Space Station.

The Space Station, for example, will require all NASA centers to work together in a disciplined manner. NASA will be studying the results of SEL research to identify strategies for the design, implementation, testing, and interfacing of the software system. Mr. Quann also emphasized the importance of conferences, such as this one, as opportunities for the exchange of ideas among managers, developers, and academicians. This is the route to excellence in the field of software engineering.

SESSION 1 - CURRENT RESEARCH IN THE SOFTWARE ENGINEERING LABORATORY

Frank McGarry--"The Software Engineering Laboratory"

Frank McGarry of GSFC summarized the efforts of the Software Engineering Laboratory over the past year. Mr. McGarry explained that the SEL is a consortium that also includes Computer Sciences Corporation and the University of Maryland. The SEL has concentrated its efforts in four major areas of software engineering research: software reliability and testing, technology evaluation, software measures, and software development management.

Many experiments have been performed by the SEL on production projects to evaluate software development technologies and to test software engineering theories. The results of some of these activities are being presented at this workshop. One of the principal areas of future activities will be the development of a software management environment to provide managers with the tools necessary to monitor and control the software development process.

Dave Card (Computer Sciences Corporation)--"Evaluating Software Engineering Technologies in the SEL"

Mr. Card's presentation described the results of a study that measured the effects of some software engineering practices, tools, and techniques on productivity and reliability in a production environment. The study was based on a sample of 22 similar software systems selected from the SEL data base. Eight widely used and accepted technologies were evaluated: quality assurance, software tools, documentation, code reading, top-down development, chief programmer team, structured coding, and design time. A statistical technique was employed to compensate for the effects of non-technological factors such as program- mer effectiveness and computer use.

The study concluded that none of the individual technologies evaluated had a significant effect on productivity during development; however, reliability was increased significantly by quality assurance, documentation, and code reading. A 30-percent improvement was achieved with these technologies, and other benefits may also be obtained. In particular, a reduction of maintenance costs seems probable.

In response to questions and comments from the audience, Mr. Card clarified the following points:

- All systems studied passed their acceptance tests, thus the quality of each was at least "good."
- The measure of programmer effectiveness used was a weighted measure of years of experience.
- Productivity was measured during development. That is, it is based on the cost to deliver the system to the customer. Subsequent maintenance costs are not included.

Victor Basili (University of Maryland)--"Dynamic Metrics for Software Management"

Dr. Basili's presentation described several efforts related to the development of a general methodology for monitoring software development for the early detection of problems. A pilot study, tool implementation, and extension activities were discussed.

The approach of the pilot study was to develop a series of baselines for critical measures. The actual values realized by a project under development can be compared with the baselines to detect significant deviations. A set of explanations was defined for each type of deviation, and the methodology provided a mechanism for rating the probability of these explanations. In the pilot study, data from eight projects formed the baseline, and one other project was compared with them.

Dr. Basili indicated that future plans include extending the methodology to include additional measures and developing a knowledge-based system incorporating this methodology. The system will be developed using KMS (a software system used in constructing knowledge-based systems) at the University of Maryland. Dr. Basili stressed that this system is not intended to replace a manager's expert judgment but rather to support it with a formal tool.

In response to questions and comments from the audience, Dr. Basili clarified the following points:

- Measurement can be extended to the whole life cycle, and this option is under study.
- The baselines are defined at discrete points corresponding to specific percents of work completed. In practice, it is difficult to determine the percent completion of a project under development.

The best way to do this can be determined only by studying the environment in which the methodology is to be applied.

- Rate of change can be used as an indicator but is not in the current methodology.
- Programmers in this environment do not appear to be changing their behavior to match the metrics.
- The KMS knowledge-based system may be transportable, but that is not an important consideration now.

Marvin Zelkowitz (University of Maryland)--"Characteristics of a Rapid Prototyping Experiment"

Dr. Zelkowitz discussed the issues of prototyping in the context of an actual prototype recently developed for GSFC. This prototype, the Flight Dynamics Analysis System (FDAS), is currently under evaluation.

FDAS is intended to provide an integrated software development environment for spacecraft attitude, orbit, and mission analysis research. It consists of a management system and a library of application software. The application software was implemented in an extended version of FORTRAN that provides data abstraction and generalized input/output capabilities.

Dr. Zelkowitz provided three definitions of a prototype: a "quick and dirty" throwaway, a partial implementation, and first build. Some portions of a quick and dirty prototype may be reused later in the final system. A prototype need not be cheap to be cost-effective if it enables the full system to be implemented less expensively and with greater reliability than it would have been without the prototype.

In response to questions and comments from the audience, Dr. Zelkowitz clarified the following points:

- The goal of FDAS was not to save code, although much will probably be reused.
- Forty-eight percent of the development effort was spent in implementation. This phase includes coding and unit testing activities.
- The full FDAS system may be implemented in a language other than FORTRAN.
- FDAS is in the public domain and will probably be made available through COSMIC when it is completed.

SESSION 2 - TESTING SOFTWARE

Jim Ramsey (University of Maryland)--"Structural Coverage of Functional Testing"

Mr. Ramsey described the initial results of an evaluation of the effectiveness of functional testing by examining structural coverage metrics. A FORTRAN program consisting of 68 subroutines was instrumented to produce structural coverage measures when executed. Then, structural coverage data were collected by performing (functional) acceptance tests. These results were compared with data from operational use of the program.

Mr. Ramsey reported that although the acceptance tests and operational use largely covered the same software, there were significant differences. Also, about one-third of the code was never executed. However, this procedure does have the potential for providing a numerical measure of the effectiveness of (functional) acceptance tests.

A much larger piece of software is now in the process of being instrumented and tested in this manner. More concrete results should be derived from this additional data.

In response to the questions and comments from the audience, Mr. Ramsey clarified the following points:

- Conclusions cannot be made at this time about whether larger or smaller modules are more fully exercised or about the nature of the untested code.
- The tests performed were derived from the functional requirements of the program, not from knowledge of the code.

Amrit Goel (Syracuse University)--"A Methodology for Detecting Errors"

Dr. Goel described a mathematical approach to selecting software tests. No testing strategy can detect all errors; however, Error Specific Tests (ESTs) can be devised to isolate those types of errors important to the tester.

In this approach, test requirements are formulated in algebraic notation. Tests are determined from the requirements specification and its functional decomposition. Next, tests specific to each type of error targeted by the user are developed and enumerated in a test plan. This process of defining functional requirements and structural parts may also provide insight to software complexity.

In response to questions and comments from the audience, Dr. Goel clarified the following points:

- The methodology discussed has not been tested on actual software development problems.
- Optimization of the test plan is necessary to avoid redundant tests.
- Automation is essential because of the complexity and comprehensiveness of the resulting test plan.
- This method of testing is different from program proofs, although the notation is similar.

Cathy Savolaine (Bell Laboratories)--"Testing and Error Analysis of a Real-Time Controller"

Ms. Savolaine reported the results of an error analysis based on data collected from the development and testing of a real-time communications controller system. The system studied was the Satellite Network Scheduler (SNS), which controls ground stations as part of a reservation system for picturephone conferencing. Testing for each release was performed by an individual not involved in the development of that release. The number of errors per module was correlated with module size and cyclomatic complexity. Errors were classified in three groups: omission, commission, and requirements. Half of the errors detected before delivery were errors of omission. In contrast, half of the errors found during operational usage were errors of commission.

Ms. Savolaine concluded from these results that complex modules should be avoided, more code inspections should be performed, and developers should look harder for commission errors because these were the principal type found by the user.

In response to questions and comments, Ms. Savolaine clarified the following points:

- Records were kept of the numbers of errors found during code inspection, but the data are not readily available.
- The development cost of an automated test package was included in the SNS development budget.
- Errors of commission were not further categorized, but this can be done.

- It is not known at this time why fatal errors seemed to cluster in the simpler modules.
- The total number of errors, not error rate, was compared with size and complexity.

SESSION 3 - HUMAN FACTORS

Ed Connelly (PMA)--"Transformation of Software Design and Code May Lead to Reduced Errors"

Mr. Connelly described a series of experiments conducted to determine how well people can use examples to specify logic. In this study, individuals were asked to devise solution algorithms to various problems (specifically, scheduling and allocation problems).

The problems were initially given to accountants, and later to programmers. The solution algorithms were fed to an inductive processor. Feedback from the processor helped to systematize the subjects' thinking. The solution algorithms were compared with FORTRAN programs, and both were tested for correctness.

Based on the results of these experiments, Mr. Connelly concluded that performance is correlated with the number of languages and operating systems the programmer is familiar with. He also indicated that the examples had fewer errors of commission than FORTRAN code developed for the same problem.

In response to questions and comments from the audience, Mr. Connelly clarified the following point:

- The dependent variable in the analysis was performance (i.e., the number of incorrect inputs recognized by the program).

Elliot Soloway (Yale University)--"You Can Observe a Lot by Just Watching" (How Designers Design)

Dr. Soloway described some observations made during a study of the work habits of novice and experienced software designers. The experts had 8 or more years of experience; the novices had 2 years or less; all were familiar with telecommunications system software.

Each individual was given the same vague set of specifications for an electronic mail system and was asked to develop a design. The design process was recorded on videotape. An interviewer prompted the designers to describe what steps they were taking. The experts approached the problem systematically in a top-down fashion. They kept detailed notes of assumptions, constraints, and expectations. In contrast, the novices immediately began working on the problem at a very detailed level.

One conclusion drawn by Dr. Soloway was that an effective design tool should provide a capability for keeping track of notes of the type made by the experts. Most such tools developed in the past have focused on what the designer should be doing rather than on facilitating what he/she actually does.

In response to questions and comments from the audience, Dr. Soloway clarified the following points:

- The expert designers were very individualistic.
- The experts seemed to have some familiarity with the problem. It would be interesting to test them in other circumstances.
- The experts were clearly designers, whereas the novices could have been programmers who were asked to design.

- The experts continued to "back up" if questions remained unanswered. It would be interesting to see and measure where this backup occurs.
- It might be possible to build a system to teach novices to become experts in design.
- The experts and novices were separated, but it might be interesting to see how they worked together.
- The experiment was exploratory in nature, rather than a rigorous test of any hypothesis.

Charles Grantham (University of Maryland)--"Evaluating Multiple Coordinated Windows for Programming Workstations"

Dr. Grantham described the results of some recent research on the design of a multiple-screen programmer workstation. Two such workstation designs are under evaluation. One station consists of three separate screens; the other consists of one screen with four windows. The information on each screen or window is coordinated with the others. The appropriate information to be displayed on each window was determined by observing the behavior of programmers while testing, debugging, and modifying software. The module specification, structure chart, and source listing are displayed under both configurations. The four-window configuration has an additional user-defined area. Ultimately, better workstation designs should improve the software development process by maximizing the number of tools that are available to the programmer at one time.

In response to questions and comments from the audience, Dr. Shneiderman and Dr. Grantham clarified the following points:

- Many multiple-screen systems do exist, but most are passive displays that do not have coordinated screen action. This study addresses dynamic screen coordination.
- Software maintenance will be facilitated by using multiple screens in this manner, because additional details about the module being maintained will be available.

- The importance of left/right orientation should be considered when selecting and arranging display contents.
- The layout of information in different screens or windows was essentially fixed (not dynamically controlled by the user).

SESSION 4 - QUALITY ASSESSMENT

Al Currit (IBM Corporation)--"Cleanroom Certification Model"

Mr. Currit described the software reliability model used for software certification in the "cleanroom" development approach. The cleanroom is a rigorous methodology that separates developers from all testing activities. It replaces unit and integration testing with rigorous code inspections. Although it is difficult to produce software with zero defects, it is hoped that this approach will produce code with a very low probability of failure.

Certification of the developed code is dependent on its achieving a specified mean time to failure (MTTF) during testing. MTTF is an appropriate measure because it is unambiguous and relates to the customer's needs. The certification model predicts MTTF based on failure data collected during testing. It shows good agreement with published data. Although mathematically similar to some popular reliability models, it is simpler than most. This MTTF model seems to be an effective tool for determining when software is ready for delivery.

In response to questions and comments from the audience, Mr. Currit clarified the following points:

- MTTF is measured in terms of usage months rather than CPU execution time.
- The cleanroom concept replaces unit testing with statistical testing. Test data are used to calculate MTTF.
- Under the cleanroom system, programmers are kept away from the computer as much as possible. They only get clean compiles of their code and are not able to debug programs on the computer.

Kyle Rone (IBM Corporation)--"Projecting Manpower To Attain Quality"

Mr. Rone described the derivation of a model to predict the manpower required to insert new technology into a system. This model will also aid in defining the distribution of manpower needed to achieve maximum quality.

The development environment studied generates software in increments, as a series of releases. The goal of this research effort is to create a model that matches this strategy. Increasing the manpower at the beginning of a project and moving more quality analysis toward the front seems to facilitate the early detection of errors. Mr. Rone believes that by following this plan, maintenance costs for the system studied, which annually are now approximately 25 percent of the development cost, will be reduced to around 15 or 20 percent.

In response to questions and comments from the audience, Mr. Rone clarified the following point:

- Maintenance includes the effort required to fix errors documented on discrepancy reports. It does not include the effort spent to complete change requests.

Jorge Romeu (ITT Research Institute)--"An Approach to Software Baseline Generation"

Dr. Romeu discussed the initial results of an ongoing research effort to define baselines for the management of software development. A baseline was defined to be an estimate of the usual value of any characteristic of a software system.

The analysis was based on data collected by the Software Engineering Laboratory. Correlations were calculated between effort and other software characteristics, and descriptive statistics were generated. The ultimate goal of this research is to develop guidelines for estimating costs and performance characteristics for software development based on historical data. The baseline approach is widely applicable and easily implemented.

D1

N84 23138

PANEL #1

CURRENT RESEARCH IN THE SOFTWARE ENGINEERING LABORATORY (SEL)

D. Card, Computer Sciences Corporation
V. Basili, University of Maryland
M. Zelkowitz, University of Maryland

EVALUATING SOFTWARE ENGINEERING TECHNOLOGIES IN THE SEL

David N. Card

COMPUTER SCIENCES CORPORATION

Frank E. McGarry

GODDARD SPACE FLIGHT CENTER

Gerald Page

COMPUTER SCIENCES CORPORATION

Prepared for the

NASA/GSFC

Eighth Annual Software Engineering Workshop

INTRODUCTION

The basic goal of software engineering is to produce the best possible software at the lowest possible cost. Many practices, tools, and techniques (collectively referred to as technologies) have been developed that purport to help do this, some of which have become widely accepted in the software industry. However, few of these technologies have been effectively evaluated experimentally (Reference 1). This is due in large part to an insufficient understanding of the software development process, a lack of recognized standards for measurement, and the prohibitive cost of large-scale controlled experiments. The analysis described in this paper addresses some of these issues. The specific objectives of this study were to

- Measure technology use in a production environment
- Develop a model for evaluating software engineering technologies
- Evaluate the effects on productivity and reliability of some specific technologies

Eight widely used technologies were selected for study, as identified in Table 1. The extent of general use shown in Table 1 is the percent of respondents reporting having successfully applied these technologies in a survey by Beck and Perkins (Reference 2).

The data analyzed in this study was collected by the Software Engineering Laboratory (SEL). The SEL has collected data from more than 45 projects during the past 6 years (Reference 3). Table 2 shows some of the characteristics of these projects. Although a controlled experiment was not performed for this study, a carefully matched sample was selected for analysis from the SEL data base. The sample

TABLE 1. TECHNOLOGY INDICES

INDEX	SEL MEDIAN (%)	GENERAL[1] USE (%)
QUALITY ASSURANCE[2]	49	49
TOOL USE[2]	49	NA
DOCUMENTATION[2]	82	78
STRUCTURED CODE	70	59
CODE READ	20	44
TOP-DOWN DEVELOPMENT	60	60
CHIEF PROGRAMMER	85	46
DESIGN TIME	32	NA

[1]FROM SURVEY BY BECK & PERKINS.

[2]COMPOSITE OF SEVERAL ITEMS.

TABLE 2. ENVIRONMENT STUDIED

TYPE OF SOFTWARE: SCIENTIFIC, GROUND-BASED, INTERACTIVE GRAPHIC, MODERATE RELIABILITY AND RESPONSE REQUIREMENTS

LANGUAGES: 85% FORTRAN, 15% ASSEMBLER MACROS

MACHINES: IBM S/360 AND 4341, BATCH WITH TSO

PROJECT CHARACTERISTICS:	AVERAGE	HIGH	LOW
DURATION (MONTHS)	15.6	20.5	12.9
EFFORT (STAFF-YEARS)	8.0	11.5	2.4
SIZE (1000 LOC)			
DEVELOPED	57.0	111.3	21.5
DELIVERED	62.0	112.0	32.8
STAFF (FULL-TIME EQUIV.)			
AVERAGE	5.4	6.0	1.9
PEAK	10.0	13.9	3.8
INDIVIDUALS	14	17	7
APPLICATION EXPERIENCE			
MANAGERS	5.8	6.5	5.0
TECHNICAL STAFF	4.0	5.0	2.9
OVERALL EXPERIENCE			
MANAGERS	10.0	14.0	8.4
TECHNICAL STAFF	8.5	11.0	7.0

SAMPLE: 22 SYSTEMS USING A VARIETY OF TECHNOLOGIES

438-CAR-(33)-5

consisted of 22 scientific software systems developed in FORTRAN on the same computers to support spacecraft flight dynamics applications.

TECHNOLOGY MEASUREMENT

A degree-of-use score (technology index) was determined for each of the technologies listed in Table 1 for every system in our sample. These scores are based on both subjective and objective information. (The table lists the median score from the sample of 22 projects.) These scores are the percentage of actual use of a technology relative to its maximum possible use. The exception is design time, which is simply the percentage of the development schedule spent in design.

For those technology indices having only one component (see Table 1), such as code reading, the score is the percentage of code to which this technology was applied. For those technology indices having more than one component, such as documentation, the score is the percentage of components applied. In the case of the documentation technology index, the score is the percentage of documents actually produced by a project of those that might be produced in this environment.

This analysis attempted to identify the effects of technology use on development team productivity and software reliability. Productivity was measured in terms of the number of noncomment lines of code designed, coded, and tested per programmer hour of effort. Reliability was measured as the inverse of the number of errors detected per noncomment line of code.

One assumption made in this analysis is that the effect of any technology is incremental. That is, a high level of use of a beneficial technology has more effect than a low level of use. A technology that is of no value unless applied perfectly is of no value at all, because it will never be applied perfectly.

TECHNOLOGY EVALUATION

Evaluating the effect of a technology on an actual software development project is not easy. In practice, several technologies may be applied together, and other factors such as programmer effectiveness and problem complexity also influence project results. Boehm (Reference 4) has pointed out the difficulty of distinguishing the effects of modern programming practices from those of related factors. Table 3 lists the nontechnology factors considered in this analysis. All of these have been suggested in the software engineering literature to affect productivity and/or reliability.

Thus, the next step of this analysis was to identify the major nontechnology factors and to develop a procedure for compensating for their effects on productivity and reliability. The analysis of covariance technique (Reference 5) was selected to deal with this situation. The Statistical Analysis System (Reference 6) software performed the computations reported in this paper.

The technology indices were collapsed for this analysis by dividing the projects into "high" and "low" groups with respect to each technology index. Although this causes some loss of information, the resulting analysis is also more robust. This analytic technique permitted tests of significance to be performed between the high and low groups with respect to productivity and reliability after compensating for the nontechnology factors (covariates).

The two most highly correlated factors from Table 3 were initially selected as covariates for productivity and reliability. Programmer effectiveness and computer use were selected as covariates with productivity. Programmer effectiveness was also selected as a covariate with reliability. However, because requirements changes was cor-

TABLE 3. OTHER FACTORS

FACTOR	MEAN	CORRELATIONS	
		PRODUCTIVITY[1]	RELIABILITY[2]
PRODUCTIVITY	3.0	–	0.51
PROGRAMMER EFFECTIVENESS (WEIGHTED YEARS)	5.7	0.53+	0.68*
REQUIREMENTS CHANGES/ SUBSYSTEMS	1.4	-0.12	-0.40
NUMBER OF SUBSYSTEMS	6	0.21	0.03
NUMBER OF DATA SETS	11	0.26	0.17
NUMBER OF DATA ITEMS	328	0.30	0.21
AVERAGE STAFF LEVEL (FTE)	3.3	0.10	-0.09
AVERAGE MODULE SIZE (NEW)	193	-0.07	-0.15
COMPUTER USE (HOURS/LOC)	0.008	-0.59*	-0.19
MANAGEMENT/SUPPORT EFFORT (%)	19	-0.47	-0.18
DATA DENSITY (DATA ITEMS/ SUBSYSTEM)	71	-0.07	0.38+

[1]PRODUCTIVITY = DEVELOPED NONCOMMENT LINES OF CODE/PROGRAMMER HOURS

[2]RELIABILITY = -ERRORS/DEVELOPED NONCOMMENT LINES OF CODE

+SECOND FACTOR SELECTED.

*FIRST FACTOR SELECTED.

436-CAR-(6f)

related with programmer effectiveness, data density was substituted as the second covariate for reliability. This prevented collinearity in the model.

Each technology was evaluated independently in this manner. One potential confounding effect recognized in an earlier SEL study (Reference 7) and by Boehm (Reference 4) was the tendency of technologies to be used together. This makes it difficult to isolate the effects of one technology from another and poses the possibility that there might be an interaction of technologies that this procedure could not detect.

Productivity Results

This approach to the evaluation of technologies resulted in the generation of a class of models (one for each technology) of the form

$$\text{Productivity} = \text{Technology} + \text{Programmer Effectiveness} + \text{Computer Use}$$

Together, programmer effectiveness and computer use accounted for 54 percent of the variation in productivity before the effects of any technologies were included in the models. Table 4 shows the additional variation accounted for by the technology factors. The magnitude and significance of the effect for each technology are also listed in the table. Individually, none of the technologies studied in this analysis showed a significant effect on productivity. However, this also indicates that any other benefits derived from these technologies are not at the expense of productivity.

Early suggestions were that the principal value of modern programming practices is primarily in the area of maintainability. Shephard (Reference 8) indicated that the effects of such technologies are more apparent in less experienced

TABLE 4. SUMMARY OF PRODUCTIVITY ANALYSES

TECHNOLOGY INDEX (EFFECT)	SIGNIFICANCE OF EFFECT (X_1)	PERCENT IMPROVEMENT	EXPLANATORY CONTRIBUTION (X_2)
QUALITY ASSURANCE	0.87	-2	0
TOOL USE	0.77	3	0
DOCUMENTATION	0.36	11	2
STRUCTURED CODE	0.82	-2	0
TOP-DOWN DEVELOPMENT	0.95	-1	0
CODE READ	0.45	8	1
CHIEF PROGRAMMER	0.16	-16	5
DESIGN TIME	0.60	7	1

ISOLATED TECHNOLOGIES HAVE NO DETECTABLE EFFECT ON PRODUCTIVITY

436-CAR-(6b)

programmers than in experienced personnel such as those studied by the SEL (see Table 2). Some other environment-specific considerations are discussed in the summary at the end of this section. Mills (Reference 9) proposed that productivity is a byproduct of quality, that is, a consequence of minimizing rework (errors). We would thus expect differences in reliability (quality) to be easier to detect.

Reliability Results

This approach to the evaluation of technologies resulted in the generation of a class of models (one for each technology) of the form

$$\text{Reliability} = \text{Technology} + \text{Programmer Effectiveness} + \text{Data Density}$$

Together, programmer effectiveness and data density accounted for 63 percent of the variation in reliability before the effects of any technologies were included in the models. Table 5 shows the additional variation accounted for by the technology factors. The magnitude and significance of the effect for each technology are also listed in the table.

Three of the technologies studied in this analysis showed significant effects on reliability: quality assurance, documentation, and code reading. All of these techniques are examples of conscious efforts to understand and verify the software product. Approximately 73 percent of the variation in reliability can be explained with a model of this type. Improvements in reliability were obtained without any apparent effect on productivity (Table 4). Furthermore, this implies that skimping on these activities will not produce any cost savings for the developer.

TABLE 5. SUMMARY OF RELIABILITY ANALYSES

TECHNOLOGY INDEX (EFFECT)	SIGNIFICANCE OF EFFECT (X_1)	PERCENT IMPROVEMENT	EXPLANATORY CONTRIBUTION (X_2)
QUALITY ASSURANCE	0.02*	29	10
TOOL USE	0.78	3	1
DOCUMENTATION	0.04*	27	8
STRUCTURED CODE	0.75	3	1
TOP-DOWN DEVELOPMENT	0.67	6	1
CODE READ	0.02*	29	10
CHIEF PROGRAMMER	0.56	8	1
DESIGN TIME	0.96	–1	0

*$P < 0.05$

436-CAR-(6b)

Summary

The numerical results just presented must be considered in the context of the local software development environment. The results for each technology are discussed below.

- Quality Assurance--A program of regular reviews (e.g., system requirements, preliminary design) improves software reliability at little or no additional cost in developers' time. Time spent on reviews is retrieved by avoiding subsequent problems.

- Software Tool Use--Extensive computer use in general seems to have a negative effect on productivity, although some specific tools may facilitate specific tasks. This index is based on the tools available in the flight dynamics environment. None of these tools has a demonstrable effect on productivity or reliability.

- Documentation--The development of effective documentation requires a careful review of the product under development. Documentation is, to some extent, a prerequisite for quality assurance reviews, and thus has a significant favorable effect on software reliability.

- Structured Code--The use of structured code produced no significant effect on productivity or reliability. However, the benefits of this technique are expected to occur in maintenance.

- Top-Down Development--The high-level designs of all of the systems in the sample studied were similar, and a substantial amount of code was reused from previous systems. Hence, it is not surprising that no benefit was identified from the use of top-down development in this environment.

- Code Reading--The simple practice of code reading improves software reliability at little or no additional cost in developers' time.

- Chief Programmer--The use of a chief programmer team produced no significant effect on productivity or reliability. However, it may provide other benefits.

- Design Time--The percent of schedule spent in design showed no significant effect on productivity or reliability. The high-level designs of all systems studied were similar, and the software development problem was well understood. In this situation, additional design time may not improve the product.

CONCLUSIONS

The analysis results presented in the preceding section lead to two types of conclusions: those pertaining to the conduct of software development in the local (SEL) environment, and those of a more general nature. For the most part, these conclusions are consistent with similar work by other researchers and with assumptions commonly accepted in the software development community.

The Local Environment

The results of this analysis provide the following suggestions for the conduct of flight dynamics software development projects:

- Use a small team of appropriately experienced individuals
- Do not depend on the computer to do the programmer's thinking
- Read all code developed
- Effectively document each phase of development
- Conduct regular quality assurance reviews

The most important lessons are that developers must be capable and must consciously seek quality. These conclusions will be fed back into the management of subsequent software development projects at Goddard Space Flight Center (GSFC).

General Implications

The analytic procedure and some results of this study are applicable to more than just the GSFC flight dynamics environment. The general conclusions of the study are as follows:

- Technology use can be measured and evaluated in a production environment.

- A model that explains much of the variation in productivity and reliability was developed for technology evaluation.
- Limited use of the technologies studied can produce up to about a 30-percent improvement.

Although the improvements identified in this study were in the area of reliability, a corresponding decrease in maintenance cost due to a smaller need for error correction should also be realized. Furthermore, productivity appears to be a companion of quality software development. In addition, some technologies may produce other beneficial effects in areas not yet studied by the SEL.

The analysis of covariance model appears to be one appropriate technique for evaluating the effects of technologies in this context. However, small improvements in productivity and/or reliability that were not detected by this procedure might occur. More such evaluation efforts are needed to provide an empirical basis for the formulation of software development standards.

ACKNOWLEDGMENT

The authors would like to thank V. Basili, B. Curtis, S. Zweben, and W. Agresti for their comments on an earlier version of this paper.

REFERENCES

1. B. A. Sheil, "The Psychological Study of Programming," ACM Computing Surveys, vol. 13, no. 1, March 1981

2. L. L. Beck and T. E. Perkins, "A Survey of Software Engineering Practice: Tools, Methods, and Results," IEEE Transactions on Software Engineering, vol. 9, no. 5, September 1983

3. Software Engineering Laboratory, SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

4. B. W. Boehm, Software Engineering Economics. New York: Prentice Hall, 1981, pp. 453-456

5. O. J. Dunn and V. A. Clark, Applied Statistics: Analysis of Variance and Regression. New York: John Wiley & Sons, 1974, pp. 307-332

6. SAS Institute, Statistical Analysis System User's Guide, J. T. Helwig and K. A. Council, December 1979

7. D. N. Card, "Identification and Evaluation of Software Measures," Proceedings of the Sixth Annual Software Engineering Workshop, December 1981

8. S. B. Shephard, B. Curtis, P. Milliman, et al., "Modern Coding Practices and Programmer Performance," IEEE Computer, vol. 12, no. 12, December 1979

9. H. D. Mills, "Software Productivity in the Enterprise," Software Productivity. New York: Little, Brown & Co., 1983, pp. 265-270

D2

ORIGINAL PAGE IS
OF POOR QUALITY

N84 23139

MONITORING SOFTWARE DEVELOPMENT THROUGH DYNAMIC VARIABLES

Carl W. Doerflinger
Victor R. Basili

University of Maryland
Dept. of Computer Science
College Park, MD 20742
(301) 454-2002

Abstract

This paper describes research conducted by the Software Engineering Laboratory (SEL) on the use of dynamic variables as a tool to monitor software development. The intent of the project is to identify project independent measures which may be used in a management tool for monitoring software development. This study examines several FORTRAN projects with similar profiles. The staff was experienced in developing these types of projects. The projects developed serve similar functions. Because these projects are similar we believe some underlying relationships exist that are invariant between the projects. These relationships, once well defined, may be used to compare the development of different projects to determine whether they are evolving the same way previous projects in this environment evolved.

Overview

The Software Engineering Laboratory (SEL) is a joint effort between the National Aeronautics and Space Administration (NASA), the Computer Sciences Corporation (CSC), and the University of Maryland established to study the software development process. To this end, data has been collected for the last six years. The data was from attitude determination and control software developed by CSC, in FORTRAN, for NASA. Additional information on the SEL, the data collection effort, and some of the studies that have been made may be found in papers from the Software Engineering Laboratory Series published by the SEL [1,2,3]

This research was supported by the National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland. Computer support provided in part by the facilities of NASA/Goddard Space Flight Center.

The interest in the software development process is motivated by a desire to predict costs and quality of projects being planned and developed. For several years, studies have examined the relationships between variables such as effort, size, lines of code, and documentation [4,5]. These studies, for the most part, used data collected at the end of past projects to predict the behavior of similar projects in the future. In 1981 the SEL concluded that many of these factors were too dependent on the environment to be useful for the models that had been developed [6]. Any model which attempts to trace these relationships should therefore be calibrated to the environment being examined. The meta-model proposed by the SEL is designed for such flexibility [6].

Another way to isolate out the environment dependent factors is by comparing two internal factors of a project, thus ignoring all outside influences. One approach that is used to monitor software development examines the time gap between the initial report of software problems and the complete resolution of the problem [7]. Comparing two variables is useful because it also accentuates problem areas as they develop, providing relative information rather than absolute information. Relative information is useful to the project manager because it accentuates trends as the project develops. If project environments are similar, then similar values should be expected. Because the project environments in the SEL are similar, it was felt that this approach could be further extended to provide managers with information about how a set of variables over the course of a project differed from the same set of variables on other projects (baselines). The managers could be alerted to potential problems and use other variable data and project knowledge to determine whether the project was in trouble.

This methodology is flexible enough to respond to changing needs. Every time a project is completed the measures collected during its development may be added in to calculate a new baseline. In this changes in the environment, as they occur.

Baselines might also be developed to reflect different attributes. For instance, several projects which had good productivity might be grouped to form a productivity baseline. Once baselines are established, projects in progress may be compared against them. All measures falling outside the predetermined tolerance range are interpreted by the manager.

Methodology

The implementation of this methodology is dependent on two factors. The first factor is the availability of measures that are project independent and can also be collected throughout a project's development. Variables like programmer hours and number of computer runs are project dependent. By comparing these variables against each other a set of relative measures may be generated which is project independent. For instance, the number of software changes may vary from project to project. The project dependent features shared by each variable will cancel out when the ratio of software changes per computer run is taken. The resulting relative measure is project independent.

The second factor is the need for fixed time intervals common to all projects. To normalize for time, project milestones were used. The time into a project might be twenty percent into coding instead of ten weeks into the project, for instance.

When computing the baselines one other factor was considered. At any given interval during development a variable may measure either the total number of events that have occurred from the beginning of development (cumulative) or the number of of events that have occurred since the last measured interval (discrete). Since these approaches may convey different information it was felt that they both should be used.

For simplicity, the baseline for each relative measure was defined as the average and standard deviation computed for the measure at predetermined intervals. A project's progress may now be charted by the software manager. At each interval in a projects development the relative measures are compared with their respective baseline. Any measures outside a standard deviation are flagged. These measures are then interpreted by the project manager to determine how the project is progressing. A flagged measure may indicate a project is developing exceptionally well or it may indicate a problem has been encountered.

The interpretation of a set of flagged measures is a three step process. First, the manager must determine the possible interpretations for each flagged relative measure using lists of possible interpretations developed and verified based on past projects.

Second, the union of the lists of possible interpretations of each flagged measure must be taken. The list formed by this union contains all the possible interpretations ordered using the number of times each interpretation is repeated in the different lists. The larger the number of overlaps a possible interpretation has, the greater the probability it is the correct interpretation.

Third, the manager must analyze the combined list and determine if a problem exists. Interpretations with an equal number of overlaps all have an equal probability of being the correct interpretation. If none of the possible interpretations for a given relative measure overlap then the relative measure should be considered separately.

When analyzing the interpretations, three pieces of information must be considered; the measurements, the point in development, and the managers knowledge of the project. A relative measure may indicate different things depending on the stage of development. For instance, a large amount of computer time per computer run early in the project may indicate not enough unit testing is being done. Personal knowledge may also give valuable insight.

A fundamental assumption for using this methodology is that similar type projects evolve similarly. If a different type of project was compared to this database, the manager would have to decide whether the baselines were applicable. Depending on the type of differences, the established baselines may or may not be of any value.

EXAMPLE 1:

Forty percent into coding a software manager finds that the lines of source code per software change is higher than normal. A list previously developed is examined to determine what the relative easure might indicate. The possible interpretations for a large number of lines of source code per software change might be:

- good code
- easily developed code
- influx of transported code
- near build or milestone date
- computer problems
- poor testing approach

If this were the only flagged measure the manager would then investigate each of the possibilities. If the value for the measure is close to the norm less concern is needed than if the value is further away.

If in addition to lines of source code per software change the number of computer runs per software change was higher than normal, the manager would also examine this measure. The possible interpretations for a large number of computer runs per software change might be:

- good code
- lots of testing
- change backlog
- poor testing approach

The union of the possible interpretations of these two measures indicates that the strongest possible interpretations are 1) good code and 2) a poor testing approach. The number of possibilities to investigate is smaller because these are the only measures which overlap. The manager must now examine the testing plan and decide whether either of these interpretations reflect what is actually occurring in the project. If these two possible interpretations do not reflect what is happening on the project, the manager would then examine the other interpretations.

Baseline Development

To develop a baseline one must first have variables whose measurements were taken weekly for several projects. Five variables in the SEL database were used. The lines of source code, number of software changes, and number of computer runs were collected on the growth history form. The amount of computer time and programmer hours were collected on the resource summary form. Measurement of these variables started near the beginning of coding. In this study, nine separate projects were examined whose development was documented, with sufficient data, in the SEL database. The projects ranged in size from 51-112K lines of source code with an average of 75K. No examination was done for the requirements or design phases.

Once the variables were chosen the average and standard deviation was computed for each baseline. Some baselines suffered from limited data points during the beginning of the coding phase. A couple of the projects, in which problems were known to have existed, were flagged as soon as data on these projects appeared, but this was fifty percent of the way into coding. It is not known how much earlier they would have appeared, if data existed at the early intervals.

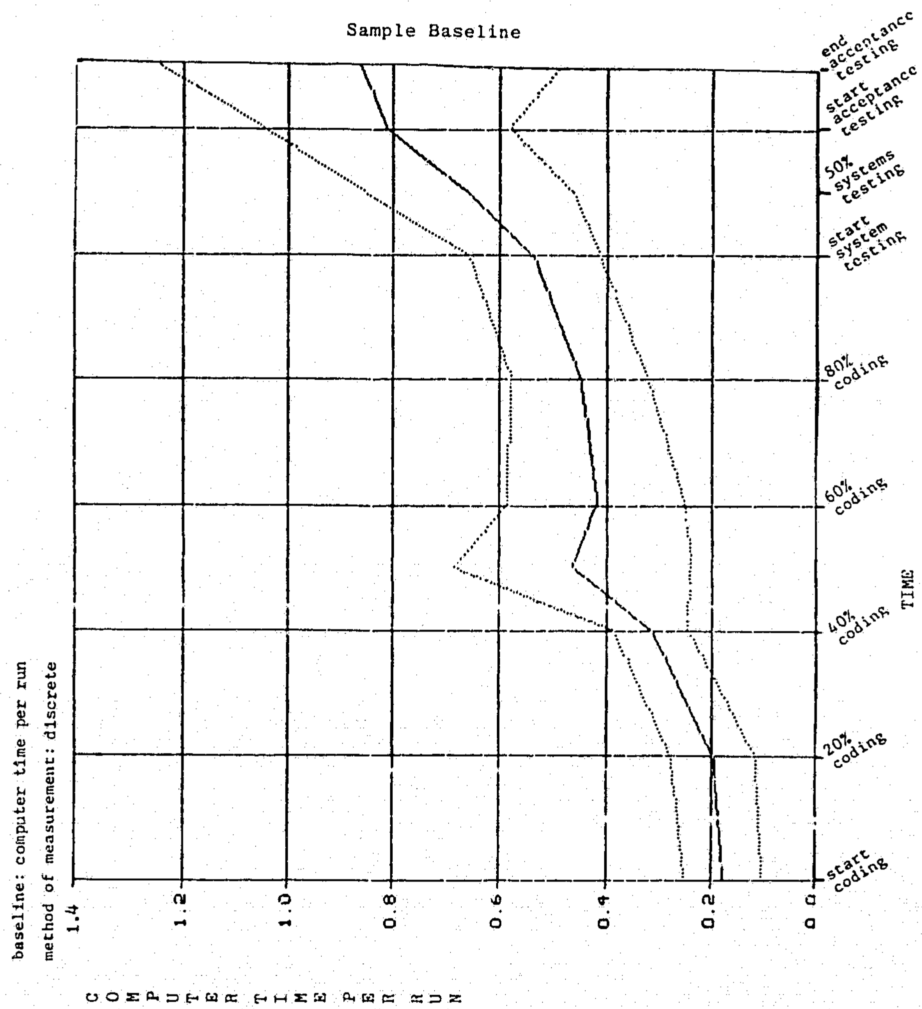
Interpretation of Relative Measures

Once a set of baselines are established new projects may be compared to them and potential problems flagged. To interpret these flagged relative measures a list should be developed with each measures possible interpretations. Each list must consider the possible interpretations of the relative measure when it is either above normal or below normal. What each component variable actually measures should also be considered when the different lists are developed.

A list was developed with possible interpretations for each relative measure being examined in the context of the SEL environment. In another environment the interpretation of these measures might be different. These lists are subdivided into two categories; above and below normal. The above normal category contains possible interpretations for the relative measure when it is outside one standard deviation from the average in the positive direction. The below normal category refers to interpretations when the measure is outside one standard deviation from the mean in the negative direction.

One of the reasons this methodology works is because of the implicit interdependencies between different relative measures. To show these interdependencies more explicitly a cross reference chart has also been provided for each interpretation to indicate other relative measures that can have the same interpretation. A number in the cross reference section indicates the list number of a relative measure that can have the same interpretation. The position of the list number in the 4-quadrant cross reference section indicates whether both interpretations are found with above normal values, both with below normal values, or one with above and the other with below normal values.

With these lists a set of flagged relative measures may be evaluated. When a relative measure is flagged, its associated list is examined for possible interpretations. Overlaps of this list with the lists of other flagged relative


Sample Baseline
baseline: computer time per run
method of measurement: discrete
COMPUTER TIME PER RUN
TIME
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
start coding
20% coding
40% coding
60% coding
80% coding
start system testing
50% systems testing
start acceptance testing
end acceptance testing

Relative Measures Examined:

- List 1 - Computer Runs per Line of Source Code
- List 2 - Computer Time per Line of Source Code
- List 3 - Software Changes per Line of Source Code
- List 4 - Programmer Hours per Line of Source Code
- List 5 - Computer Time per Computer Run
- List 6 - Software Changes per Computer Run
- List 7 - Programmer Hours per Computer Run
- List 8 - Computer Time per Software Change
- List 9 - Programmer Hours per Software Change

List 1: Computer Runs per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-low productivity	2 4	
	-high complexity	2 4 7 8 9	
	-lots of testing	2	6 7
	-removal of code (testing or transported)	2 3 4	
	-bad specifications	2 3 4	
below normal			
	-influx of transported code		2 3 4
	-near build or milestone date	6	2 3 4 8 9
	-little on line testing being done		2
	-little executable code being developed		2
	-computer problems		3

List 2: Computer Time per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-high complexity	1 4 7 8 9	
	-low productivity	1 4	
	-bad specifications	1 3 4	
	-lots of testing	1	6 7
	-unit testing being done	8	5
	-code being removed (testing or transported)	1 3 4	
below normal			
	-influx of transported code		1 3 4
	-near build or milestone date	6	1 3 4 8 9
	-little on line testing being done		1
	-code error prone	3 4 5 6	7 8 9
	-little executable code being written		1

List 3: Software Changes per Line of Source Code

type	interpretation	cross reference above normal	cross reference below normal
above normal			
	-good testing	6	8 9
	-error prone code	4 5 6	2 7 8 9
	-bad specifications	1 2 4	
	-code being removed (testing or transported)	1 2 4	
below normal			
	-influx of transported code		1 2 4
	-near build or milestone date	6	1 2 4 7 8
	-good code	8 9	6
	-poor testing program	8 9	6
	-change backlog		6
	-low complexity		4
	-computer problems		1

List 4: Programmer Hours per Line of Source Code

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-high complexity	1 2 7 8 9	
	-error prone code	3 5 6	2 7 8 9
	-bad specifications	1 2 3	
	-code being removed (testing or transported)	1 2 3	
	-changes hard to isolate	7 8 9	
	-changes hard to make	7 9	
	-low productivity	1 2	
below normal			
	-influx of transported code		1 2 3
	-near build or milestone date	6	1 2 3 8 9
	-low complexity		3

List 5: Computer Time per Computer Run

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-system & integration testing started early	6	
	-error prone code	3 4 6	2 7 8 9
	-compute bound algorithms being tested	8	
below normal			
	-unit testing going on	2 8	
	-easy errors being found		7 9

List 6: Software Changes per Computer Run

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-good testing	3	8 9
	-system & integration testing started early	5	
	-error prone code	3 4 5	2 7 8 9
	-near build or milestone date		1 2 3 4 8 9
below normal			
	-good code	3 8 9	
	-lots of testing	1 2	7
	-poor testing program	3 8 9	
	-change backlog	3	

List 7: Programmer Hours per Computer Run

type	interpretation	cross reference	
		above normal	below normal
above normal			
	-high complexity	1 2 4 8 9	
	-modifications being made to recently transported code		9
	-changes hard to isolate	4 8 9	
	-changes hard to make	4 9	
below normal			
	-easy errors being fixed		5 9
	-error prone code	3 4 5 6	2 8 9
	-lots of testing	1 2	6

List 8: Computer Time per Software Change

type	Interpretation	cross reference above normal	cross reference below normal
above normal	-good code	3 9	6
	-poor testing program	3 9	6
	-high complexity	1 2 4 7 9	
	-changes hard to isolate	4 7 9	
	-unit testing	2	5
	-compute bound algorithms being tested	5	
below normal	-near build or milestone date	6	1 2 3 4 9
	-good testing	3 6	9
	-error prone code	3 4 5 6	2 7 9

List 9: Programmer Hours per Software Change

type	Interpretation	cross reference above normal	cross reference below normal
above normal	-good code	3 8	6
	-poor testing program	3 8	6
	-changes hard to isolate	4 7 8	
	-changes hard to make	4 7	
below normal	-good testing	3 6	8
	-near build or milestone date	6	1 2 3 4 8
	-easy changes		5 7
	-transported code being modified	7	
	-error prone code	3 4 5 6	2 7 8

measures form the new list of what these relative measures together might indicate. The more overlaps a particular interpretation has, the greater the chance it is the correct interpretation. Interpretations with the same number of overlaps must be considered equally. The more relative measures flagged the more serious the problem may be. It is up to the manager to determine whether the deviation is good or bad.

Monitoring a Software Project's Development

Once the baselines have been developed and the lists of possible interpretations have been put together a software manager may monitor the actual development of a project. Example 1 demonstrated how a single interval may be interpreted. The following discussion will trace the development of an actual project. During the actual use of this methodology, influence would be exerted to correct problems as soon as they are identified. With this study, we must be content to study a projects evolution, without hindrance, and see at what points problems could of been detected.

Project twenty* was chosen for this examination because data existed throughout the projects development. In most respects project twenty was an average project. The project did have a lower than normal productivity rate. The lower rate may be partially explained by the fact the management was less experienced when compared to other projects. The project also suffered from some delayed staffing. Changes in staffing will be noted when the different time intervals are discussed.

The tables on the following page show which relative measures were flagged when project twenty was compared to the baselines for each stage of development. The numerical values represent how many standard deviations each flagged relative measure was from the baseline. The baseline for each relative measure was calculated using all nine projects.

Start of Coding:

At the start of coding only one relative measure is flagged. The smaller than normal number of software changes per line of source code using the discrete approach reflects work done during the design phase. The lists designed in the previous section were directed towards code production and testing and do not apply to this time interval when using the discrete approach. This measure may indicate good specifications or lots of PDL being generated. The manager might want to examine this measure later if it constantly repeated. Since it is the only measure flagged at this time it will be ignored.

---

* The numbering convention used is an extension of the one first used by Bailey and Basili [6].

project: 20

method of measurement: cumulative

number of standard deviations from norm: start code	20% code	40% code	50% code	60% code	80% code	start sys	50% sys	start accpt	end	relative measures
									1.3	>1 SD programmer hours/lines of source
						1.1				>1 SD runs/lines of source
						1.8	1.5	1.2		>1 SD computer time/lines of source
		1.1	1.2	1.1		1.1				<1 SD programmer hours/run

method of measurement: discrete

number of standard deviations from norm: start code	20% code	40% code	50% code	60% code	80% code	start sys	50% sys	start accpt	end	relative measures
	1.0	1.1	1.8			1.5		2.0	2.4	>1 SD programmer hours/lines of source
	1.2		1.8			1.8		1.7		>1 SD runs/lines of source
1.1										<1 SD changes/lines of source
	1.1		1.1			2.0		2.0	2.4	>1 SD changes/lines of source
	1.2	1.3	1.7			2.1		2.0		>1 SD computer time/lines of source
	1.2									<1 SD programmer hours/run
						1.2				>1 SD computer time/change

ORIGINAL PAGE IS OF POOR QUALITY

20% Coding:

The flagged relative measures found using the discrete approach at this point represent the work done from the start of coding until twenty percent of the way through coding. The list of possible interpretations for the flagged relative measures, generated from the lists made previously for the individual relative measure, would look like:

# overlaps	interpretation
3	bad specifications
3	code removed
2	low productivity
2	high complexity
2	error prone code
1	lots of testing
1	good testing
	changes hard to isolate
	changes hard to make
	unit testing being done
	easy errors being found

The strongest interpretations are bad specifications and code being removed. If the actual history is examined one finds that during this period there were a lot of specifications being changed. This resulted in code which was to be modified being discarded and new code being written. During the early period lots of PDL was being produced but very little new executable code. The list of possible interpretations does show that low productivity is also a strong possibility.

40% Coding:

The flagged relative measures which appear using the cumulative approach, from this time period on, are stronger indicators than the ones used in the first couple of intervals because the average is computed using more data points. The use of the discrete approach for the interval of twenty to forty percent is still dependent on three data points. The list of possible interpretations for this time period is:

# overlaps	interpretation
1	low productivity
1	high complexity
1	error prone code
1	bad specifications
1	code being removed
	changes hard to isolate
	changes hard to make
	lots of testing
	unit testing being done
	good testing
	easy errors

The number of possibilities is larger with this set of possible interpretations. Five interpretations are slightly stronger than the others. During the actual development, the first release of the project was made. The amount of code actually written was also lower than normal during this period. The use of the discrete approach gives a stronger feeling that code is not being written. Transported code tends to be installed in large blocks which can be isolated using the discrete approach.

50% Coding:

The relative measures flagged during this period are the same as the ones flagged at the twenty percent coding interval. The deviation from the norm for this interval is larger. The larger deviation may indicate a more serious problem. The problem may of been just as serious earlier but without the extra data points, that are now available, it could not be determined. The possible interpretations may be taken from the list developed earlier. Bad specifications and code removal were not factors during this period. The next three highest priority interpretations were; high complexity, error prone code, and low productivity. In addition to this the manager should be concerned with the continued appearance of the relative measure, programmer hours per computer run, as seen using the cumulative approach. This may indicate a lot of testing going on. This in conjunction with error prone code as a possible interpretation may indicate trouble. During actual development this period was spent developing code for the second release. The project manager felt that code was still not being developed quickly enough during this period.

60% Coding:

Only one relative measure is shown at this interval. The number of programmer hours per computer run using the cumulative approach is lower than normal for the third consecutive time. This should concern the manager because when examining the list for this measure one finds:

error prone code
lots of testing
easy errors being fixed

Since the occurrence of this measure is persistent it may indicate that the problem was corrected but not enough effort was expended to completely compensate for the past problems. It might also indicate the problem still exists. During the actual project it was found that while a lot of code was written, it had not been throughly tested. Release two was made during this period which could explain a heavy test load. Two additional staff members were added to the project during this phase to aid in coding and testing.

<u>80% Coding</u>:

The eighty percent coding interval does not show any measures outside the normal bounds. The addition of two staff members during the sixty percent coding phase, as well as the addition of a senior staff member during this phase, appears to have adjusted the project back along the lines of normal development. To fully compensate for the earlier problems one might expect some of the measures to swing in the other direction away from the average. The fact this over correction did not occur might explain the problems encountered in the next section.

<u>Start of System and Integration Testing</u>:

The flagged relative measures at this time period reflect the build up of effort for the third and final release. The list of possible interpretations for the collective set of flagged measures looks like:

# overlaps	interpretation
3	high complexity
3	bad specifications
3	code being removed
2	error prone code
2	low productivity
2	lots of testing
1	changes hard to isolate
1	unit testing being done
1	good code
1	poor testing
	changes hard to make
	good testing
	compute bound algorithms being run
	easy errors being fixed

Since the code did have a past history of poor testing an unusually large build up of testing should be expected. The two interpretations that apply most to this situation are lots of testing and error prone code.

<u>50% System and Integration Testing</u>:

Only one relative measure is flagged at this interval. This measure was flagged using the cumulative approach. An examination of the measure at the previous interval shows a very high value. A slow drop off from this high measure is to be expected when using the cumulative approach. An examination of possible interpretations that would apply for this period of development include:

high complexity
lots of testing
unit testing being done
testing code being removed

A lot of testing is certainly indicated by past history.

<u>Start Acceptance Testing</u>:

The relative measures flagged at this interval reflects the build up in testing before the start of acceptance testing. The list of possible interpretations looks like:

# overlaps	interpretation
3	bad specifications
3	code being removed
2	high complexity
2	low productivity
1	error prone code
1	lots of testing
	changes hard to isolate
	changes hard to make
	unit testing being done
	good testing

Since little code was being developed during the testing period, a large amount of testing with errors being found is the most reasonable interpretation of these flagged measures. The early history of poor testing may be seen here with errors being uncovered late.

<u>End Acceptance Testing</u>:

The two flagged relative measures at the end of acceptance testing reflect the clean up effort being made on the code. An average amount of computer time and an average number of computer runs indicates that the acceptance testing is going well. The project was behind schedule due to the earlier problems encountered. Clean up was done during the acceptance testing phase in an attempt to get the project out the door as soon as possible.

As seen in this example, the problems that occur during a projects development are reflected in the values calculated for the relative measures. The methodology preposed can be used to monitor projects. The number of possible interpretations increases with each new flagged relative measure. The ordering of the measures by

ORIGINAL PAGE IS
OF POOR QUALITY

the number of overlaps provides an easy method of sorting the possible interpretations by priority. Another method of sorting the possible interpretations could include a factor that considers both the number of overlaps and the probability of a given interpretation being the cause at a given interval. The weighting of interpretations for a given interval could be calculated using the pattern of occurrence of the different interpretations which have appeared during the same interval in past projects.

An Alternate Approach

Flagged relative measures might also be interpreted using a decision support system. The data for the various relative measures would be stored in a knowledge base along with a set of production rules. To evaluate a project the values for each relative measure would be entered into the system. The knowledge base would compare the relative measures to their respective baselines, determine which relative measures were outside the norm, and interpret these relative measures using the production rules. A list of possible interpretations ordered by probability would be generated as a result.

The difference between a decision support system and the approach presented in this paper is the method of interpreting the flagged relative measures. Each production rule in the decision support system is the logical disjunction of several flagged measures which yields a given interpretation. Each production rule is assigned a confidence rating which is then used to rate the possible interpretations. The lists for the relative measures provided earlier in the paper may be easily converted to production rules using the cross reference section. To develop the production rules for an interpretation one must generate the various combinations of relative measures which might reasonably imply the interpretation. Some relative measures may not imply a particular interpretation unless they are found in conjunction with another relative measure. Once the production rules are known and a knowledge base constructed a decision support system may be built. For an example of a domain independent decision support system see Reggia and Perricone[8].

Summary

The methodology presented in this paper showed that invariant relationships exist for similar projects. New projects may be compared to the baselines of these invariant relationships to determine when projects are getting off track.

The ability of the manager to interpret the measures that fall outside the norm is dependent on the amount of information the underlying variables convey. The manager must decide what attributes are to be measured (e.g. productivity) and pick variables that are closely related to them and are also measurable throughout the project. As an example, a variable like lines of code may be too general when measuring productivity. Measuring the newly developed code, either source code or executable code, would be more informative since these variables are more directly related to effort. How applicable an interpretation is for the period currently being examined should also be considered when ordering the list. The variables the manager finally decides on are then combined to form relative measures.

One method of interpreting a relative measure is by associating lists of possible interpretations with it. When a relative measure appears outside the norm, the list of possible interpretations is considered. If more than one relative measure is outside the norm the lists are combined. The more times a possible interpretation is repeated in the lists, the greater the probability it is the cause. How applicable an interpretation is for the period being examined should also be considered when ordering the list. The manager must investigate the suggested causes to determine the real one.

Conclusion

The ability to monitor a projects development and detect problems as they develop may be feasible. The methodology proposed showed favorable results when examining a past case.

The use of baselines and lists of interpretations for comparing projects provides an easy method for monitoring software development. Both the baselines and the lists of interpretations may be updated as new projects are developed. As more knowledge is gleaned the accuracy of this system should improve and provide a valuable tool for the manager.

Acknowledgements

The authors would like to thank Dr. Jerry Page of Computer Sciences Corporation and Frank McGarry of NASA/Goddard Space Flight Center for their insight and advice.

References

[1] Card, David, Frank McGarry, Jerry Page, Suellen Eslinger, and Victor Basili, The Software Engineering Laboratory, SEL-81-104, Software Engineering Laboratory Series, Goddard Space Flight Center, February 1982.

[2] Church, Victor, David Card, Frank McGarry, Jerry Page, and Victor Basili, Guide To Data Collection, SEL-81-101, Software Engineering Laboratory Series, Goddard Space Flight Center, August 1982.

[3] SEL,, Collected Software Engineering Papers: Volume 1, SEL-82-004, Software Engineering Laboratory Series, Goddard Space Flight Center, July 1982.

[4] Walston, C. E. and C. P. Felix, A Method of Programming Measurement and Estimation, IBM Systems Journal, January 1977.

[5] Basili, Victor R. and Karl Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, Journal of Systems and Software, 1981.

[6] Bailey, John W. and Victor R. Basili, A Meta-Model for Software Development Resource Expenditures, Proceedings, Fifth International Conference on Software Engineering, September 1981.

[7] The Role of Measurements in Programming Technology, Lecture presented at University of Maryland, November 15, 1982.

[8] Reggia, James and Barry Perricone, KMS Manual, TR-1136, Department of Mathematics, University of Maryland Baltimore County, January 1982.

[9] Minsky, M. L., A Framework for the Representation of Knowledge, The Psychology of Computer Vision, pp. 211-280, McGraw Hill, New York, 1975.

OVERVIEW

- A GENERAL METHODOLOGY TO MONITOR SOFTWARE DEVELOPMENT TO DETECT PROBLEMS EARLY

- THE METHODOLOGY MUST:

  REQUIRE MINIMAL OVERHEAD FOR DATA COLLECTION

  PROVIDE AN EASY WAY TO INTERPRET DATA

  BE ADAPTABLE TO CHANGING CONDITIONS

METHODOLOGY

- DEVELOP A SET OF GOOD PREDICTORS FOR THE DEVELOPMENT ENVIRONMENT
- NORMALIZE THE MEASURES TO DEVELOP BASELINES BASED UPON PAST PROJECTS
- COMPARE A DEVELOPING PROJECT TO KNOWN BASELINES TO DETERMINE DIFFERENCES FROM KNOWN BASELINES
- INTERPRET THE DATA TO EVALUATE THIS DEVIATION
- IF THERE IS A PROBLEM, DETERMINE HOW TO CORRECT IT

APPROACH

- PERFORM A PILOT STUDY
  TRIAL METRICS, BASELINES
  EVALUATE FEASIBILITY
  (DONE: CARL DOERFLINGER)
- BUILD KNOWLEDGE-BASED SYSTEM
  USING PILOT STUDY METRICS
  IMPROVING INTERPRETATION AND KNOWLEDGE MECHANISM
  (JUST STARTED: CONNIE RAMSEY)
- INVESTIGATE OTHER METRICS
  ERRORS
  ERROR CATEGORIES
  (IN PROGRESS: DEBA PATNAIK)

- MEASUREMENT POINTS (Pi)

  COMMON ACROSS DATA BASE OF PROJECTS

  NORMALIZED OVER TIME

  REASONABLE TO MEASURE

- PILOT STUDY MEASUREMENT POINTS:

  START DESIGN

  50% DESIGN

  START OF CODING

  20% CODING

  40% CODING

  50% CODING

  60% CODING

  80% CODING

  START OF SYSTEM & INTEGRATION TEST

  50% SYSTEM & INTEGRATION TEST

  START ACCEPTANCE TEST

  END ACCEPTANCE TEST

- MEASURES (Mi)
  AVAILABLE ACROSS MOST OF PROJECT
  INVARIANT TO SIZE, CALENDAR TIME, ETC.
  AVAILABLE ON SEVERAL PRIOR PROJECTS
  EASY TO COLLECT
- DATA AVAILABLE IN SEL:
  COMPUTER TIME
  COMPUTER RUNS
  PROGRAMMER HOURS
  LINES OF SOURCE CODE
  SOFTWARE CHANGES
- TRAIL METRICS FOR PILOT:
  COMPUTER RUNS/LINE OF SOURCE CODE
  COMPUTER TIME/LINE OF SOURCE CODE
  SOFTWARE CHANGES/LINE OF SOURCE CODE
  PROGRAMMER HOURS/LINE OF SOURCE CODE
  COMPUTER TIME/COMPUTER RUN
  SOFTWARE CHANGES/COMPUTER RUN
  PROGRAMMER HOURS/COMPUTER RUN
  COMPUTER TIME/SOFTWARE CHANGE
  PROGRAMMER HOURS/SOFTWARE CHANGE

BASELINES/DEVIATIONS

- ASSUMPTIONS:

  METRICS HAVE SIMILAR BEHAVIOR AT EACH POINT

  METRICS DO NOT VARY TOO MUCH OR TOO LITTLE AT Pi

  PROJECT ENVIRONMENTS ARE SIMILAR

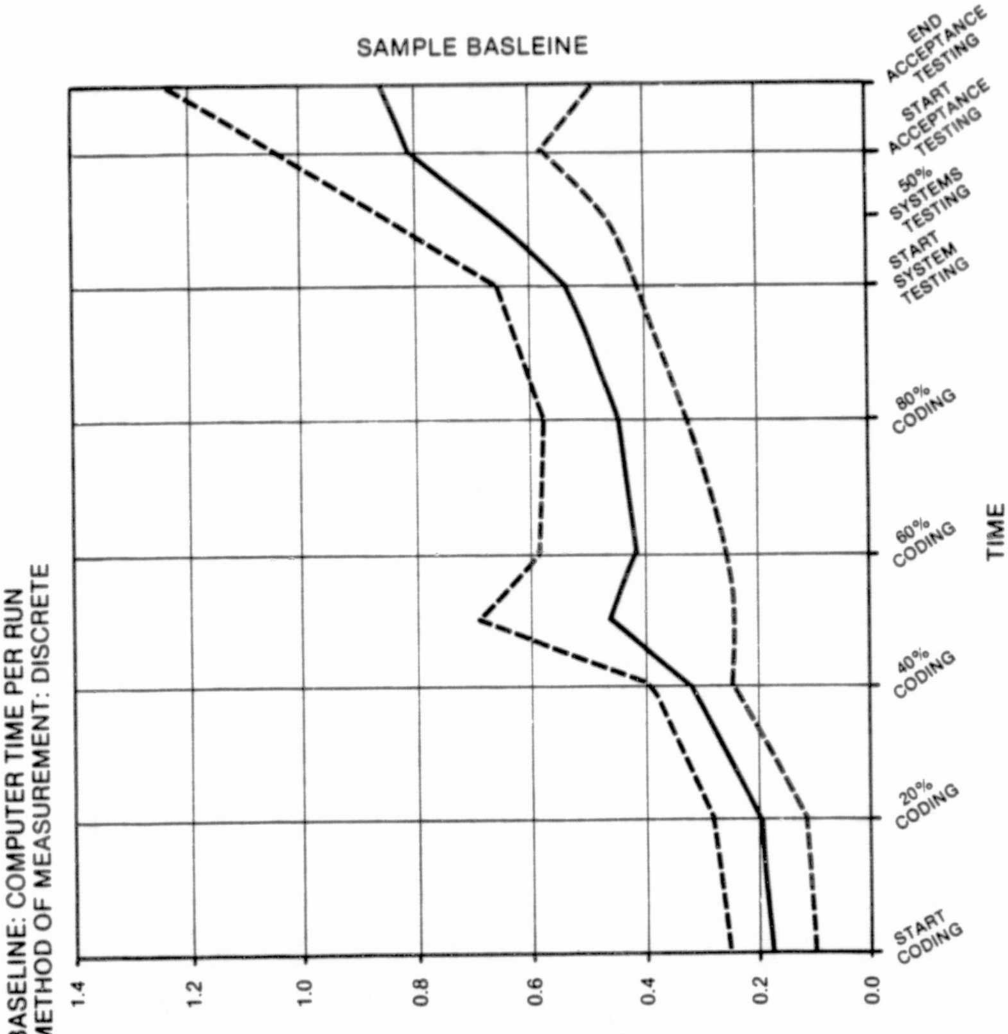
  DEVIATION FROM NORM IMPLIES SOMETHING INTERESTING

- PILOT STUDY:

  DATA: 9 PROJECTS IN BASELINE

  BASELINES: METRIC AVERAGE AT Pi
  CUMULATIVE
  DISCRETE

  DEVIATION: MORE THAN ONE STANDARD DEVIATION FROM THE NORM


SAMPLE BASLEINE
BASELINE: COMPUTER TIME PER RUN
METHOD OF MEASUREMENT: DISCRETE
COMPUTER TIME PER RUN
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
TIME
START CODING
20% CODING
40% CODING
60% CODING
80% CODING
START SYSTEM TESTING
50% SYSTEMS TESTING
START ACCEPTANCE TESTING
END ACCEPTANCE TESTING

INTERPRETATION

- SET OF MEANINGS FOR EACH Mi AT EACH Pi
  FOR DEVIATION ABOVE THE NORM
  FOR DEVIATION BELOW THE NORM
- SET OF MEANINGS AT Pi COMBINED
- MOST LIKELY INTERPRETATION DERIVED FROM SET OF MEANINGS
- MANAGERS PERSONAL KNOWLEDGE ELIMINATES SOME INTERPRETATIONS
- PILOT STUDY:
  MEANINGS ASSOCIATED WITH Mi AT Pi GIVEN BY MANAGERS
  VALUE OUTSIDE STANDARD DEVIATION GENERATES MEANING SET
  RANKING BASED ON NUMBER OF TIMES EACH MEANING APPEARS

MEANING + RANKING + PERSONAL KNOWLEDGE = INTERPRETATION

PROGRAMMER HOURS PER LINE OF SOURCE CODE

TYPE	INTERPRETATION	CROSS REFERENCE ABOVE NORMAL	CROSS REFERENCE BELOW NORMAL
ABOVE NORMAL			
	— HIGH COMPLEXITY	1 2 7 8 9	
	— ERROR PRONE CODE	3 5 6	2 7 8 9
	— BAD SPECIFICATIONS	1 2 3	
	— CODE BEING REMOVED (TESTING OR TRANSPORTED)	1 2 3	
	— CHANGES HARD TO ISOLATE	7 8 9	
	— CHANGES HARD TO MKAE	7 9	
	— LOW PRODUCTIVITY	1 2	
BELOW NORMAL			
	— INFLUX OF TRANSPORTED CODE		1 2 3
	— NEAR BUILD OR MILESTONE DATE	6	1 2 3 8 9
	— LOW COMPLEXITY		3

SAMPLE MEANINGS FOR PILOT ON TENTH PROJECT

AT 80% CODE:

TWO METRICS ABOVE NORM, ONE METRIC BELOW NORM

ABOVE NORM:

1. NUMBER OF COMPUTER RUNS/LINES OF SOURCE (S.D. = 1.6)
2. NUMBER OF PROGRAMMER HOURS/LINES OF SOURCE (S.D. = 1.3)

BELOW NORM:

3. NUMBER OF PROGRAMMER HOURS/COMPUTER RUN (S.D. = 1.5)

# OF OCCURANCES	MEANINGS	CONTRIBUTORS
2	HIGH COMPLEXITY	1,2
2	REMOVAL OF CODE	1,2
2	LOTS OF TESTING	1,3
1	LOW PRODUCTIVITY	1
1	BAD SPECIFICATIONS	2
1	CHANGES HARD TO MAKE	2
1	EASY ERRORS FIXED	3

PERSONAL KNOWLEDGE: NO CODE REMOVED
STANDARD AMOUNT OF TESTING

PILOT STUDY CONCLUSIONS

- METHOD VIABLE

  WORKED FOR ONE PROJECT STUDIED IN DEPTH

  MEASURES WERE EASY TO GATHER

  ADAPTABLE TO CHANGING ENVIRONMENT AND KNOWLEDGE

  AUTOMATABLE

- NEXT STEPS:

  ADD OTHER METRIC

  KNOWLEDGE BASED SYSTEM

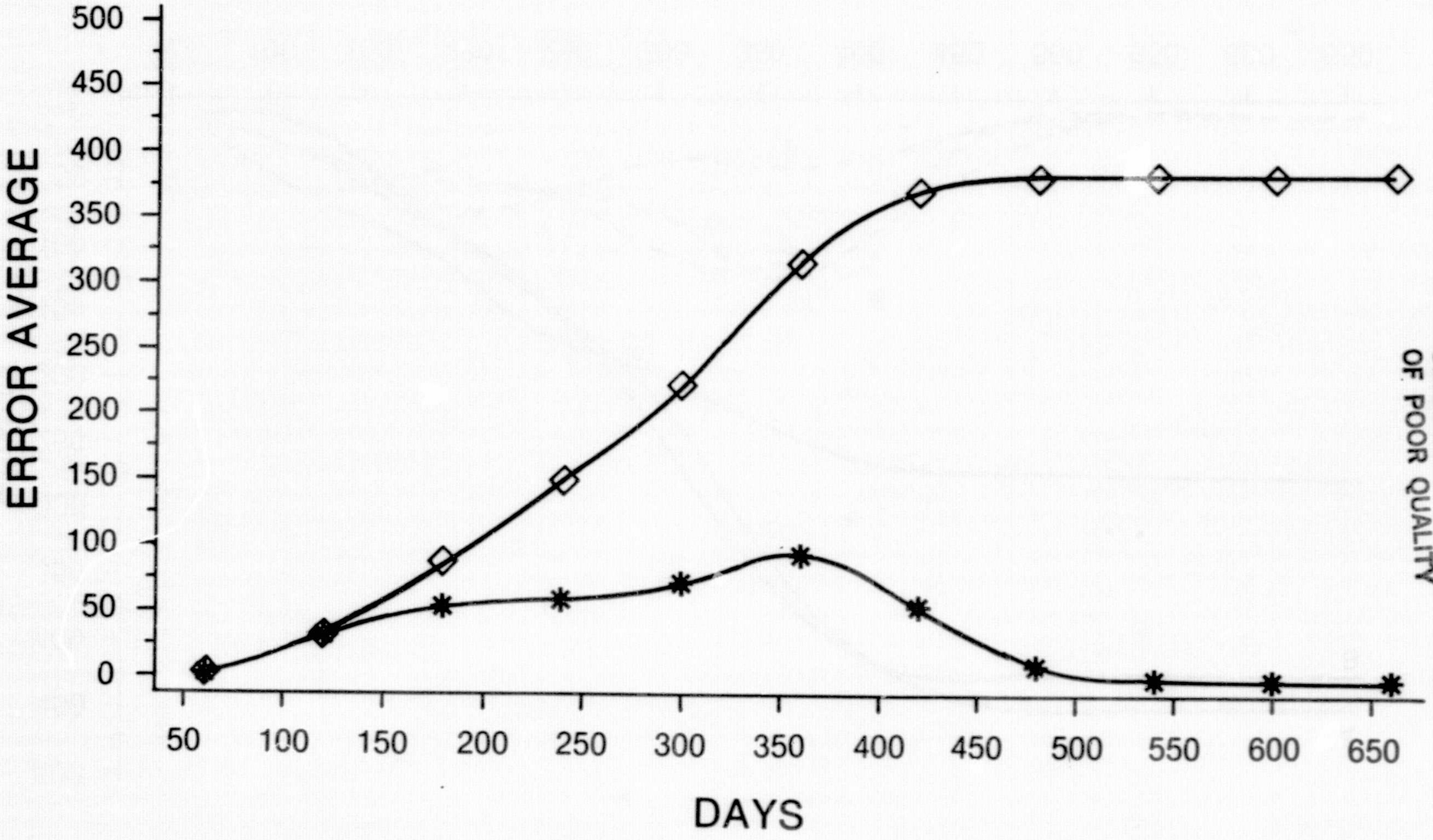
OTHER METRICS UNDER STUDY

- METRICS: ERRORS AND ERROR CLASSES
- MEASUREMENT POINTS: SAME

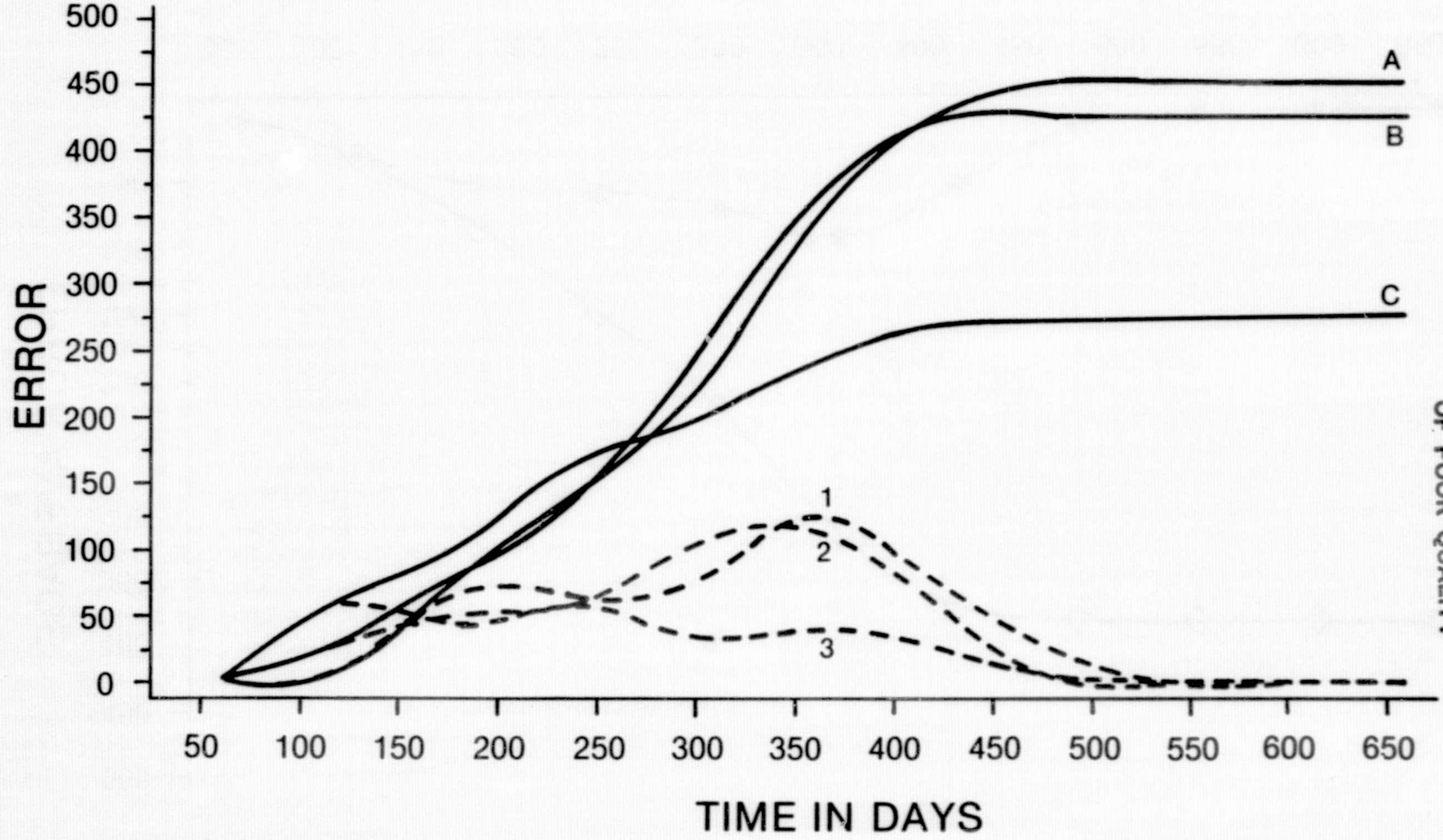
  NUMBER OF TEST RUNS TO DATE
- BASELINES: SAME

  CUMULATIVE AND DISCRETE

AVERAGE NUMBER OF ERRORS OVER TIME



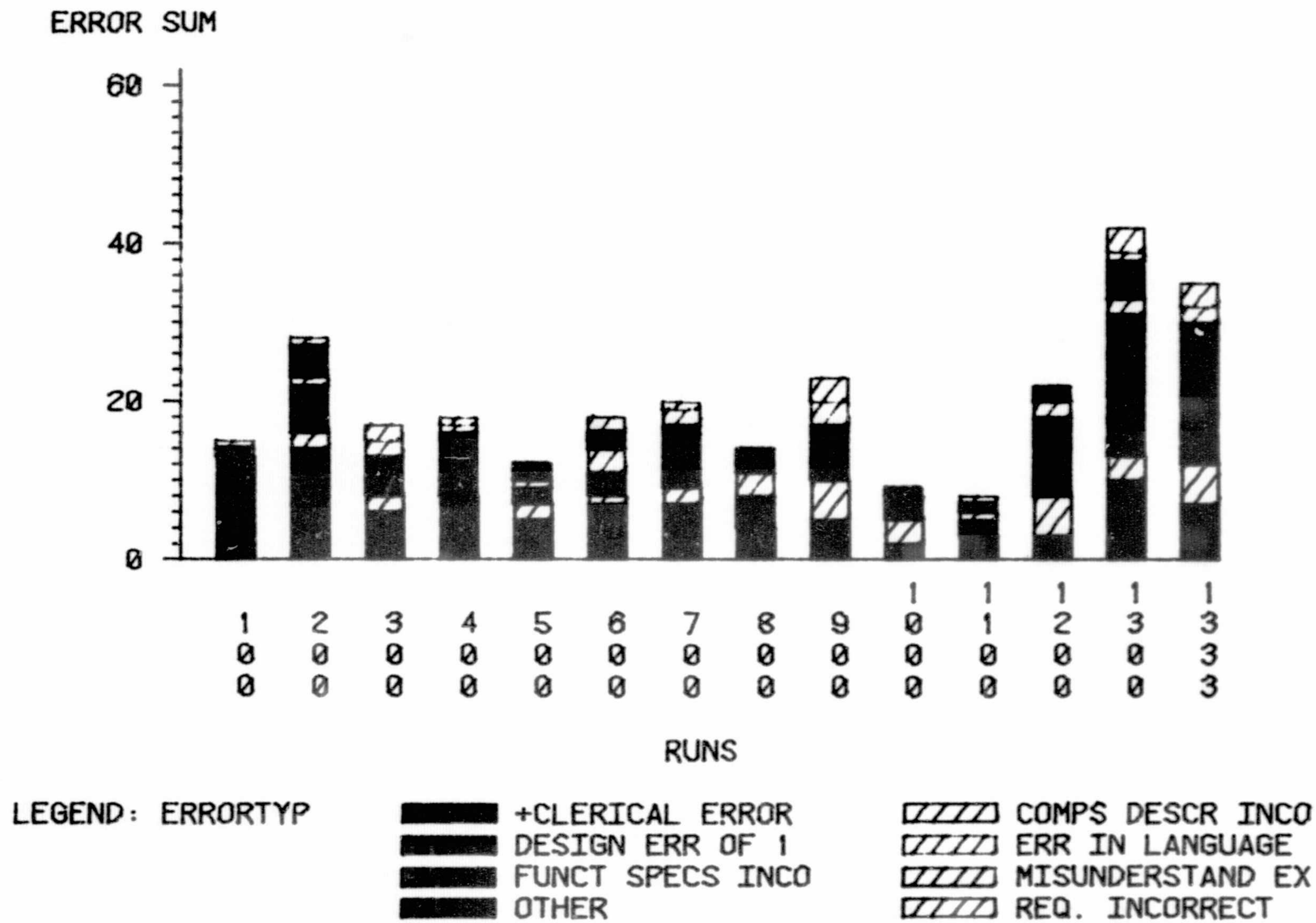
ERRORS OVER TIME



ORIGINAL PAGE IS OF POOR QUALITY

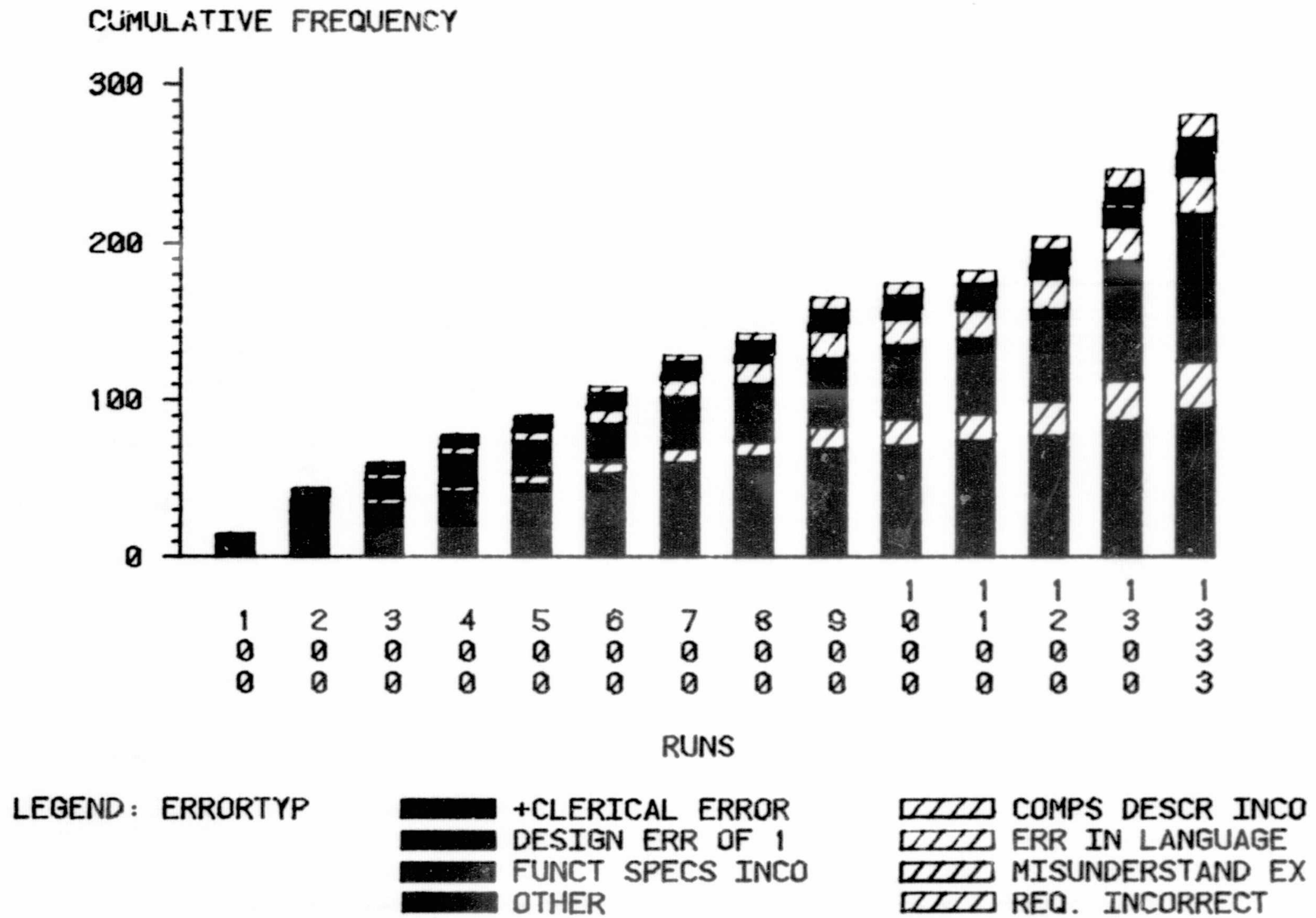
V. Basili
U of M

CHANGES DUE TO ERROR BY CAUSE



ORIGINAL PAGE IS OF POOR QUALITY

CHANGES DUE TO ERROR BY CAUSE



NEXT STEP

- WE ARE GOING TO BUILD A KNOWLEDGE-BASED SYSTEM

- HOW WILL THIS SYSTEM BE USED?

  A TOOL FOR MANAGEMENT

  — WILL INDICATE WHETHER A CURRENT PROJECT IS ON SCHEDULE

  — AUTOMATED

  — CAN BE UPDATED EASILY TO INCLUDE INFORMATION FROM NEW PROJECTS AND NEW INTERPRETATIONS AS MORE IS LEARNED

  — MANAGER MUST USE HIS OWN KNOWLEDGE OF THE PROJECT WHEN LOOKING AT THE RESULTS

- **BUILDING A KNOWLEDGE-BASED SYSTEM:**

  — USE KMS — A GENERAL SYSTEM USED FOR BUILDING KNOWLEDGE-BASED TOOLS (AVAILABLE AT UNIVERSITY OF MARYLAND)

  — THERE ARE TWO DIFFERENT APPROACHES:

  - PRODUCTION RULES
  - HYPOTHESIZE AND TEST

  WE WILL TRY BOTH AND COMPARE

- **METHOD**

  1. BUILD RULES FOR KMS
  2. INPUT DATA FROM MANY SIMILAR PROJECTS IN SAME ENVIRONMENT
  3. GIVEN NEW PROJECT, CAN COMPARE CERTAIN METRICS TO THOSE IN THE SYSTEM IN AUTOMATED MANNER. KNOWLEDGE-BASE INDICATES ABNORMALITIES.
  4. UPDATE

POTENTIAL SCENARIO
BETWEEN MANAGER AND SYSTEM

KB = KNOWLEDGE-BASED SYSTEM
M = MANAGER

KB: READY FOR COMMAND

M: OBTAIN DIAGNOSIS

KB: STAGE:

(1) START CODING
(2) 20% CODING
(3) 40% CODING
(4) 50% CODING
(5) 60% CODING
(6) 80% CODING
(7) START SYSTEM TESTING
(8) 50% SYSTEM TESTING
(9) START ACCEPTANCE TESTING
(10) END ACCEPTANCE TESTING

M: <u>8</u>

KB: GOODNESS OF TESTING:

(1) GOOD
(2) FAIR
(3) POOR

M: <u>3</u>

KB: DIAGNOSIS:

POOR TESTING PROGRAM	<0.60>
GOOD CODE	<0.05>
CHANGES HARD TO ISOLATE	<0.25>
CHANGES HARD TO MAKE	<0.10>

SUMMARY

- CHOOSE MEASUREMENT POINTS (Pi)
- CHOOSE A SET OF NORMALIZED INVARIANT MEASURES (Mi)
- DEVELOP A SET OF BASELINES FOR EACH Mi AT EACH Pi
- CHOOSE BOUNDS ON DEVIATIONS FROM THE BASELINES
- ASSOCIATE POSSIBLE MEANINGS FOR DEVIATIONS (+ AND −) FROM THE BASELINES FOR EACH Mi AT EACH Pi
- DEVELOP A MECHANISM FOR DERIVING INTERPRETATIONS
- INCORPORATE PERSONAL KNOWLEDGE OF PROJECT
- GENERATE MOST LIKELY INTERPRETATION(S)

D3

ORIGINAL PAGE IS
OF POOR QUALITY

N84 23140

CHARACTERISTICS OF A PROTOTYPING EXPERIMENT

JUDIN SUKRI AND MARVIN V. ZELKOWITZ
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND 20742

INTRODUCTION

In 1982, NASA Goddard Space Flight Center began a project to prototype a new proposed software system. Since the system, the Flight Dynamics Analysis System (FDAS), was to be a source code control system, and not the more typical flight dynamics software which NASA personnel were more familiar with, the decision was made to prototype an initial implementation in order to gain insights into the actual features needed to build a full FDAS and to evaluate the idea of a prototype in the NASA environment. This report describes the status of that project at the end of 1983.

PROTOTYPING

In developing the prototype for NASA we need to understand what a prototype is. More importantly, for NASA, the issue of prototyping must answer the following questions:

(1) What are the goals of a prototype? Is it to develop the requirements for a product? Evaluate its performance? Predict its final costs?

(2) What are the issues involved? How does one design for a prototype? Does the software lifecycle change? Do we want multiple prototypes for different phases of the life cycle? How do we use a prototype when built?

(3) What tools can be used to design a prototype? to build a prototype? to evaluate a prototype?

(4) How does one measure a prototype? How do you know if your prototype was successful? Should you invest the cost and build the full system or abandon the project? What SHOULD a prototype cost? 10% of the final product or 50% or even 100%?

FLIGHT DYNAMICS ANALYSIS SYSTEM (FDAS)

The Flight Dynamics Analysis System (FDAS) is being built to aid experimenters try alternative flight dynamics models. Currently if an experiment is to be run (e.g., try a new orbit calculation model), the experimenter must access the Fortran source library, know which module to modify, make the changes, test the changes, recreate a new load module, and then run the experiment. The experimenter must have detailed knowledge of the software.

With FDAS, the experimenter enters the system, and interacts with a data base, directs the system to modify the correct module and aids in the change. Thus changes to software are easier, require less time and less expertise about the internals.

FDAS consists of two major components - a source code control system to manage the libraries of software modules needed for each application program, and a form of data abstraction allowing applications programmers the ability to write programs using flight dynamics data types (e.g., state, cartesian coordinates, vector locations, etc.). These features are somewhat independent and can be evaluated separately.

In order to manage source code, the applications programmer enters a tree chart of modules (the program's structure). Usually this will be a full system developed by someone else. The applications programmer can then tell the system to edit specific modules and to replace other modules by new ones. The system maintains the current set of modules for the system, and keeps track of which modules have been altered and which ones need to be compiled. In some ways, this model is very much like a combination of both the Source Code Control System (SCCS) and the MAKE processors running under UNIX systems.

In order to aid the applications programmer, a form of data abstraction has been proposed. A set of standard types have been defined. A programmer may code using these types, and a preprocessor converts this code into standard Fortran. A generalized input-output structure has been defined for data of this type. The programmer may write (PUTOUTPUT) the name and value of any datum from one module, and read (GETINPUT) the name and value of that datum in another module. An intial design decision was to restrict abstract data to their own statements and not mix them with the Fortran statements.

In order to build the prototype, the following general strategy is being used:

(1) A subset of the requirements for FDAS were written and a prototype built to those requirements.

(2) Data was collected automatically by the FDAS prototype on user interaction with the system.

(3) The usual Software Engineering Laboratory data on programmer activities were collected during the development phase.

(4) The prototype will be evaluated by four groups representing four different views of the system. A group of applications programmers (the "users") will use FDAS and report on its usefulness in solving their flight dynamics problems, a group from the Software Engineering Laboratory will evaluate the FDAS model as an appropriate one for solving flight dynamics problems, a research group is looking at FDAS as an example of a source code control system, and the developers are evaluating the implementation itself, and issues such as efficiency, size, and extendability to a full system.

(5) Beginning in the early spring of 1984, a new task will begin to design the "full" FDAS system. The experiences in the prototype will undoubtedly be helpful in designing and building the full system, but there is no committment to using either the design or the source code of the prototype.

(6) After the full system is built, it will be compared with the initial effort. The effectiveness of the prototype on the final product will be evaluated. Was FDAS cheaper to build? Will it be more reliable? Will it be more efficient? Will it have a better man/machine interface?

INITIAL EVALUATION

The initial requirements for FDAS began early in 1982. The requirements and initial design for the prototype were done in the fall of 1982 and the initial implementation of the prototype began in January, 1983. As with many software projects, the task was bigger than expected, so an initial prototype was tested in July of 1983, but the "full" prototype was not available until October. The evaluation phase is to last until late February, 1984.

Although it is a prototype, it is not a small system. There are 34K lines of Fortran source code running under VMS on a VAX 11/780 computer. Of the 34,000 lines (including comments), there are 20,200 lines of executable Fortran source statements. The prototype was installed with only one small applications system of 3,000 lines for experimentation. This size of 34K is already within the size range of other larger "full" systems built by NASA.

Some of the data collected can be summarized by the following table. In addition to FDAS, there is data from 11 previous projects monitored by the Software Engineering Laboratory and data from two other projects now under development.

Phase	11 Proj	Cont 1	Cont 2	FDAS
Design	22%	31%	31%	39%
Code	48%	43%	69%	61%
Test	30%	26%*	0%*	0%*

ORIGINAL PAGE IS
OF POOR QUALITY

*- Data still being collected

As can be seen, historically, coding is over twice the design effort. That is also true with one of the current projects and is almost true with the other contemporary project. But it is most definitely not true with FDAS. This reflects the high design costs since it had "never been done before." It also reflects the relatively low priority given to full debugging and testing, up to NASA standards, of the resulting code. Since the prototype has a limited lifetime, "hard" problems were deleted from the prototype requirements, and users had to live with annoying but non-critical bugs. (Note: At the time that this was written, the full data from testing FDAS was not yet entered into the data base, so full testing data is not yet available.)

The time spent in design, can be summarized as follows:

Hours	11 Proj	Cont 1	Cont 2	FDAS
Design	21709	5885	10758	4508
Total	100324	19085	34461	10477*

*- Still being collected

As can be seen, the 10,477 hours represents a sizeable effort, and is beyond the "toy" prototype stage.

Just using the system has shown some other useful aspects to the system. One critical command, the DEFINE command, has been particularly hard to use, so it will need a better definition and documentation in the full system. The overhead imposed by FDAS also seems tolerable. For example, with compilation times of 10 seconds standard, a preprocessor overhead of 2 seconds is tolerable. In addition, since the linkage time for the application system is 18 seconds, the 3 second FDAS overhead on top of this is also small. However, the use of the preprocessor seems unduly inflexible and should be revised for the full system.

A final complexity in this evaluation is the always changing requirements. When originally conceived, FDAS would be an experimental system used on a VAX 11/780. However, in the two years since the idea was proposed, the operational groups at NASA are interested in the system, and would like such a tool on their operational computer - an IBM 4341. Thus part of the evaluation (new requirements?) is to consider a 4341 implementation, or an implementation that can easily be transported to both systems. While this will undoubtedly make a comparison between the full system and the prototype harder to do, since the operational environments (and hence the projects' requirements) are different, it is certainly to NASA's advantage to have built the prototype so that all groups can view it before a final decision was made to build it in one particular environment.

SUMMARY

The evaluation phase is still going on, so it is not possible to give a full evaluation. However, some results are now apparent.

(1) The source code control aspects of FDAS are useable, and can be developed into a good operational system.

(2) The data abstraction language and preprocessor need to be rethought and the features need to be generalized.

(3) The prototype and the underlying application are both written in Fortran. There is no need for that to be so. It should be possible to monitor any source code application package regardless of the language in which FDAS is written.

(4) The use of the prototype has uncovered many minor and major defects in the design of such a flight dynamics analysis system. Some original assumptions made during the design phase turned out not to be true under actual usage conditions.

Because of these experiences, many defects in FDAS have been discovered before a full system is built. From the data collected so far, it appears as if FDAS will be a large system when built. The development of the prototype should aid NASA in avoiding costly mistakes later.

ACKNOWLEDGEMENT

This paper was supported by NASA grant NAG5-368 to the University of Maryland. We also acknowledge the many programmers and analysts at NASA Goddard Space Flight Center and Computer Sciences Corporation, including Sharon Walagora and Glenn Snyder, principal designers of the prototype, for their efforts in building FDAS. This report is mainly a collection of their experiences.

CHARACTERISTICS OF A PROTOTYPING EXPERIMENT

Marvin V. Zelkowitz

Department of Computer Science

University of Maryland

PROTOTYPING IS OF CURRENT INTEREST

But is it:

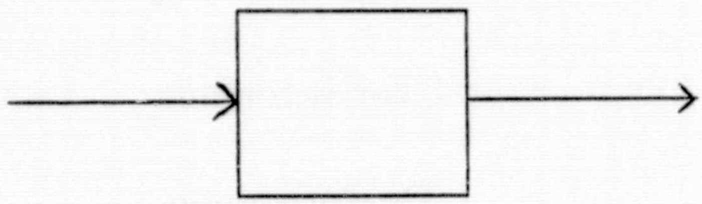
Quick and dirty throw-away?

Subset implementation?

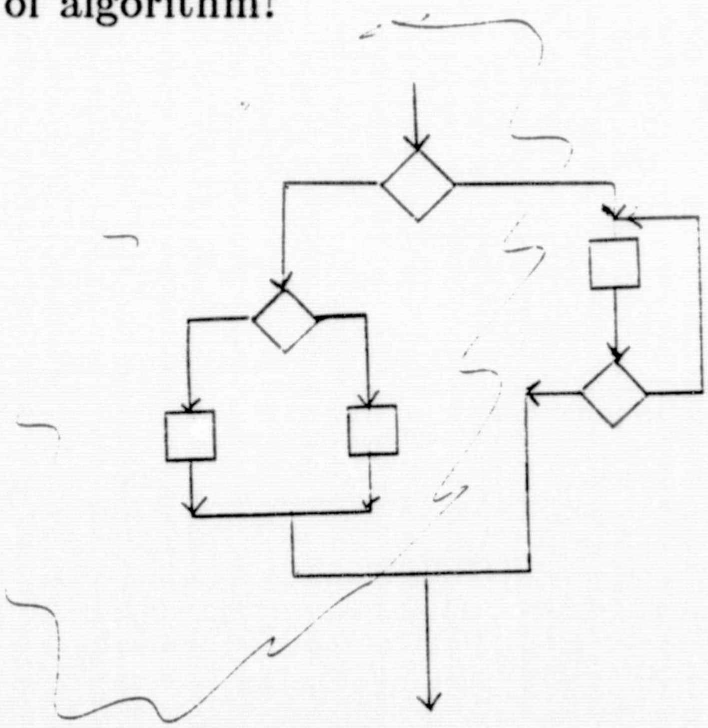
Release 1 of full system?

DO YOU MODEL:

Input-output behavior?



Part of algorithm?



USES OF A PROTOTYPE:

Feasibility of full system

User interface

Performance

Costs

M. Zelkowitz
U of M

RESEARCH ISSUES:

How to measure a prototype?

What are profiles of a prototype

(baselines)?

How to evaluate a prototype?

PROTOTYPING MODELS

Prototype is cheap, system expensive

Prototype is expensive, system cheap

Both expensive, but better system

(more reliable, better user interface)

NASA/GSFC FDAS PROTOTYPE

(FLIGHT DYNAMICS ANALYSIS SYSTEM)

Now:

Access Fortran library

Modify subroutines

Recompile and link

Run experiment

===> Need details of implementation

FDAS:

Access FDAS

FDAS accesses Fortran code

Modifications easier

===> Modifications require less time and effort

FACTORS IN SOFTWARE DEVELOPMENT

FACTOR	Usual Project	FDAS
Requirements	Known	?
Size	Known	?
Execution	Known	?
Algorithm design	Known	?
User interface	Known	?
Cost	Known	?

GOALS OF FDAS:

Decrease experimental setup time

Solve more problems than is possible today

Lower required knowledge of system

Ease of use of experimental system

Lower software costs to add to FDAS

FEATURES:

Source code control

Data abstractions

(e.g., state, cartesian)

Generalized input-output

SCHEDULE

Requirements - Summer-Fall, 1982

Implementation - January-June, 1983

ACTUAL SCHEDULE:

Requirements - Summer-Fall, 1982

Release 1 - January-July, 1983

Release 2 - July-October, 1983

Evaluation - October-December, 1983

SIZE OF FDAS

Source code - 34K

Executable Fortran statements - 20.2K

Application area - 3K

EFFORT BY MILESTONES

Phase	**11 Projects**	**Pred.**	**Cont 1**	**Cont 2**	**FDAS**
Design	22%	17%	31%	31%	39%
Code	48%	36%	43%	69%	61%
Test	30%	47%	26%*	0%*	0%*
Code/Design	2.2	2.1	1.4	2.2	1.6
Hours					
Design	21709	2045	5885	10758	4508
Total	100324	11835	19085	34461	10477

* - Data still being collected

EVALUATORS

NASA/GSFC - FDAS for flight dynamics

CSC SEL - Use of data types

UNIV. OF MD - FDAS as source code support

Developers - Evaluate FDAS capabilities

EVALUATION CRITERIA

Usable - How easy to set up

Flexible - Can user alter code easily

Adaptable - Can FDAS be altered

Consistent - Can it be used across applications

Reliable - Can new applications be added

Stable - Does is fail

Speed - How fast does it execute

SOME SUBJECTIVE COMMENTS:

As expected, some hard decisions delayed

Addition of release 1 to schedule

Some features dropped

Reliability not up to usual standards

But system is not an operational one

Floating requirements

Full system on VAX or 4341s?

C-2

ADDITIONAL COMMENTS

Some commands redefined

(DEFINE not well understood)

Cost of system minimal compared to system overhead

(preprocessor-2 sec. compiler-10 sec.)

(build time-3 sec. link time-18 sec.)

SUMMARY

Still need to complete evaluation -

More data to collect

Need to evaluate error data

Prototype profile reflects quick development

Problems in user interface discovered early

PANEL #2

TESTING PROCEDURE

J. Ramsey, University of Maryland
A. Goel, Syracuse University
C. Savolaine, Bell Labs

D4

N84 23141

Structural Coverage of Functional Testing.

James Ramsey

University of Maryland
at College Park.

Abstract

A FORTRAN program has been instrumented to produce structural coverage measures. The structural coverage profiles of functionally generated acceptance tests and operational usage are used to examine two areas in software engineering: the examination of faults and the applicability of reliability models.

This paper describes a study performed at NASA's Goddard Space Flight Center, Greenbelt, Maryland by researchers at the University of Maryland at College Park. A ten thousand line FORTRAN program was modified to produce a structural coverage metric. After execution, the modified program produces a list of executed statements. The program was executed using both functionally generated acceptance tests and operational usage cases yielding structural coverage measures [CSC 78]. The program's software failures during maintenance were recorded.

The study collected structural coverage data for both acceptance test and operational usage and error data about faults revealed during maintenance. Using these data, some simple questions can be answered immediately. "How much of the code is executed by functionally generated acceptance testing? (both by individual tests and by the entire test suite)". Individually, the test cases execute from 27% to 47% of

This research is funded by NASA grant NSG-5123.

the executable statements. In total, 56% of executable statements are executed. This percentage does not include statements executed in either unit test or system test.

"How many procedures are executed by functionally generated acceptance test"? Anywhere from 48% to 69% for individual tests, for a total of 75% of procedures.

More complicated questions compare acceptance test coverage to operational usage coverage. "Does acceptance test execute the same code as operational usage"? Yes, more or less. "Does operational usage exercise code not exercised by acceptance test"? Yes, about 8% of the total executed code. The code executed by operational usage but not by acceptance test contained a mix of statement types different than acceptance test alone.

There were eight faults revealed during maintenance. Each fault was contained in one procedure; one procedure contained two faults. There are not enough faults to reach any firm conclusions, however I feel there is enough information to inspire interesting questions.

Are there faults revealed in maintenance in sections of code unexecuted in acceptance test? No, although 8% of the code could contain such a fault. If faults had occurred in the untested 8% then perhaps the functional tests could be improved by structural coverage testing. Since structural coverage testing would require executing every statement, it might have executed the code and revealed the fault.

"Are faults more likely to be revealed in heavily executed procedures?" Procedures were classified by the number of times they were executed in operational usage. Half of the procedures were executed by more than 90% of the operational usage cases. About half of the revealed faults occurred in this group of procedures (3 of 8).

Information on each fault was collected using the SEL change report form [SEL 82]. Faults are categorized by "time to isolate the error", "the time to understand and implement", and the section "type of error"*.

Time to isolate the change seems to be independent of procedure coverage. Increased usage seems to be associated with a longer time to understand and implement a change. This might be explained by suggesting that the lightly exercised procedures contain fairly simple code whereas the heavily exercised code is, by necessity, more complicated and requires more time to modify. There are too few faults to reveal any interesting patterns between fault types and procedure coverage in operational usage.

References

[CSC 78] Computer Sciences Corporation, Acceptance Test Methods, CSC/TM-78/6296, 1978.

[SEL 82] Guide to Data Collection, SEL-81-101, Software Engineering Laboratory Series, Goddard Space Flight Center, Greenbelt, Maryland, August 1982.

---

* Time to isolate the error is classified as taking: less than one hour, one hour to one day, greater than one day, never found. Time to understand and implement the change is classified as taking: less than one hour, one hour to one day, one day to three days, or greater than three days. Faults are categorized as originating in the: requirements, functional specification, design (either involving data or expression), external environment, use of language, clerical or other.

Statement Coverage
by 10 Acceptance Test Cases.
(Percentage of Maximum)

Case	Procs	Exec	Assign	Calls	Do	If	Reads	Writes
t1	50.0	27.5	31.1	27.5	34.4	34.1	17.6	6.3
t1a	48.5	24.9	28.3	18.2	33.1	32.7	17.6	6.3
t1b	44.1	21.2	23.9	20.1	23.6	27.0	17.6	4.9
t2	50.0	27.2	30.6	27.5	34.4	33.9	17.6	6.3
t2a	48.5	24.8	28.3	18.2	33.1	32.7	17.6	6.3
t2b	44.1	21.7	24.4	20.1	24.8	27.8	17.6	5.3
t3	48.5	24.4	27.8	18.4	32.5	32.0	17.6	5.8
t4	60.3	37.9	43.3	37.8	53.5	45.3	32.4	12.1
t4a	54.4	30.3	33.8	26.3	39.5	38.2	32.4	10.7
t4b	44.1	21.6	24.3	20.1	24.8	27.6	17.6	4.9
t4c	52.9	28.6	33.3	24.2	38.9	36.9	17.6	6.8
t4d	44.1	21.6	24.3	20.1	24.8	27.6	17.6	4.9
t5	69.1	47.1	52.6	55.7	54.8	55.0	41.2	12.6
t5a	64.7	39.0	43.9	38.5	45.2	48.9	32.4	10.2
t5b	67.6	41.5	45.7	51.7	48.4	49.8	26.5	7.8
t6	67.6	42.7	47.4	51.7	48.4	51.8	29.4	10.7
t6a	55.9	29.9	34.2	24.4	36.9	37.8	26.5	9.7
t6b	58.8	33.7	37.0	39.7	36.3	43.0	20.6	5.8
t7	66.2	39.0	43.8	40.4	44.6	48.7	26.5	9.7
t8	66.2	45.6	51.2	50.0	54.1	55.0	38.2	12.1
t9	66.2	41.0	46.0	42.3	46.5	50.9	35.3	11.7
t10	66.2	40.2	44.9	40.9	45.2	50.3	35.3	11.7
Union	75.0	56.0	63.5	68.4	68.8	65.1	41.2	14.6
Intersect	42.6	18.1	20.8	10.0	22.3	24.7	17.6	4.9

Statement Coverage
by 60 Operational Useage Cases.
(Percentage of Maximum)

Case	Procs	Exec	Assign	Calls	Do	If	Reads	Writes
1	57.4	31.8	35.3	29.9	33.1	43.0	29.4	6.8
2	63.2	39.8	44.5	46.2	51.0	50.6	29.4	9.2
3	66.2	42.6	47.9	44.0	49.7	54.6	38.2	10.7
4	54.4	29.3	33.4	20.6	36.3	36.9	41.2	11.7
5	54.4	29.1	33.0	28.7	33.8	36.7	29.4	7.3
6	52.9	25.5	28.7	20.1	31.8	34.3	26.5	6.8
7	48.5	23.5	26.0	22.5	24.8	31.3	26.5	6.3
8	57.4	31.6	34.9	30.9	33.1	44.0	26.5	6.3
9	54.4	29.0	33.1	20.1	35.7	36.5	41.2	11.2
10	54.4	29.1	33.0	28.7	33.8	36.7	29.4	7.3
11	64.7	40.5	44.4	46.9	48.4	50.7	32.4	9.2
12	54.4	29.0	32.9	28.7	33.8	36.5	29.4	7.3
13	51.5	30.1	35.6	19.4	43.3	40.6	29.4	9.2
14	51.5	29.9	35.3	19.4	43.3	40.5	29.4	9.2
15	51.5	26.4	29.1	25.4	28.7	36.1	26.5	6.8
16	67.6	41.7	45.6	51.9	48.4	50.2	35.3	9.2
17	54.4	29.6	34.1	20.6	36.3	36.9	41.2	11.7
18	54.4	29.1	33.0	28.7	33.8	36.7	29.4	7.3
19	54.4	29.5	34.0	20.6	36.3	36.9	41.2	11.7
20	54.4	29.0	32.9	28.7	33.8	36.5	29.4	7.3
21	54.4	26.0	28.4	27.0	24.8	33.6	20.6	4.4
22	63.2	38.5	43.2	37.1	43.3	48.2	41.2	12.1
23	44.1	23.1	27.0	14.8	26.8	32.1	23.5	6.3
24	44.1	22.9	26.5	15.8	26.8	32.0	23.5	6.3
25	57.4	31.7	34.5	31.6	33.8	42.8	29.4	6.8
26	50.0	28.7	34.1	18.2	42.7	38.2	29.4	9.2
27	54.4	26.1	28.3	24.9	33.1	35.2	26.5	6.8
28	54.4	29.3	33.5	20.3	36.3	36.7	41.2	11.7
29	54.4	29.5	34.0	20.6	36.3	36.9	41.2	11.7
30	63.2	41.4	45.8	45.9	51.0	54.8	29.4	9.7
31	54.4	28.3	31.7	28.9	31.8	37.5	26.5	6.3
32	44.1	23.2	26.7	15.8	26.1	32.8	23.5	6.3
33	48.5	24.9	28.8	15.1	31.2	35.1	26.5	7.3
34	30.9	13.0	16.0	5.0	15.9	14.3	23.5	5.3
35	57.4	33.1	36.4	39.2	38.2	40.5	29.4	7.3
36	54.4	29.1	33.1	20.3	35.7	36.5	41.2	11.7
37	64.7	40.5	44.4	46.9	48.4	50.7	32.4	9.2
38	54.4	29.3	33.6	20.6	36.3	36.9	41.2	11.2
39	64.7	40.7	44.5	47.6	49.0	50.9	32.4	9.2
40	55.9	29.3	32.7	28.0	35.0	39.6	29.4	7.3

ORIGINAL PAGE IS
OF POOR QUALITY

Statement Coverage
by 60 Operational Useage Cases.
(Percentage of Maximum)
(cont.)

Case	Procs	Exec	Assign	Calls	Do	If	Reads	Writes
41	57.4	30.0	34.1	24.2	36.9	38.0	35.3	11.2
42	52.9	31.4	37.2	20.8	45.2	43.3	26.5	8.7
43	54.4	29.0	33.1	20.1	35.7	36.5	41.2	11.2
44	66.2	40.4	44.8	41.1	45.2	50.7	44.1	13.1
45	66.2	46.6	51.9	51.0	54.8	57.8	47.1	13.6
46	64.7	39.2	43.8	38.8	45.2	49.3	41.2	11.7
47	57.4	30.0	34.2	24.2	36.9	38.0	35.3	11.2
48	66.2	39.1	43.7	40.7	44.6	49.1	35.3	11.2
49	66.2	45.8	51.1	50.2	54.8	55.4	47.1	13.6
50	66.2	41.2	45.9	42.6	46.5	51.3	44.1	13.1
51	57.4	31.1	34.0	30.4	34.4	42.1	29.4	7.8
52	54.4	29.6	34.0	20.6	36.9	37.2	41.2	11.2
53	50.0	27.5	31.3	26.1	29.3	35.5	26.5	7.3
54	58.8	31.5	34.8	30.1	33.1	44.2	26.5	6.3
55	58.8	33.9	36.8	40.0	36.3	43.4	29.4	7.3
56	54.4	29.1	33.0	28.7	33.8	36.5	29.4	7.3
57	54.4	29.0	32.2	27.5	34.4	40.1	26.5	6.8
58	54.4	29.6	34.1	20.6	36.3	36.9	41.2	11.7
59	50.0	24.4	27.6	17.2	31.8	32.1	26.5	7.3
60	29.4	12.3	14.6	4.5	15.3	14.1	23.5	5.3
UNION	80.9	64.1	71.9	78.2	76.4	77.2	55.9	17.5
INTERSECT	27.9	10.3	12.2	3.8	12.1	11.4	20.6	4.4

ORIGINAL PAGE IS
OF POOR QUALITY

Comparison of Statement Coverage by 10 Acceptance Test Cases and 60 Operational Usage Cases.								
Case	Procs	Exec	Assign	Calls	Do	If	Reads	Writes
Acpt	51	2408	1187	286	108	490	14	30
Usage	55	2757	1345	327	120	581	19	36
Union	55	2768	1353	327	120	581	19	36
Intersect	51	2397	1179	286	108	490	14	30
A-U	0	11	8	0	0	0	0	0
U-A	4	360	166	41	12	91	5	6

Comparison of Statement Coverage by 10 Acceptance Test Cases and 60 Operational Usage Cases. (by percentage of Maximum)								
Case	Procs	Exec	Assign	Calls	Do	If	Reads	Writes
Acpt	75.0	56.0	63.5	68.4	68.8	65.1	41.2	14.6
Usage	80.9	64.1	71.9	78.2	76.4	77.2	55.9	17.5
Union	80.9	64.4	72.4	78.2	76.4	77.2	55.9	17.5
Intersect	75.0	55.7	63.0	68.4	68.8	65.1	41.2	14.6
A-U	0.0	0.3	0.4	0.0	0.0	0.0	0.0	0.0
U-A	5.9	8.4	8.9	9.8	7.6	12.1	14.7	2.9

Time to Understand and Implement the Change vs
Number of Times Procedure was Exercised /
Total Operational Executions.

(Effort to Isolate the Cause in Parenthesis)

	< 1 hour	1 hour < 1 day	1 day < 3 days	> 3 days
100%		(1 h < 1 d)	(1 h < 1 d)	
		(1 hour <)		
90%				
80%		(1 h < 1 d)	(>1 day)	
70%				
60%				
50%		(1 hour <)		
40%	(1 hour <)			
30%				
20%				
10%		(1 h < 1 d)		

Time to Isolate the Change vs
Number of Times Procedure was Exercised /
Total Operational Executions.

(Effort to Understand and Implement in Parenthesis)

	< 1 hour	1 hour < 1 day	> 1 day	never found
100%	(1h < 1d)	(1h < 1d)		
		(1d < 3d)		
90%				
80%		(1h < 1d)	(1d < 3d)	
70%				
60%				
50%	(1h < 1d)			
40%	(1 hour <)			
30%				
20%				
10%		(1h < 1d)		

Faults by CRF Classification vs
Number of Times Procedure was Exercised /
Total Operational Executions.

	Req.	Func. Specs.	Design Data	Design Exp	Extern. Env.	Lang.	Cler.	Other
100%				x,x		x		
90%								
80%		x	x					
70%								
60%								
50%						x		
40%					x			
30%								
20%								
10%			x					

D5

N84 23142

Examining Functional Acceptance Testing With Structural Coverage Metrics

James Ramsey

University Of Maryland
At College Park

November 1983

Overview

Functionally generated acceptance tests are examined using structural coverage metrics.

Reliability Models

Software faults

Management of acceptance testing process

DEFINITIONS

Functionally generated acceptance test:
derived from the program's specifications

Structural coverage metrics:
procedure coverage
How many procedures were executed?
statement coverage
How many statements were executed?

Reliability Models
Given a history of software failures, predict:
mean time to next failure
total number of faults in the program

The programs:

Finished:

A subset of a large satellite system
FORTRAN
68 procedures
10k lines of source
4.3k executable statements
Ten acceptance tests
not a rigorous sampling of the input domain
but not trivial
60 operational use cases
Fault data for acceptance test and operation

In progress:

A whole satellite system:
FORTRAN
300 procedures
50k lines of code
20k executable statements
Fault data for system test, acceptance test, and operation

Structural Coverage of Acceptance Test

Executable Statement Coverage by 10 Test Cases.			
Case	Procedures Executed (%)	Executable Statements (%)	% Unique Code
t1	50.0	27.5	0.0
t2	50.0	27.2	0.0
t3	48.5	24.4	0.0
t4	60.3	37.9	4.4
t5	69.1	47.1	1.7
t6	67.6	42.7	0.0
t7	66.2	39.0	0.0
t8	66.2	45.6	1.0
t9	66.2	41.0	0.0
t10	66.2	40.2	0.0
Cumulative	75.0	56.0	
Intersect	42.6	18.1	

Note:

44% of executable statements were not exercised in acceptance test. They may have been executed in system / unit testing.

Structural coverage of 60 executions by users after acceptance test:

Structural Coverage of 60 Operational Usage Cases.		
	Procedures Executed (%)	Executed Statements (%)
Cumulative	80.9	64.9
Intersection	27.9	10.3

10% of the code was executed by ALL of the operational cases.

Are the acceptance tests representative of operational usage?

This assumption MUST be true if using acceptance test failures to predict failures in operational usage.

Are the acceptance tests representative of operational usage?

Might not be valid to use reliability data gathered in acceptance test to predict failures in operational use

The "mix" of statements in the 8.4% differs from the "mix" of statements in the 55.7%

twice as likely to execute a CALL or IF

Otherwise, cannot distinguish acceptance tests from operational usage cases by their structural coverage numbers

No faults were revealed in the 8.4%

- If faults had been revealed in the 8.4%, then there was a flaw in the test plan
  - chance to augment the tests
  - chance to re-evaluate how tests are written

Faults

8 faults revealed in operation

all repaired by changing one procedure

one procedure contained two faults

How are these related?

Time to isolate the fault

Time to understand and implement the change

Number of times the procedure is executed / 60

Questions:

Are faults more likely to be revealed in heavily exercised code? lightly exercised code?

Are there relationships between time to isolate the fault and how thoroughly the procedure is exercised?

Are "time to isolate" and "time to understand and implement" related?

Are heavily exercised procedures more likely / less likely to contain a fault? Enticing but inconclusive with only 8 faults.

Number of Times Procedure was Exercised / Total Operational Executions	Faults	Procedures
100%	* * *	p p p p p p p p p p p p p p p p p p p p p p p p p p p
90%		p p p
80%	* *	p p p p
70%		p
60%		p p p p
50%	*	p p p p
40%	*	p p p p p p
30%		p p p
20%		
10%	*	p p p p p p p
0%		u u u u u u u u u u u u u

Half of the 55 procedures were executed by 90% or more of operational usage cases.

Is there a relation between time to isolate the fault and how well the procedure was exercised?

Time to Isolate the Change vs Number of Times Procedure was Exercised / Total Operational Executions (Plus Effort to Understand and Implement the Repair)			
	Time to Isolate the Change		
	< 1 hour	1 hour < 1 day	> 1 day
100%	hours	hours days	
90%			
80%		hours	days
70%			
60%			
50%	hours		
40%	minutes		
30%			
20%			
10%		hours	

Time to isolate the fault is related to time to understand and implement the fix.

Conclusions

Generated a method of comparing acceptance test and operational usage

Acceptance test is representative of operational usage except for the "mix" of statement types (at least in this study)

Structural coverage metrics may provide insight into software faults

Future Activities

The next study will attempt to reinforce the results of this study.

More faults and fault data

Larger, more representative NASA/SEL program

Exact order of acceptance test

36

N84 23143

ORIGINAL PAGE IS
OF POOR QUALITY

An Error-Specific Approach to Testing

Peter M. Valdes[1]

Amrit L. Goel[2]

Syracuse University

The main objective of software testing in the software development life cycle is to verify conformance of the implemented software with its intended requirements. Such requirements include

1. System requirements
2. Functional requirements
3. Programming requirements

Non-conformance with such requirements causes what are known as software errors.

Specificying an appropriate testing strategy to expose software errors is still an art. Traditional approaches do succeed in revealing many errors but none is powerful enough to expose all errors. The best that can be hoped for is to use a specific test strategy to expose a specific error type in specific program locations. It is this limitation that we exploit to develop a new approach to software testing which we call an error-specific testing (EST) strategy. It is in fact a dual to the traditional testing approaches.

---

[1]Research Assistant.

[2]Professor of Electrical & Computer Engineering, Syracuse University, Syracuse, NY 13210

The EST approach hypothesizes and tests on specific error-types in specified program locations. When applied to all error types of interest, it becomes powerful enough to satisfy the original objective of testing.

In the presentation we give highlights of the EST approach. Then we show how such an approach can be used to expose errors in a simple program, triangle. The material presented here is not meant to be self-contained. Mathematical results and other features (positive and negative) of this testing strategy are discussed in technical reports available from the authors. Further work on the use of this approach for determining software reliability (a different definition than commonly used) is also in progress and will be published in the near future.

An Error - Specific Approach to Testing

Peter M. Valdes

Amrit L. Goel

Syracuse University

Syracuse, N.Y 13210

OUTLINE

1. Testing
2. Error-Specific Testing (EST)
3. Related Work
4. EST Methodology
5. Assumptions and Limitations
6. EST of Triangle.
   - 6.1 Functional Requirements (FR_i's) Decomposition
   - 6.2 Structural Parts (SP_i's) Decomposition
   - 6.3 FR-SP Mapping
   - 6.4 Error Hypotheses
     - Function-Based Errors (EF's)
     - Structure-Based Errors (ES's)
   - 6.5 Test of Error Hypotheses
     - Function-Based Error Testing Strategy
     - Structure-Based Error Testing Strategy
   - 6.6 Recording Test Results in the FR-EF and SP-ES Matrices
7. Extensions of EST Philosophy

TESTING

- The main objective of testing is to verify conformance of the implemented software with its intended requirements such as
  - System requirements
  - Functional requirements
  - Programming requirements

- Non-conformance with intended requirement is known as a software-error.

Error-Specific Testing

- Traditional testing strategies can expose embedded software errors but none is powerful enough to expose all possible errors - therefore
  - use a specific strategy to expose a specific error type in specific program locations, i.e., Error Specific Testing (EST)

- EST is really a dual approach to traditional testing. When applied to all possible hypothesized errors, it becomes powerful enough to satisfy the original objective of software testing.

Error - Specific Testing

- Focuses on specific error types in specific locations
- Intuitively appealing and simple to use
- Number of test cases is bounded
- Can be automated
- Permits trade-offs in allocation of resources

	Traditional Software Testing	Error-Specific Testing (EST)
Specify	Testing strategy or strategies	Error-type in a specific program location and an appropriate testing strategy
Expose	Different types of software errors in various program locations	Specific error-types in the specified locations
Limitations	Not all possible errors can be exposed	Only the specified error (and some incidential errors) is exposed. However, it can be used to expose all errors if all these errors are tested for existence using appropriate testing strategies.

ORIGINAL PAGE IS
OF POOR QUALITY

RELATED WORK

Traditional

Use of non-error specific test
Strategies, e.g., path testing,
cause-effect graphing

Weyuker and Ostrand

Introduced error-based testing which uses all available information in exposing certain types of errors.

Howden

Realized the limitations of traditional test strategies but used them to expose certain types of errors (weak mutation).

Clark, et al.

Used the notion of error-sensitive-testing.

EST METHODOLOGY

1. Determine s/w functional requirements (FR_i's).
2. Decompose code into structural parts (SP_j's).
3. Hypothesize specific error types of interest for each FR_i and SP_j.
4. Specify EST strategy for error types in (3).
5. Determine test requirements for each EST strategy.
6. Optimize test requirements.
7. Generate test cases from the optimized test requirements.
8. Execute test cases, debug exposed errors, retest the changed code including affected code.

ASSUMPTIONS/LIMITATIONS

- FUNCTIONS REQUIREMENTS ARE CORRECT
- EST STRATEGY AVAILABLE FOR EACH HYPOTHESIZED ERROR TYPE
- NEED TO TEST FOR EACH HYPOTHESIZED ERROR TYPE

Error-Specific Testing of TRIANGLE

I. Functional Requirements Decomposition

		Description
FR_1	IF	$\sim(A \geq B \geq C)$ then not Δ
FR_2	IF	$(A = B = C)$ then equilateral Δ
FR_3	IF	$(A = B > C$ or $A > B = C)$ then Isosceles Δ but not equilateral Δ
FR_4	IF	$(A > B > C$ and $A^2 = B^2 + C^2)$ then right Δ
FR_5	IF	$(A > B > C$ and $A^2 > B^2 + C^2)$ then obtuse Δ and $A < B + C$
FR_6	IF	$(A > B > C$ and $A^2 < B^2 + C^2)$ then acute Δ

Code

```
Statement #

 0          procedure TRIANGLE (A, B, C)
 1          if A ≥ B go to 1
 2          go to 2
 3     1    if B ≥ C go to 3
 4     2    Print ('Illegal Input') return
 5     3    if A = B go to 4
 6          if B = C go to 4
 7          A : = A * A
 8          B : = B * B
 9          C : = C * C
10          D : = B + C
11          if A ≠ D go to 5
12          Print ('Right Δ') return
13     5    if A < D go to 6
14          Print ('Obtuse Δ') return
15     6    Print ('Acute Δ') return
16     4    if A = B go to 7
17          go to 8
18     7    if B = C go to 9
19     8    Print ('Isosceles Δ') return
20     9    Print ('Equilateral Δ') return
21     end procedure
```

II. Structured Parts Decomposition

	Statement Number (see TRIANGLE code)
SP_1	1
SP_2	2
SP_3	3
SP_4	4
SP_5	5
SP_6	6
SP_7	7,8,9,10
SP_8	11
SP_9	12
SP_{10}	13
SP_{11}	14
SP_{12}	15
SP_{13}	16
SP_{14}	17
SP_{15}	18
SP_{16}	19
SP_{17}	20

III. Functional Requirement - Structured Parts Mapping

$$
\begin{array}{c|ccccccccccccccccc|}
 & SP_1 & SP_2 & SP_3 & SP_4 & SP_5 & SP_6 & SP_7 & SP_8 & SP_9 & SP_{10} & SP_{11} & SP_{12} & SP_{13} & SP_{14} & SP_{15} & SP_{16} & SP_{17} \\
\hline
FR_1 & 1 & 1 & 1 & 1 & & & & & & & & & & & & & \\
FR_2 & 1 & & 1 & & 1 & & & & & & & & 1 & & 1 & & 1 \\
FR_3 & 1 & & 1 & & 1 & 1 & & & & & & & 1 & 1 & 1 & 1 & \\
FR_4 & 1 & & 1 & & 1 & 1 & 1 & 1 & 1 & & & & & & & & \\
FR_5 & 1 & & 1 & & 1 & 1 & 1 & 1 & & 1 & 1 & & & & & & \\
FR_6 & 1 & & 1 & & 1 & 1 & 1 & 1 & & 1 & & 1 & & & & & \\
\hline
\end{array}
$$

IV. Error Hypotheses

Functional-based Errors (EF's)

EF_1	Non-satisfaction of FR_1 (i.e. program not catching an illegal input)
EF_2	Non-satisfaction of FR_2
EF_3	Non-satisfaction of FR_3
EF_4	Non-satisfaction of FR_4
EF_5	Non-satisfaction of FR_5
EF_6	Non-satisfaction of FR_6

Structure-based Errors (ES's)

$ES_{1.1}$, $ES_{3.1}$, $ES_{5.1}$, $ES_{6.1}$, $ES_{8.1}$, $ES_{13.1}$, $ES_{15.1}$, $ES_{10.1}$	Incorrect relational operator
Note for subscript notation:	Left of dot gives structure part number when error is possibly embedded. Right of dot gives error number for the given structured part.
$ES_{1.2}$, $ES_{2.1}$, $ES_{5.2}$, $ES_{6.2}$, $ES_{8.2}$, $ES_{13.2}$, $ES_{14.1}$, $ES_{15.2}$, $ES_{10.2}$	Incorrect transfer of control flow.
$ES_{7.1}$	Incorrect Arithmetic Operator
$ES_{7.2}$	Incorrect Arithmetic Expression (Formula)
$ES_{7.3}$	Incorrect Assignment

ORIGINAL PAGE IS
OF POOR QUALITY

V. TEST OF ERROR HYPOTHESES

Function-based Error Testing Strategy

- Assume functional requirements given as

  If (input conditions) then (output conditions)

- Generate test requirements for every <u>valid</u> and <u>invalid</u> combination of the inputs

	Input Condition	Valid Combination	Invalid Combination
FR_1	$\sim(A \geq B \geq C)$	$(A < B) \wedge (B \geq C)$ $(A \geq B) \wedge (B < C)$ $(A < B) \wedge (B < C)$	$(A > B) \wedge (B > C)$ $(A = B) \wedge (B = C)$ $(A = B) \wedge (B > C)$ etc.
FR_2	$(A = B = C)$	$(A = B) \wedge (B = C)$	$(A \neq B) \wedge (B = C)$ $(A = B) \wedge (B \neq C)$ $(A \neq B) \wedge (B \neq C)$
FR_3	$(A = B \quad C$ <u>or</u> $A > B = C)$	$(A = B) \wedge (B > C)$ $(A > B) \wedge (B = C)$	$(A > B) \wedge (B \neq C)$ $(A \neq B) \wedge (B = C)$ etc.
FR_4	$(A > B > C$ <u>and</u> $A^2 = B^2 + C^2)$	$(A > B) \wedge (B > C) \wedge$ $(A^2 = B^2 + C^2)$	$(A > B) \wedge (B > C) \wedge$ $(A^2 \neq B^2 + C^2)$ etc.
FR_5	$(A > B > C$ <u>and</u> $A^2 > B^2 + C^2$ <u>and</u> $A < B + C)$	$(A > B) \wedge (B > C) \wedge$ $(A^2 > B^2 + C^2)$ $(A < B + C)$	$(A > B) \wedge (B > C) \wedge$ $(A^2 > B^2 + C^2)$ $(A > B + C)$ etc.
FR_6	$(A > B > C$ <u>and</u> $A^2 < B^2 + C^2)$	$(A > B) \wedge (B > C) \wedge$ $(A^2 < B^2 + C^2)$	$(A > B) \wedge (B > C) \wedge$ $(A^2 > B^2 + C^2)$ etc.

Structure-based Errors Testing Strategy:

Structural Part	**Testing Strategy**
Incorrect Relational Operator	
Simple relational expression (SRE) of the form $A < B$	Test Cases: $A = B$, $A > B$
SRE of the form $A \leq B$	Test Cases: $A < B$, $A = B$
SRE of the form $A = B$ or $A \neq B$	Test Cases: $A = B$, $A \neq B$
SRE of the form $A \geq B$	Test Cases: $A = B$, $A > B$
SRE of the form $A > B$	Test Cases: $A < B$, $A = B$

Incorrect Construct in a SRE	
SRE of the form $A < k$ where k = constant	Test Cases: $A = k$, $A^* < k$ where $A^* = \max\{\text{domain of } A\}$
SRE of the form $A \leq k$	Test Cases: $A = k$, $A_* > k$ where $A_* = \min\{\text{domain of } A\}$
SRE of the form $A = k$ or $A = k$	Test Cases: $A = k$
SRE of the form $A \geq k$	Test Cases: $A = k$, $A^* < k$
SRE of the form $A > k$	Test Cases: $A = k$, $A_* > k$

Incorrect Relational Operator and Constant	
SRE of the form $A < k$	Test Cases: $A^* < k$, $A = k$, $A_* > k$, $(A < A^*) \wedge (A^* < k)$
SRE of the form $A \leq k$	Test Cases: $A^* < k$, $A = k$, $A_* > k)$ $(A > A_*) \wedge (A_* > k)$
SRE of the form $A = k$	Test Cases: $A^* < k$, $A = k$, $A_* > k$, $(A < A^*) \wedge (A^* < k)$
SRE of the form $A \geq k$	Test Cases: $A^* < k$, $A = k$, $A_* > k$, $(A < A^*) \wedge (A^* < k)$
SRE of the form $A > k$	Test Cases: $A^* < k$, $A = k$, $A_* > k$, $(A > A_*) \wedge (A_* > k)$

Testing of TRIANGLE's ES's

Hypothesized Error	Testing Strategy
$ES_{1.2}$, $ES_{2.1}$, $ES_{5.2}$,	Simple traversal of go to statement
$ES_{6.2}$, $ES_{8.2}$, $ES_{13.2}$,	
$ES_{13.2}$, $ES_{14.1}$, $ES_{18.2}$	
$ES_{10.2}$	
$ES_{1.1}$	Test Cases: $(A = B)$, $(A > B)$
$ES_{3.1}$	$(B = C)$, $(B > C)$
$ES_{5.1}$	$(A = B)$, $(A \neq B)$
$ES_{6.1}$	$(B = C)$, $(B \neq C)$
$ES_{8.1}$	$(A^2 = B^2 + C^2)$, $(A^2 \neq B^2 + C^2)$
$ES_{10.1}$	$(A^2 = B^2 + C^2)$, $(A^2 > B^2 + C^2)$
$ES_{13.1}$	$(A = B)$, $(A > B)$
$ES_{15.1}$	$(B = C)$, $(B > C)$
$ES_{7.1}$	Simple traversal of statements 7, 8, 9, 10
$ES_{7.2}$	Simple traversal of statements 7, 8, 9, 10
$ES_{7.3}$	Simple traversal of statements 7, 8, 9, 10

ORIGINAL PAGE IS
OF POOR QUALITY

VI. Recording Test Results in the FR-EF and SP-ES Matrices

Let M_{FR-EF} = element of FR-EF matrix

M_{SP-ES} = element of SP-ES matrix

Then, assuming we have a sufficient error-based strategy

$$M_{FR-EF} \text{ or } M_{SP-ES} = \begin{cases} 0 & \text{If test result is negative} \\ 1 & \text{If test result is positive} \end{cases}$$

If error-based strategy is imperfect

$$M_{FR-EF} \text{ or } M_{SP-ES} = \begin{cases} rdi & \text{If test result is negative but test's relative degree of imperfection is rdi} \\ 1 & \text{If test result is positive} \end{cases}$$

FR-EF Matrix

$$\begin{array}{cc} & \begin{array}{cccccc} EF_1 & EF_2 & EF_3 & EF_4 & EF_5 & EF_6 \end{array} \\ \begin{array}{c} FR_1 \\ FR_2 \\ FR_3 \\ FR_4 \\ FR_5 \\ FR_6 \end{array} & \begin{bmatrix} \ & \ & \ & \ & \ & \ \\ \ & \ & \ & \ & \ & \ \\ \ & \ & \ & \ & \ & \ \\ \ & \ & \ & \ & \ & \ \\ \ & \ & \ & \ & 1 & \ \\ \ & \ & \ & \ & \ & \ \end{bmatrix} \end{array}$$

SP-ES Matrix

$$\begin{array}{cc} & \begin{array}{ccccc} \left\{\begin{array}{c} ES_{1.1} \\ ES_{3.1} \\ \vdots \\ ES_{15.1} \end{array}\right\} & \left\{\begin{array}{c} ES_{1.2} \\ ES_{2.1} \\ \vdots \\ ES_{15.2} \end{array}\right\} & ES_{7.1} & ES_{7.2} & ES_{7.3} \end{array} \\ \begin{array}{c} SP_1 \\ SP_2 \\ \\ \\ \\ SP_{17} \end{array} & \begin{bmatrix} \ & \ & \ & \ & \ \\ \ & \ & \ & \ & \ \\ \ & \ & 0 & \ & \ \\ \ & \ & \ & \ & \ \\ \ & \ & \ & \ & \ \\ \ & \ & \ & \ & \ \end{bmatrix} \end{array}$$

EXTENSIONS OF EST PHILOSOPHY

- MEASURE OF COMPLEXITY
- MEASURE OF CORRECTNESS
- TRADE-OFF STUDIES FOR ALLOCATION OF RESOURCES

D7

ORIGINAL PAGE IS
OF POOR QUALITY

N84 23144

Testing and Error Analysis of a Real-Time Controller

C. G. Savolaine

Bell Laboratories
Holmdel, New Jersey 07733

1. INTRODUCTION

This paper outlines inexpensive ways to organize and conduct system testing that were used on a real-time satellite network control system. This system contains roughly 50,000 lines of executable source code developed by a team of eight people. For a small investment of staff, the system was thoroughly tested, including automated regression testing, before field release.

Detailed records were kept for fourteen months, during which several versions of the system were written. A separate testing group was not established, but testing itself was structured apart from the development process. The errors found during testing are examined by frequency per subsystem by size and complexity as well as by type. The code was released to the user in March, 1983. To date, only a few minor problems have been found with the system during its pre-service testing and user acceptance has been good.

2. THE SYSTEM BEING TESTED

The Satellite Network Control System (SNCS) is a real-time, mini-computer based, call-processing system developed for Picturephone[R] Meeting Service (PMS). It controls the switching of both 1.5 and 3.0 Mb/s digital circuits over a satellite using Frequency Division Multiple Access (FDMA) technology. The SNCS runs on a dedicated Western Electric 3B-20S computer (similar in capacity to a DEC VAX 11/780) and supports interfaces to:

1. Earth stations
2. A customer reservations system
3. A satellite maintenance center
4. A computer operator console

Satellite connectivity requests are sent to the SNCS, which verifies these requests and assigns satellite transponder channels to each. Every 15 minutes commands are generated and sent to microprocessors located in the earth stations that tune the modems. The real-time control interface to the microprocessors is complicated by inter-dependencies among the commands across earth stations. To compensate, a sequencing is generated by the SNCS for the commands, which changes with every reconfiguration. The central SNCS multiplexes these earth station work lists and simultaneously distributes them to the stations, maintaining this sequencing.

3. TESTING METHODOLOGY

A prototype of the system was available in February, 1982. It needed significant enhancement to provide full service, and it had not been thoroughly tested. The methods used in testing the system while new versions were being developed concurrently are described

ORIGINAL PAGE IS
OF POOR QUALITY

here. The next section will evaluate their usefulness.

The major techniques used were:

- tester selected from the development team
- rotation of testing assignment
- testing was automated
- formal testing of all versions
- careful tracking of error causes and effort to correct
- deferring correction of low severity errors
- full regression testing
- releasing test cases to user with code

A person from the development team was assigned the full-time task of creating and organizing test cases. The system was divided into subsystems and test cases were created consisting of multiple test situations per case. Each test case had the objective of testing a particular system feature. The running of all cases was automated with a difference program used on the output to isolate potential errors. This made full regression testing possible. This testing was done on each version even though only the last version was released to the field.

The testing assignment was rotated among the group, changing with each of the three versions created. Tests were automated and conducted by the tester, but problems, after being given a severity code, were assigned to individuals in the development team. The correction of bugs having a low severity was deferred to the next version to avoid correcting multiple versions.

Each error was classified into one of three types: omission, commission or requirements. The errors due to a requirements misunderstanding often stimulated additional documentation to clarify the mis-conception. The system was divided into nine subsystems, and each error was allocated to one of these. The subsystems and their errors were then analyzed verses code size and complexity as determined by the McCabe complexity measure. [7]

For every error, the time was recorded to find the cause, to fix it, and to test it. In addition, the number of iterations through the cycle and whether other errors were caused or found in the process was recorded.

Test cases were released to the user along with the code. This provided a foundation for their testing efforts as well as serving as detailed documentation. By running known good cases in the users environment, problems unique to their configuration could be identified quickly.

ORIGINAL PAGE IS
OF POOR QUALITY

4. RESULTS

By devoting 15% of the available resources to testing, an extensive set of test cases was created and automated. By rotating the person responsible for testing, the testing process became more robust and independent of individual traits. It also made a task that was perceived to be onerous more palatable. The training investment in rotating the testing position was low, since each tester was previously in the development team.

By automating the running and output comparison of the test cases, it was easy to run regression tests on a system that was growing and changing. The number of test cases grew steadily. Phased fixes were manageable because the less critical errors were the ones being deferred, and none were delayed more than a few months.

The bugs correlated well with subsystem complexity and lines of code. Nearly half the errors were attributed to omission. Of these, half occurred in the two largest and most complex subsystems. About half the errors required only a one line correction. For the first version, the time to find an error and to correct it were equal. For later versions it took longer to find the errors than to fix them.

Inviting the user to participate in generating and reviewing the test cases made it possible to gain early user involvement. Releasing the test cases with the code gave the user an extensive set of test cases upon which to build, and served as examples for user training.

Thus, by formalizing and automating the testing process a thoroughly tested, stable system, plus test cases were delivered to the user on schedule.

References

1. McCabe, Thomas J., "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December, 1976, pp. 308-320.

2. Metzger, P.W., Managing a Programming Project, Prentice-Hall, Inc., 1981.

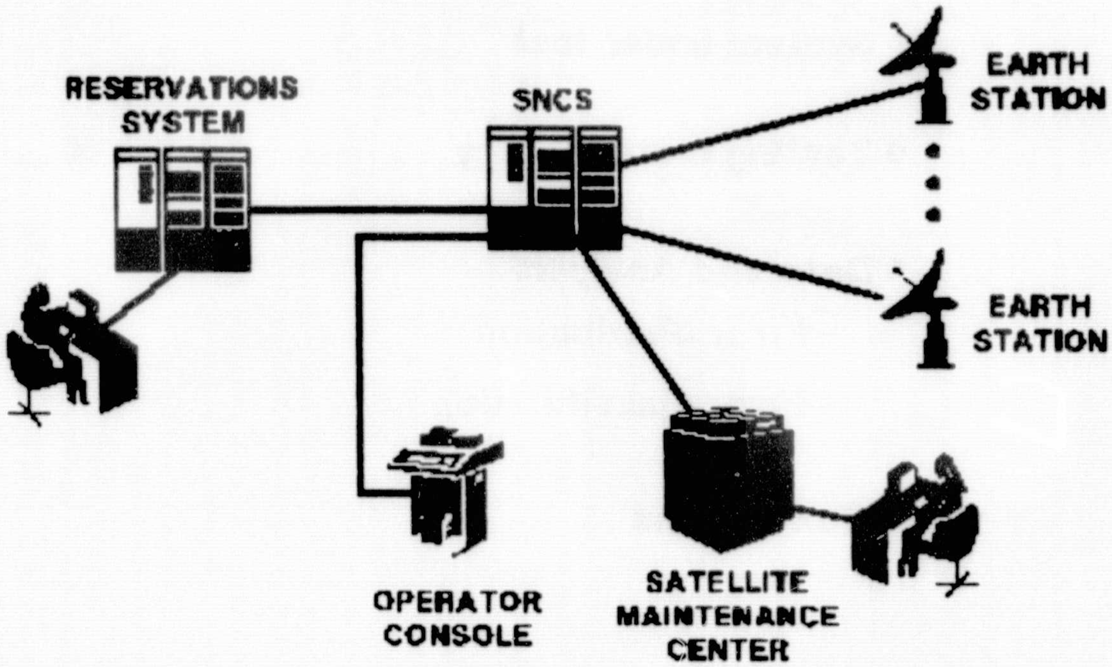
3. Myers, G.J., The Art of Software Testing, John Wiley & Sons, Inc., 1979.

TESTING AND ERROR ANALYSIS OF A REAL-TIME CONTROLLER

- System under test
- Testing methodology
- Data and Analysis
  - Error distribution
  - Error classification
- Conclusions

C. Savolaine
Bell Labs

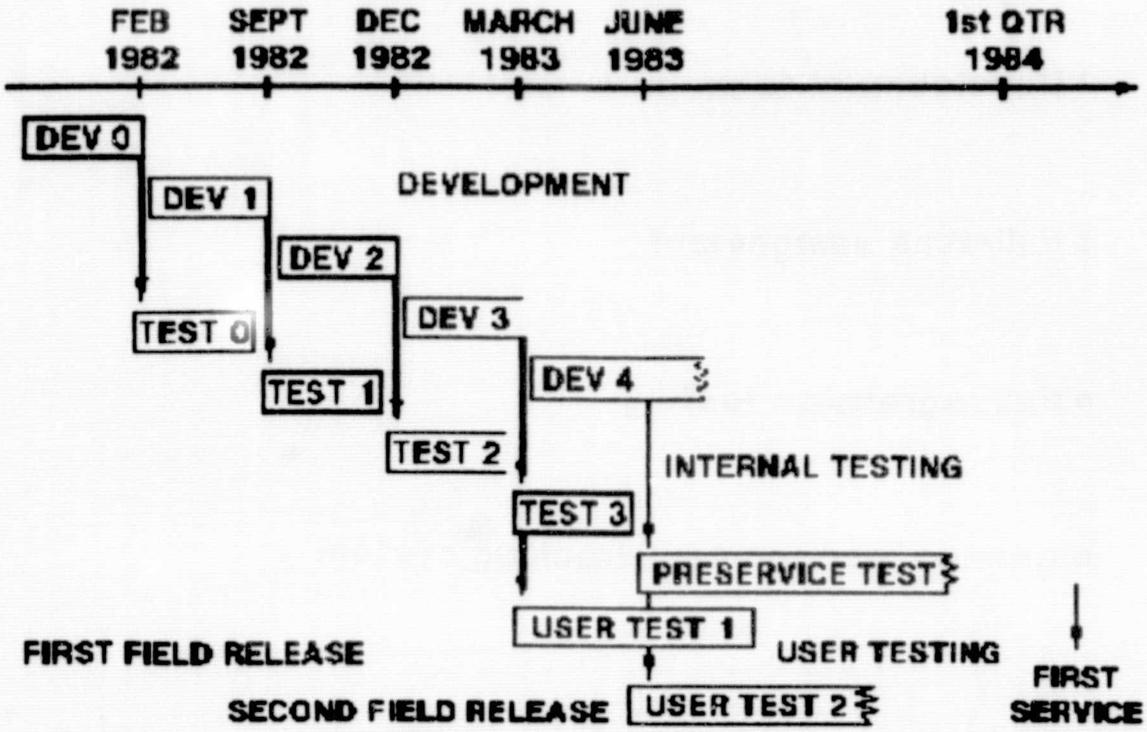
SATELLITE NETWORK CONTROL SYSTEM



TESTING METHODOLOGY

- Development team personnel
- Full-time assignment
- Full regression testing
- Change management tracking system

DEVELOPMENT/TESTING CYCLE



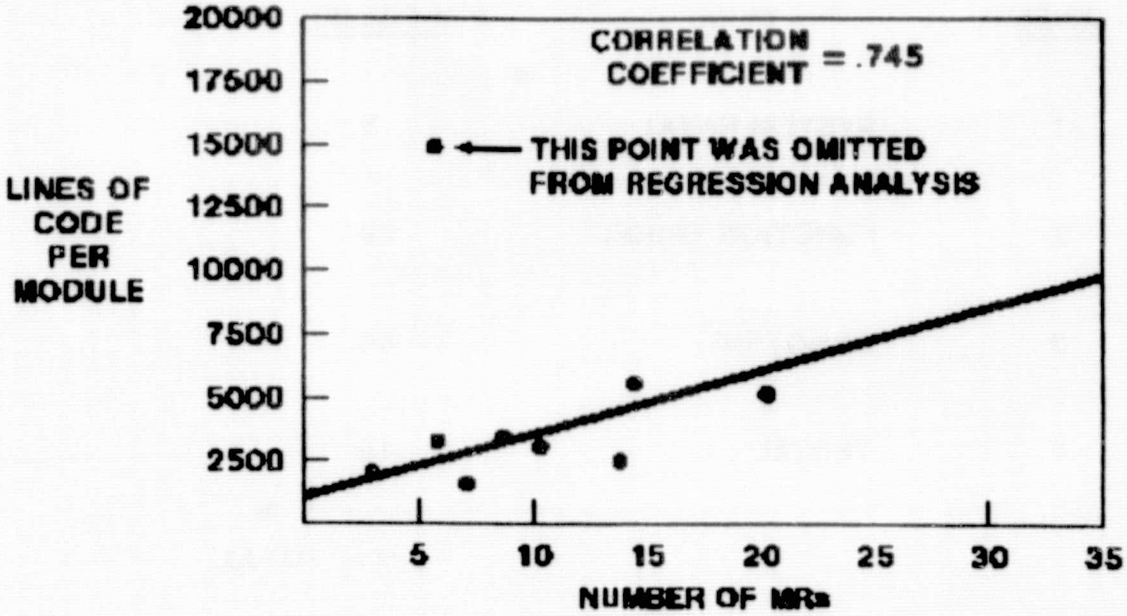
C. Savolaine
Bell Labs

ERROR SEVERITY

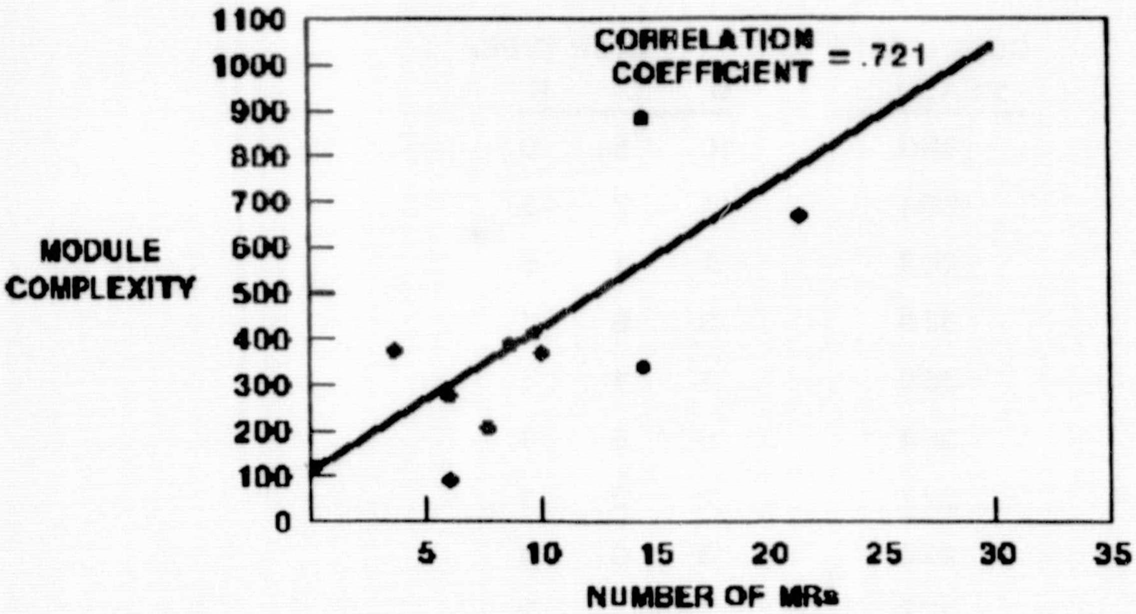
LEVEL	TYPE	# FOUND
1	SYSTEM FATAL	5
2	FUNCTION ERROR	29
3	ANNOYING	44
4	TRIVIAL	19
		91 TOTAL

ORIGINAL PAGE IS
OF POOR QUALITY

LINES OF CODE VERSUS MRs



COMPLEXITY VERSUS MRs



COMPLEXITY VERSUS ERROR TYPE

COMPLEXITY MEASURE	ERROR TYPE: O	C	R
880	10	5	0
664	11	7	3
403	2	1	6
379	3	6	1
369	1	1	1
364	6	5	3
277	2	3	1
226	3	0	4
107	3	1	2

41 + 29 + 21 = 91

ORIGINAL PAGE IS
OF POOR QUALITY

COMPLEXITY VERSUS ERROR TYPE

COMPLEXITY MEASURE	INTERNAL TESTING ERROR TYPE: O	C	R		USER TESTING ERROR TYPE: O	C	R	
880	10	5	0	= 36	1	4	4	= 20
664	11	7	3		3	7	1	
403	2	1	6		-	-	-	
379	3	6	1		-	-	1	
369	1	1	1		-	-	-	
364	6	5	3		1	2	-	
277	2	3	1		-	2	-	
226	3	0	4		-	-	-	
107	3	1	2		-	-	-	
	41 +	29 +	21	= 91	5 +	15 +	6	= 26

COMPLEXITY VERSUS FATAL ERRORS

MODULE	COMPLEXITY MEASURE	FATAL ERRORS
1	880	
2	664	
3	403	
4	379	
5	369	1
6	364	
7	277	2
8	226	1
9	107	1

COMPLEXITY VERSUS FATAL ERRORS

MODULE	COMPLEXITY MEASURE	FATAL ERRORS
1	880	1
2	664	
3	403	
4	379	
5	369	1
6	364	1
7	277	2
8	226	1
9	107	1
		7

RESULTS

- Fatal errors occurred in less complex modules
- Non-fatal errors correlated well with complexity
- ~~Most~~ errors found in pre-field testing were omission type
- Most errors found in field testing were comission type

CONCLUSIONS

- Avoid complex modules
- In design phase, inspect for omission errors
- In internal testing, look for comission errors

PANEL #3

HUMAN FACTORS

E. Connelly, Performance Measurement Associates
E. Soloway, Yale University
C. Grantham, University of Maryland

D8

N84 23145

TRANSFORMATIONS OF SOFTWARE DESIGN AND CODE MAY LEAD TO REDUCED ERRORS

Edward M. Connelly

Performance Measurement Associates, Inc.
Vienna, Virginia 22180

ABSTRACT

This research investigated the capability of programmers and non-programmers to specify problem solutions by developing example-solutions and also for the programmers by writing computer programs; each method of specification was accomplished at various levels of problem complexity. The level of difficulty of each problem was reflected by the number of steps needed by the user to develop a solution. Machine processing of the user inputs permitted inferences to be developed about the algorithms required to solve a particular problem. The interactive feedback of processing results led users to a more precise definition of the desired solution.

Two participant groups (programmers and bookkeepers/accountants) working with three levels of problem complexity and three levels of processor complexity were used. The experimental task employed in this study required specification of a logic for solution of a Navy task force problem. This task involved choosing ships from a ship list which identified the ship type, the transiting time (the time required for the ship to get from its present position to the desired site), and stationing time (the number of days the ship can remain on station with available provisions). In addition to this specification of ship combinations the participants had to specify by the example-solution the range of transiting and stationing times required. In another related experiment, participants developed FORTRAN IV code to solve the same problems.

The performance both of programmers and non-programmers was found to decrease with increasing levels of problem complexity and with reduced processor support. For both the groups, errors of commission were relatively infrequent compared to errors of

omission. It was found that the degree of processor complexity was much more influential than problem complexity in predicting performance scores. When little computer generalization of user input was provided, performance was significantly lower than during all other experimental conditions. Results also showed that participant-strategy in the generation of problem solutions was a significant factor in performance, though years of experience and years of education were not found to be good predictors of performance. The feedback aids were shown to be most effective when they included the logic implied by the example-solutions. These experiments demonstrate the effectiveness of the on-line use of computer software to create and modify software routines.

Results also suggest that a measure for evaluating a programmer's skill should involve evaluation of procedure that programmers use in developing example-solutions, and in designing and writing program code. Finally, the superiority of using example-solutions with inductive feedback over writing code suggests that the transformation process provided by the induction might be applied analogously to software development. Considering designs and code in multiple transformed forms may reduce software errors to a level found for example-solutions.

INTRODUCTION

Six experiments were conducted, with the same problems used in all experiments. The ability of the participants to develop example-solutions was evaluated as a function of the participant's background and experience, the complexity of the problem to be solved, and the level of processing provided by the computer, and the level of feedback aids, when aids were available.

Experiments 1 and 2 were designed to investigate the ability of expert programmers and of bookkeepers/accountants who were not expert programmers to develop example-solutions for the hypothetical Navy task force problem. The experimental variables for both experiments were problem complexity and processor complexity, i.e., the amount of machine processing of user inputs.

Experiments 3 and 4 were designed to investigate the ability of expert programmers and non-programmers to develop accurate and complete example-solutions using various feedback aids at various levels of problem complexity. The feedback aid designs were based on the results of Experiments 1 and 2, where the systematic generation of example-solutions, as measured by a combinational measure, had been shown to be highly correlated with performance (explaining 63% of the score variance).

Experiment 5 was designed to investigate the capability of expert programmers to revise problem solutions' specifications in the form of example-solutions in which various numbers of initially incorrect entries had been introduced, using the feedback-aids developed in Experiments 3 and 4.

Finally, Experiment 6 called upon expert programmers to develop computer code written in FORTRAN IV for various levels of data input - a design intended to be analogous to the design of Experiment 1. The results of Experiment 6 were sub-routines written in FORTRAN IV that should accept or reject a ship combination, as that combination was correct or incorrect.

The performance measures used in the experiments consisted of error measures and strategies measures. Three error measures were:

a. P_T, the probability that a given ship combination was correctly classified as acceptable or unacceptable.

b. P_C, the probability that a correct ship combination was accepted.

c. P_{IC}, the probability that an incorrect ship combination was rejected.

In addition to the error measures above, relative error measures were used. A relative error measure was defined as a participant's error score (P_T, P_C, P_{IC}) on an experimental problem minus his/her error score on the pretest problem. The relative error measures thus tended to remove the effect of the participant's innate capability, and, as a result, were more sensitive to experiment factors than were the error measures alone.

Two strategy measures were used to detect the frequency with which participants used specific strategies. One strategy measure, the combinational measure, detected the frequency with which a participant changed only one component at a time of each successive example-solution. Another strategy measure, a sequence measure, detected the use patterns of the various feedback aids.

Results of Experiments 1, 2, and 3 in which programmers and bookkeepers/accountants provided example-solutions are compared with the results of Experiment 6 where experienced programmers wrote FORTRAN IV program code for the same problems. Results of the other experiments can be found in Connelly (1982 a, b).

RESULTS OF EXPERIMENTS 1, 2, & 3

Processor Complexity and Error Reduction

First, as expected, more errors occurred during the work on the more complex problems. However, the level of processing, or generalization, of the example-solutions was found to be an important error reducing factor, i.e., a significant reduction in errors occurred when data from example-solutions were processed into a standard form and presented to the participant.

Systematic Strategies and Feedback-Aids

A second result, and perhaps the most important, was that participants in both categories who performed well tended to use a systematic, step-by-step strategy in selecting example-solutions. This result, together with the first, noted above, suggested that feedback aids might be designed to encourage participants to use systematic strategies, by processing their example-solutions and then feeding back the resultant data to suggest possible additional inputs. A description of the aid design results obtained in using them are given in Connelly (1982 a, b).

Breadth vs. Depth of Experience

A third result of the first two experiments applied to the subsequent experiments was that the number of years advanced education (i.e., beyond high school) and the number of years of professional experience were found to be relatively unimportant factors in predicting performance.

The lack of a strong predictive relationship between years of higher education or years of experience and performance may come as a surprise to educators and directors of personnel departments. This result was found in all of the experiments, so that very strong evidence is available to support the assertion that years of education and relevant work experience are not good predictors of problem-solving performance. Additional results suggest that the "number of programming languages (used on 1 or more programs)" and "number of operating systems used" are better predictors of the capabilities of computer users/programmers.

Low Frequency of Errors-of-Commission

The fourth result applied to the subsequent experiments was the observation that only a few errors of commission occurred during the generation of the example-solutions. The majority of errors that did occur were errors of omission. This intriguing result influenced the design of Experiment 6, where FORTRAN IV code was written to solve the same problems used in Experiment 1, so that a comparison of error rates would be possible.

RESULTS FOR EXPERIMENT 6

Two types of errors were analyzed. One type, termed an "error of omission", referred to an error that resulted in a failure to accept a correct entity (e.g., ship combination). When specifying a problem solution with example-solutions, an error of omission could be directly traced to a failure to enter an example of a suitable entity (ship combination). The second type of error considered was an "error of commission." When example-solutions were used to specify a problem solution, an error of commission corresponded to an incorrect example entered into the processor which was then treated by the processor as a correct example. An error of commission resulted in erroneously accepting incorrect entities (ship combinations).

Errors-of-Omission

There was little difference in the effect of problem complexity on errors of omission between the two methods of specifying problem solutions, i.e., by example-solution or by FORTRAN IV subroutines.

Errors-of-Commission

When generating example-solutions without feedback aids, the rate of errors of commission increased sharply at a problem complexity-level near 20,821, as measured by Halstead's E Metric (Connelly, Comeau, & Johnson 1981). But, given a suitable feedback aid environment, such as in Experiment 3, this problem complexity limitation could be eliminated, as evidenced by the Experiment 3 data in which performance degradation did not appear.

The most important result regarding errors of commission was that specification by example-solutions was superior to specification by program code. Analysis of the mean scores from Experiments 1, 2, and 3 provided strong evidence that using example-solutions substantially reduced errors of commission compared to using FORTRAN IV program code. The 3% rate for errors of commission with example-solutions compared favorably with 18% for program code.

Three hypotheses concerning the superior performance of the example-solution method seem plausible:

1. It was working with examples and dealing with each individual combination of items one-at-a-time that resulted in a low rate of errors of commission.

2. It was the specification of each combination one-at-a-time that alone was important. Consequently, if computer programs were developed to specify each solution combination one-at-a-time, the rate of errors of commission would be low.

3. The success of the example-solution method was due, in part, to the transformation of example-solutions from one logic form into another, such as the ship selection logic (SSL), or into several different forms, such as the feedback aids. Thus, it was the transformation of logic which enabled the user to view the problem in more than one way and that resulted in a low rate of errors of commission. Consequently, if program code entered by the user were transformed into a different logic form and fed back to the user for approval, a low rate of errors of commission would be obtained.

These hypotheses are not alternative hypotheses - all could be true. We have strong evidence that the first hypothesis is true. If the second is true but not the third, program design and coding methods could be adapted to a more combination dependent structure. And finally, if the third hypothesis were found to be true, pre-compilation aids could be designed to convert the user's program code into another form (while maintaining the same program logic) for feedback to the user.

CONCLUSIONS

1. The lack of a strong relationship between "years of higher education", "years of experience" and performance, coupled with the strong relationship between "number of computer languages" known and "number of operating systems" used, suggests that education and experience should not be used as they have been in the past for hiring, promoting, determining salary level, and assigning tasks. Instead, the number of operating systems used, which are better performance predictors, should be used until the underlying factors included in each are discovered.

2. Apparently, the depth of an individual's experience is not as important to performance as is breadth of his experience.

3. A possible common underlying experience related factor is the ability to view problems from alternative viewpoints, or the ability to develop alternative approaches to problems - an ability that might be enhanced with feedback aids.

4. The performance prediction capability of strategy measures, developed as moment-to-moment measures, not only clearly demonstrates that systematic strategies were used by successful participants (which led to the design of the feedback aids), but also convincingly demonstrates that moment-to-moment measures provide the sensitivity to explain considerable performance variance (approximately 60% in Experiments 1 thru 4.)

5. The superior performance (fewer errors of commission) achieved when using example-solutions and inductive processing to specify problem solutions over the performance achieved when using FORTRAN IV code may provide a basis for determining the underlying mechanism for that success and a means for incorporating that mechanism into program designing and coding aids. Apparently, superior performance was obtained either because each combination of the input variables was treated individually and/or because the example-solutions were transformed into another logic form -- the ship selection logic (SSL). If the former is a significant factor, then aids described in this report should be

adapted to program designing and coding aids. If the latter is a significant factor then designing and coding aids should be developed to transform the logic provided by the user into another form which is then fed back to the user for his review. Such a transformation might present the program's equivalent logic.

REFERENCES

Connelly, E. M. A comparison of the accuracy and completeness of problem solutions produced by example-solutions and program code. (Technical Report 82-362). Performance Measurement Associates, Inc. September 1982 (a).

Connelly, E. M. Accuracy & completeness of problem solutions with example-solutions. (Technical Report 82-363). Performance Measurement Associates, Inc. November 1982 (b).

Connelly, E. M., Comeau, R. F., & Johnson, P. Effect of automatic processing on specification of problem solutions for computer programs. (Technical Report 81-361). Performance Measurement Associates, Inc. March 1981. AD A108570

Halstead, M. H. Elements of software science. New York: Elsevier, 1977.

The research reported here was supported by the Engineering Psychology Programs, Office of Naval Research. The views, opinions and findings are those of the author and should not be construed as an official Department of the Navy position, policy, decision.

PROGRAMMING VIA EXAMPLE-SOLUTION CAN RESULT IN FEWER ERRORS

EDWARD M. CONNELLY

PERFORMANCE
MEASUREMENT
ASSOCIATES,
Incorporated

RESEARCH METHOD

TEST ABILITY OF INDIVIDUALS

TO SPECIFY PROBLEM SOLUTIONS:

- EXAMPLE SOLUTIONS
- FORTRAN IV CODE

SIX EXPERIMENTS

ORIGINAL EXAMPLE SOLUTIONS

1. PC/IR, PROGRAMMERS
2. PC/IR, BOOKKEEPERS
3. PC/FA, PROGRAMMERS
4. PC/FA, BOOKKEEPERS

REVISE EXAMPLE SOLUTIONS

5. PC/FA, PROGRAMMERS

FORTRAN IV CODE

6. PC/IR, PROGRAMMERS

PC = PROBLEM COMPLEXITY

IR = INFORMATION REQUIRED

FA = FEEDBACK AIDS

PROBLEM STATEMENT

1. THE SHIPS NEEDED FOR THE TASK FORCE ARE:
   - 2 AIRCRAFT CARRIERS — NUCLEAR (CVAN) OR NON-NUCLEAR (CVA)

     AND
   - 2 SUBMARINES (SS)

2. THE TRANSITING TIME MUST BE 5 DAYS OR LESS AND

3. THE STATIONING TIME MUST BE 10 DAYS OR MORE.

THIS TASK FORCE CRITERIA SPECIFIES THREE COMBINATIONS OF SHIP TYPES AS FOLLOWS:

- 2 CVA AND 2 SS

  OR

- 2 CVAN AND 2 SS

  OR

- 1 CVA AND 1 CVAN AND 2 SS

SHIP SELECTION LOGIC (SSL)

Ship Type	No. of Ship Type	Transit Time MIN	Transit Time MAX	Stationing Time MIN	Stationing Time MAX
CVAN	0				
CVA	1	1	5	10	50
CA	0				
CGN	0				
CG	0				
DD	0				
SSN	0				
SS	2	1	5	10	50
AO	0				
TOTAL:	3				

DEMOGRAPHICS

- YEARS OF EXPERIENCE AND YEARS OF HIGHER EDUCATION ARE NOT IMPORTANT TO PREDICTING PERFORMANCE.

- NUMBER OF COMPUTER LANGUAGES KNOWN AND NUMBER OF OPERATING SYSTEMS USED ARE IMPORTANT TO PREDICTING PERFORMANCE.

- UNDERLYING FACTOR MAY BE ABILITY TO VIEW PROBLEMS FROM ALTERNATIVE VIEWPOINTS.

DEMOGRAPHICS

- YEARS OF EXPERIENCE AND YEARS OF HIGHER EDUCATION ARE NOT IMPORTANT TO PREDICTING PERFORMANCE.

- NUMBER OF COMPUTER LANGUAGES KNOWN AND NUMBER OF OPERATING SYSTEMS USED ARE IMPORTANT TO PREDICTING PERFORMANCE.

- UNDERLYING FACTOR MAY BE ABILITY TO VIEW PROBLEMS FROM ALTERNATIVE VIEWPOINTS.

EXAMPLE SOLUTIONS/
FORTRAN IV CODE

- EXAMPLE SOLUTIONS AND FEED-BACK AIDS YIELDS SAME ERROR OF OMISSION RATE AS FORTRAN IV PROGRAMS

- EXAMPLE SOLUTIONS AND FEED-BACK AIDS YIELD MUCH LOWER RATE OF ERROR OF COMMISSION AS FORTRAN IV PROGRAMS

	ERRORS OF COMMISSION
EXAMPLE SOLUTIONS PLUS INDUCTIVE FEEDBACK	3%
PROGRAM CODE	17.7%

HYPOTHESES

SUPERIOR PERFORMANCE OBTAINED WITH EXAMPLE SOLUTIONS MAY BE DUE TO:

- WORKING WITH EXAMPLES

  OR

- WORKING WITH EACH SOLUTION ONE-AT-A-TIME

  OR

- THE TRANSFORMATION FROM ONE FORM TO ANOTHER (EXAMPLES TO EQUIVALENT LOGIC)

SYMT

ORIGINAL PAGE IS
OF POOR QUALITY

EXTENDED ABSTRACT

"You can observe a lot by just watching"[1]
How Designers Design[2]

David Littman*, Kate Ehrlich*, Elliot Soloway*, John Black**

Department of Computer Science* Department of Psychology**
Yale University
New Haven, Connecticut 06520

(Please address all correspondence to Elliot Soloway)

1. Introduction: Motivation and Goals

Rather than developing design languages and support environments on the basis of what we think designers *should* be doing, we felt that a more informed process would be to first find out what they do do. To this end, we interviewed for two hours each 4 expert software designers and 2 novice designers as they designed an electronic mail system; subjects were encouraged to talk-aloud as they worked, and the design session was video-taped. Here we briefly summarize the key observations based on an analysis of these tapes.

2. Subjects and Task

All designers were professionals supplied to us by a nearby branch of ITT. Expert designers had at least 8 years of design experience in commercial settings, while novices had less than 2 years of similar experience. Note, however, that the novices were without question bright, competent individuals; they simply had less experience than the experts. Subjects were given the following task:

> ***TASK*** -- Design an electronic mail system around the following primitives: READ, REPLY, SEND, DELETE, SAVE, EDIT, LIST-HEADERS. The goal is to get to the level of pseudocode that could be used by professional programmers to produce a running program. The mail system will run on a very large, fast machine so hardware considerations are not an issue.

[1] Quote from Yogi Berra, a catcher for the New York Yankees.

[2] This work was sponsored by a grant from ITT.

3. Observation I: How the Design Progressed

All our expert designers considered the same topics, almost always in the same order, and usually at the same level of detail. This surprisingly consistent observation led us to posit the concept of a *session meta-plan*, which we believe guided the expert software designer's treatment of the electronic mail system. Novices did not seem to use anything analogous to a common plan of attack on the mail system problem: their design sessions were less systematic than those of the experts.

As illustrated in the time line shown below, the meta-plan of our experts contained five distinct phases: first the experts described how a user would view the mail system, then they stated various assumptions (e.g, we will use dumb terminals); then experts used models of mail systems at various levels of generality (e.g., at the most general level was the flow of information model, followed by examples of other mail systems they have known followed by the specific system at hand); finally, the experts worked on the concrete design. Notice that the novices dove right into the detailed specifications of the system. We asked all subjects to provide a wrap up evaluation at about 10 minutes before the end of the session.

```
          Start                                      Finish
NOVICES:  ............. concrete design.................wrap-up

                        ~100   mins                  ~10 mins

EXPERTS:  user....assump-....abstract.........concrete...wrap-up
          model   tions      models of          design
                             mail system

        ~10 mins ~10 mins          ~80 mins          ~10 mins
```

The following quotes taken from the protocols are representative and support the above claims:

At 3 minutes into the task, one novice said:

> (Writes SAVE) "To save I have to open a file and then write to that file... If I have 5 or 6 messages I have to consider if I want to save all of them or whether I should save a specific one and specify which one I am saving."

Similarly, at 3 minutes into the task, one expert said:

> "I guess I have to establish a set of assumptions of my own"

At 10 minutes into the task, one novice said:

> "The number of the message line has to be specified... In order to get the message,.. if I have 4 messages, I need to know which lines I'm going to take if the user only wants to save one message.."

Similarly, at 10 minutes into the task, one expert said:

"Let me start looking at the states of a user, first of all, and what the world is going to seem from the view of a user."

4. Observation II: Design Strategies

We observed that experts all employed the following four general design strategies --- and the novices did not.

1. The experts were *purposeful*: experts continually stated explicit goals and subgoals, and continually checked to see how their design satisfied those goals. For example, one expert said:

   "I want to go backwards for a minute. I want to think about how I got to here and from here to there and how I'm now going to go back to the user. OK I've got it."

   In contrast, novices operated in a more bottom up fashion: they pursued goals as problems came up, without a global sense of where they were going.

2. The experts were *model-directed*: experts drew on their experience and continually manipulated models of the mail system at various levels of abstraction, e.g., at the most abstract level, one expert viewed mail as a stream of incoming data that needed to be routed to the appropriate place. These models were used to set up goals to be pursued.

3. The experts always followed a course of *balanced development*; components of the system were designed in a breadth-wise fashion: at each level, the detail of each component was about the same. For example, one expert said:

   Subject: "So I'm trying to keep all the things level. ...."

   Intervierwer: "Knowing a little bit about each one."

   Subject: "Knowing a little bit about each one. The same level of complexity with each one and hopefully the questions may have,.. and as you've seen before sometimes when I ask a question about one thing it reminds me of another thing I had passed over before and if I'm at the same level of decomposition I can see some links between them."

   In contrast, novices plunged into the details of a specific component only to find when they came to the next component that assumptions and constraints of the earlier component were violated --- and thus bugs were introduced.

4. The experts employed a variety of *notes* that they used during the design:

   - *Assumptions:* these notes set out the parameters of the system; they were typically specified early in the design, e.g., we will be using dumb terminals.
   - *Constraints:* as components were being defined certain properties that would have a global effect needed to be noted, e.g., in working on the REPLY command a constraint was set up that the buffer pointer to the current message should not be updated by the READ command.
   - *Expectations:* these notes set up demons that would interrupt the designer at key points in the process, e.g., in reviewing the LIST-HEADERS command, the designer realized that the data structure for the mail messages better permit access to the subject field, as well as the contents field.

   The notes were used by the experts to continually monitor and evaluate the progress of the design.

5. Implications for the Software Aids

What are the implications of these observations for the design languages and support software? Where *did* the experts need assistance? It is clear to us that information management was a key skill that experts had, but which they could use some assistance on --- especially when the complexity of the task grows large. However, the type of information management that we think designers need is *not* simple "version management"; this type of assistance merely regurgitates back to the user exactly what he/she has typed in. Rather, the software aids that we see relevant to enhancing the design process are those that can *digest* the information provided by the designer. In particular, one aspect in which the designers seemed to need assistance was in the keeping track of the "notes" they made (the assumptions, expectations, and constraints) *and* recalling them at just the appropriate time. Software that could perform this type of assistance would require considerable understanding of the design process itself, and information that is problem specific.

For example, in designing an electronic mail system, assume the designer noted the following assumption to the software aid:

```
Assumption: use only dumb terminals
Reason: keep costs down
```

Then later when the designer was working on, say, the SEND command, and contemplating how a message could be edited, the software aid should respond with:

```
Careful: you assumed that dumb terminals would be used; this type of terminal
        does not have local editing capability
```

This type of *reminding* assistance would provide powerful assistance to an expert. Moreover, it might help a novice designer learn good habits, by encouraging him/her to carry out the design using notes about assumptions, expectations, and constraints.

6. Concluding Remarks

The verbal protocols we collected and analyzed from our subjects provide a tantilizing glimpse into the *process* of design. While even in this small pilot study we saw clear convergence of techiques among the experts — and clear differences between the novices and the experts, we see the observations made in this paper as only a beginning. We feel strongly that studies of the type reported here are necessary in order to get a better understanding of design -- which in turn can knowledgeably inform the development of design aids. Yes, Yogi, you *can* observe a lot by just watching!

D1

ORIGINAL PAGE IS
OF POOR QUALITY

N84 23146

EVALUATING MULTIPLE COORDINATED WINDOWS FOR PROGRAMMER WORKSTATIONS

Ben Shneiderman*, Charles Grantham, Kent Norman#, Judd Rogers, and Nicholas Roussopoulos*

*Department of Computer Science
#Department of Psychology

Human-Computer Interaction Laboratory
University of Maryland
College Park, MD 20742

October 14, 1983

ABSTRACT: Programmers might benefit from larger screens with multiple windows or multiple screens, especially if convenient coordination among screens can be arranged. This research project explores uses for multiple coordinated displays in a programmers workstation. Initial efforts focus on the potential applications, a command language for coordinating the displays, and the psychological basis for effective utilization so as to avoid information overload. Subsequent efforts will be devoted to implementing the concepts and performing controlled psychologically oriented experiments to validate the hypotheses.

INTRODUCTION

Full screen display editors are rapidly replacing line oriented editors, because they offer a larger window and more intuitively clear operations. Comparative studies indicate display editors can be learned in half the time and permit twice the productivity for many tasks (Roberts, 1979).

Similar productivity gains may be possible by further expanding the personal workstation to include multiple coordinated windows. Multiple windows have been used in graphics systems where one screen provides command facilities for the graphic display. Applications with complex information display requirements often employ multiple computer displays, e.g. nuclear reactor control, air traffic control, manufacturing control, spacecraft control, and commodity exchanges.

Multiple display research in programmer workstations has been conducted by the Japanese (Mano et al., 1982), in the Spatial Data Management project at Computer Corporation of America (Herot, 1980), and by Xerox with their overlapping windows strategy (Smith et al., 1982). This latter approach, often called the "cluttered desk model", allows the user to create

Presented at EIGTH ANNUAL SOFTWARE ENGINEERING WORKSHOP, NASA, Goddard Space Flight Center, 11/30/83

multiple windows in which independent processes can be initiated. Other researchers are developing the software architectures necessary to support multiple window activity (Gonzalez, 1982; O'Hara, 1983, Weiser et al., 1983).

Larger displays and multiple windows are attractive (IBM, 1983), but can overwhelm the user with too much information and the frustration of having to issue many commands to accomplish their tasks.

RESEARCH DIRECTION

In this project we propose to go beyond these early efforts and evaluate a multiple window environment in which the activities across windows can be coordinated to support programming tasks. Appealing applications include:

1) A central window shows program text, while the left window shows input test cases and the right window shows output results. Each press of a function key moves the left window to the next test case and the right window to the next output case. The programmer can then examine the code and verify the correctness of the output or track anomalies.

2) One window shows program text and as the cursor is moved onto a variable, the declaration, recent values, and cross-reference list automatically appear in another window.

3) One window shows the module design specification, another window shows the flowchart, and the third window shows the program code under development. As the user enters the name of another module, the specification, flowchart, and code appear simultaneously.

4) The top-down structure chart appears in one window, and as the user moves the cursor onto one of the boxes, the code and/or specifications appear in other windows.

5) Three windows show a contiguous section of a program 120 lines long, 40 lines per window. The command DOWN 25 causes all three screens to move down 25 lines.

6) With a single command the user can display all three (or more) modules invoked by a higher-level module, to check for commonality of argument passing strategies.

The list could be made much longer, but these examples convey the rich potential for multiple windows, if useful coordination and synchronization can be achieved conveniently. Multiple screens are advantageous for situations which require correlation between two segments of text, fuller context for comprehension of local code, and concurrent viewing of the root, sub-tree, and leaves of a tree structure.

We are in the process of designing a language to specify window coordination. Our initial approach is to use text editor macros to create a set of commands which would fit in the editor environment. For example, the macro nc (for Next Case) might be specified as 1>L /**/; 3>L /CASE/ which means locate on screen 1 the string ** (a marker for the beginning of an input test case) and simultaneously locate on screen 3 the string CASE (a header field for each output case). Conjointly, we will study programming behavior to isolate those tasks which can benefit from the user of multi-screen information presentation strategies.

We are in the process of designing a three screen programmer workstation to test alternative strategies. We hope to refine successful strategies by using the initial system for our own use and to test the system with programmers recruited to perform benchmark tasks. In addition to producing a useful system, we expect to develop a better understanding of how programmers do their work. An additional benefit would be the development of simplified strategies for coordinating split screens on single display systems - these concepts might be rapidly applied to currently available programmer workstations.

EVALUATION STRATEGY

Our early experiments will concentrate on comprehension tasks which can be administered in a well-controlled manner (Shneiderman, 1980). For example, we have observed that in-line comments tend to clutter the listing and cause more window movement commands to study a program. There are three experimental conditions:

1) Single screen with in-line comments - the control group.
2) One screen with program text only and one screen with comments only. A single window movement command will cause both screens to move in synchrony.
3) Two screens which are linked together to show twice as many lines of program text with in-line comments. The screens are linked so that they act as simply a doubly long window.

Subjects will be given a comprehension test forward trace (for a given input what is the output), backward trace (for a given output what must the input have been), value of variables, counts of execution, and other questions. Subject evaluations complement the objective test scores.

As our implementation becomes more powerful we will explore program debugging, modification, and composition tasks.

Acknowledgements: We are grateful to IBM Federal System Division for support of this project.

ORIGINAL PAGE IS
OF POOR QUALITY

REFERENCES

Gonzalez, J. C., Implementing a window system for an all points addressable display, IBM Cambridge Scientific Center Report G320-2141, (December 1982).

Herot, Christopher, Spatial management of data, ACM Transactions on Databases 5, 4 (December 1980), 493-513.

IBM 3290 Information Panel: Description and Reference, Form GA23-0021-0, Kingston, N.Y., (March 1983).

Mano, Yoshihisa, Omaki, Kazuhito, and Torii, Koji, Early experiences with a multi-display programming environment, Proc. 6th International Software Engineering Conference, Available from IEEE, (1982), 422-423.

O'Hara, Robert P., An interactive display environment or knitting sheep's clothing for a wolf, Proc. National Computer Conference, Vol. 52, AFIPS Press, Arlington, VA, (1983), 329-339.

Roberts, Teresa L., Evaluation of computer text editors, Ph. D. dissertation, Stanford University, (1980). Available from University Microfilms, Ann Arbor, MI, AAD 80-11699.

Shneiderman, Ben, Software Psychology: Human Factors in Computer and Information Systems, Little, Brown and Co., Boston, MA (1980).

Smith, Cranfield, et al., Designing the STAR user interface, BYTE 7, 4, (April 1982), 242-282.

Weiser, M., Torek, C., Trigg, R., and Wood, R., The Maryland window systems, University of Maryland Computer Science Technical Report TR-1271, (January 1983).

EVALUATING MULTIPLE COORDINATED WINDOWS for PROGRAMMER WORKSTATIONS

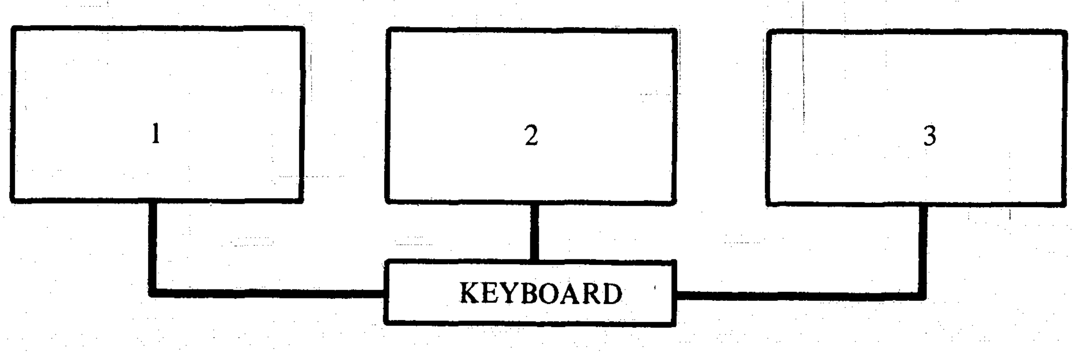
Ben Shneiderman*, Charles Grantham, Kent Norman#, Judd Rogers and Nicholas Roussopoulos*

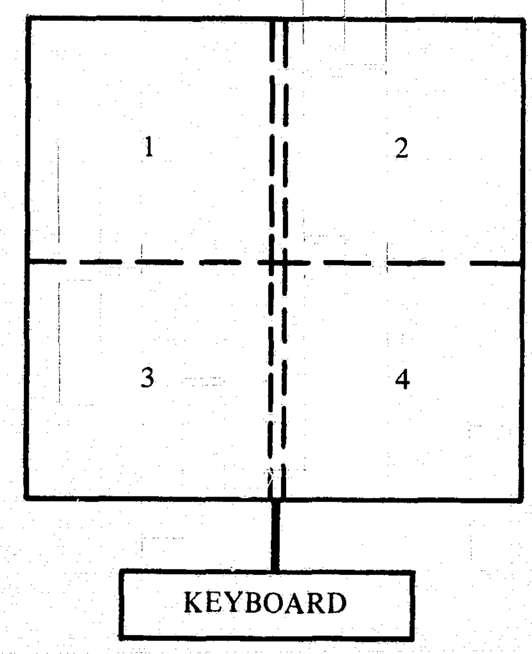
* Department of Computer Science
# Department of Psychology

Human-Computer Interaction Laboratory
University of Maryland
College Park, MD 20742

ORIGINAL PAGE IS
OF POOR QUALITY

TWO RESEARCH STRATEGIES





E. Grantham
U of M

PROGRAMMER TASKS TO BE EVALUATED

(all require comprehension)

A – Composition

B – Testing

C – De-bugging

D – Modification

FOCUS IS ON COORDINATED USE OF INFORMATION FOR EACH GIVEN COGNITIVE TASK

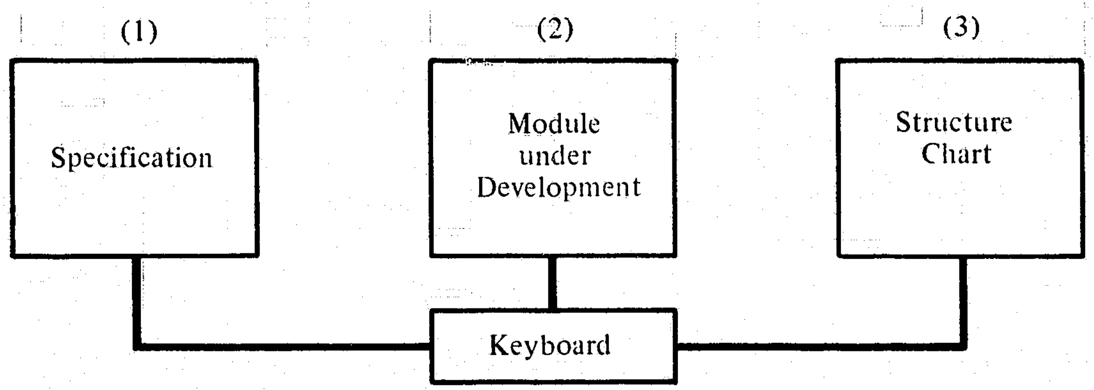
Summary of Observational Study Results

TASK	COGNITIVE ACTIVITIES	
COMPOSITION	INTEGRATION	• Data Structure • Control Structure • Modular Design
TESTING	CORRELATION	• Input • Expected utput
DE-BUGGING	VARIANCE FROM PLAN	• Semantics • Syntax
MODIFICATION	REFERENCE + LOCATION	• Semantics • Specifications

EXAMPLE:

CONFIGURATION A

PROGRAM COMPOSITION
TASK



Reference to a module causes all screens to move in a linked fashion.

Screen 2 is the 'work area'; Screen 1 displays specifications; Screen 3 – that portion of the Structure Chart where the module appears.

EXAMPLE:

CONFIGURATION B

PROGRAM COMPOSITION
TASK

(1) Specifications	(2) (User Definable Area) Second Module
(3) Module under Development	(4) Structure Chart

Coordinated changes of information occur in same fashion as Configuration A except user definable space is in window 2.

SOFTWARE MACROS

SEMANTIC ORGANIZATION:

One screen will be control target. Action (input) on this screen causes correlated changes in other screens.

In Configuration A: Screen 2

In Configuration B: Window 3

POTENTIAL SYNTAX:

```
MACRO NC              (macro definition)
1>L/**/
3>L/CASE/
END MACRO
```

E. Grantham
U of M

PANEL #4

QUALITY ASSESSMENT

P. Currit, IBM
K. Rone, IBM
J. Romeu, IITRI

D/10

N84 23147

Cleanroom Certification Model

P. A. Currit

October 13, 1983

Introduction

The "Cleanroom" software development methodology is designed to take the gamble out of product releases for both suppliers and receivers of the software. The ingredients of this procedure are a life cycle of executable product increments, representative statistical testing, and a standard estimate of the MTTF (Mean Time To Failure) of the product at the time of its release.

In the paper we consider a statistical approach to software product testing using randomly selected samples of test cases. A statistical model is defined for the certification process which uses the timing data recorded during test. A reasonableness argument for this model is provided that uses previously published data on software product execution. Also included is a derivation of the certification model estimators and a comparison of the proposed least squares technique with the more commonly used maximum likelihood estimators.

A Statistical Model of Software Reliability

If there are errors in a software product, users may experience intermittent failures as the product is executed. Unlike the possibility of intermittent failures in hardware, these intermittent failures in software are repeatable -- that is, if the software is executed again under identical initial conditions, then the failures will occur in exactly the same places. The appearance of intermittent failure in software in a given instruction seeming to fail one time and not another is due to the complexity of circumstances in which the instructions are executed rather than in underlying physical problems that occur during the execution of the instruction.

In the case of hardware failures, the basis for a statistical model appears in the very physical behavior of the hardware. But in the software, we must find another basis for statistical behavior. Fortunately, that basis is close at hand -- it is in the nature of the usage of the software by various users. Any particular user will make use of the software from time to time with different initial conditions and different inputs. During any specific use of the software, inputs may be entered from time to time and outputs observed from time to time during the course of the execution. The only failures detectable in the software are either from its aborting or from producing faulty output. But any one execution from a fixed initial condition from fixed inputs will behave similarly for every user every time they use it.

We call any such fixed use an "execution" which is distinguishable from all other executions by its initial condition and its inputs. Any given execution may have one or more failures associated with it, which is determined by the software itself as compared to the specification it is intended to satisfy. An execution will require a fixed number of machine cycles.

Now, we can imagine an "execution lifetime" for any given user to be the sequence of executions the user calls for with the software. Such sequences of executions for each user can be assembled into a collection of sequences of executions -- one for each user -- and the statistical properties of this collection identified as a stochastic process. That is, we consider, for the software product, a statistical pattern of usage for the product in terms of its initial conditions and inputs. Any execution selected in such a stochastic process will in general depend upon the past history of the sequence. For example, it is very unlikely that a user will query files before the files are loaded or that a user will call for two successive file maintenance executions. These kinds of conditions can be represented in a stochastic process which defines probabilities at any point in time to depend upon the state of the past history of the process.

With a statistical basis of user usage of the product, we can determine various statistical measures such as the MTTF, or the variance around the MTTF, etc. where time is measured in machine cycles. We are interested in failure free execution intervals, rather than trying to estimate the errors remaining in a software design. Our objective is to measure operational reliability which is the reason for the user usage perspective.

The Effect of Engineering Changes on the MTTF of Software

Consider a software increment under test and certification in which failures are observed and the results returned to the development group. On the analysis of these failures, the development group may propose engineering changes to correct the software. These engineering changes can increase the MTTF of the software, and we wish to account for that increase in the MTTF.

When engineering changes are made to software, it is only prudent to undertake regression testing to insure that these changes have not created new failures in execution. This regression testing should use previously generated statistical tests. It goes without saying that this regression testing cannot be considered part of the statistical sample used for estimating the reliability of the software. Instead, the increased MTTF, if any, must be detected and accounted for by new samples independent of the old ones (very likely new samples in later increments in which the retested software is only part of the total software being tested).

Suppose at a certain point in time that a set of engineering changes EC_1, EC_2, EC_m, has been applied to the software as a result of certification testing and analysis. Suppose that the failure rate of the software is λ and the failure rate associated with engineering change EC_i is λ_i. Then the failure rate associated with all the engineering changes made to date is the sum

$$\lambda_1 + \lambda_2 + \ldots + \lambda_m$$

If the engineering changes have corrected all errors in the software, then the foregoing sum will equal λ; otherwise, it will be less than λ. But, in fact, no one will ever know which case holds, and we assume neither case.

For convenience, we define λ_0 to be the deficiency, if any, between λ and the foregoing sum. That is

$$\lambda_0 = \lambda - \lambda_1 - \lambda_2 \ldots - \lambda_m$$

In this case, the quantities

$$p_0 = \lambda_0/\lambda,\ p_1 = \lambda_1/\lambda,\ p_2 = \lambda_2/\lambda,\ \ldots\ p_m = \lambda_m/\lambda.$$

are probabilities -- namely, p_1 is the probability that a failure was caused by the error corrected by engineering change EC_1. (p_0 is the probability a failure was caused by an error not corrected by any engineering change.)

If we assume an exponential distribution for time to next failure, in line with Adams' (8) and Nagel's (9) findings, the MTTF is the reciprocal of failure rate. We can then calculate a new MTTF after each successive engineering change has been made - namely, beginning with $MTTF_0$, the $MTTF_m$ of the original product after m changes will be

$$MTTF_1 = MTTF_0/(1-p_1)$$

$$MTTF_2 = MTTF_0/(1-p_1 - p_2)$$

$$\ldots$$

$$MTTF_m = MTTF_0/(1-p_1- \ldots - p_m)$$

The p_i values, or correspondingly λ_i/λ, can be expected to be decreasing in size, even though we cannot observe them directly with any certainty. This is because the errors with the highest associated rates of failure will be most likely detected and corrected earliest. That can't be guaranteed, of course, because a rare failure may well occur early as well and a correction made for it.

This expected decrease in size can be modeled in a simplified form if the p_i are defined by the probability distribution of geometrically decreasing terms

$$p_i = (1-\alpha)\ \alpha^{i-1},\ 0 > \alpha > 1$$

That is, each p_i is a fraction α of the preceeding p_{i-1}. We can explicitly sum the denominator on the right side of the $MTTF_m$ equation to get a new formula

$$MTTF_m = MTTF_0 R^m, \text{ where } R = 1/\alpha$$

In this formula, R is the average fractional improvement of the MTTF for each engineering change. In fact, in actual practice, R is just an average. Some engineering changes will affect the MTTF more than others depending upon the rate of failure associated with the error that has been fixed.

This particular software reliability model has been independently derived by several other people starting with different initial assumptions. It is equivalent to the Moranda geometric de-Eutrophication model and the Ramamoorthy-Bastani input domain based model. Moreover all of these models can be viewed as special cases of the Cox Proportional Hazard model.

It is well known that engineering changes themselves can introduce more errors in a software product. It appears that errors induced by such changes are much smaller when carried out by the original development group than with a separate field support group; but, nevertheless, we augment the above model with a contribution of error from engineering changes themselves. For this purpose, assume that engineering change EC_i introduces a failure rate at the level of ρ_i, then the above calculation needs to be modified to alter the definition of the p_i to the following

$$p_i = (\lambda_i - \rho_i)/\lambda$$

In this case, the remaining calculations go as before with p_0 again defined to account for the discrepancy but with the same end result, namely

$$MTTF_m = MTTF_0\ R^m$$

A Reasonableness Check of the Model

In 1980 Adams analyzed the software failure history of a number of large software products. Table 1 is taken from that work and illustrates the percent of errors in various failure rate classes. Two striking features of this data are the wide range of failure rates and the high percentage of very low rate errors. One third of the errors have MTTF of 5000 years.

Table 1

FITTED PERCENTAGE DEFECTS

MEAN TIME TO PROBLEM OCCURRENCE IN KMONTHS BY RATE CLASS

PRODUCT	60	19	6	1.9	.6	.19	.06	.019
1	34.2	28.8	17.8	10.3	5.0	2.1	1.2	0.7
2	34.3	28.0	18.2	9.7	4.5	3.2	1.5	0.7
3	33.7	28.5	18.0	8.7	6.5	2.8	1.4	0.4
4	34.2	28.5	18.7	11.9	4.4	2.0	0.3	0.1
5	34.2	28.5	18.4	9.4	4.4	2.9	1.4	0.7
6	32.0	28.2	20.1	11.5	5.0	2.1	0.8	0.3
7	34.0	28.5	18.5	9.9	4.5	2.7	1.4	0.6
8	31.9	27.1	18.4	11.1	6.5	2.7	1.4	1.1
9	31.2	27.6	20.4	12.8	5.6	1.9	0.5	0.0

Table 1 gives a new insight into the power of statistical testing, relative to selective testing or inspection, for improving MTTF. Finding errors at random is a very different matter than finding execution failures at random. One third of the errors found at random will hardly affect the MTTF at all; the next quarter of the errors found at random do little more. The two highest rate classes, some two percent of the errors, cause a thousand times more failures per error than the two lowest rate classes, some sixty percent of the errors. That is, statistical testing will uncover the high rate errors by a factor of 2000/60, some 30 to 1, while randomly finding errors uncovers high rate errors by a fraction of only 1 to 30.

The availability of the Adams data provides a unique opportunity for checking model(s) reasonableness, since it can provide failure rate as a function of engineering change. Most available data is given in terms of errors found or inter-fail times but not true failure rate. With the Adams data separate examinations of model assumptions and parameter estimation techniques can be performed. Quantile-Quantile and trend plots have been previously proposed for comparing the goodness of fit of different models but without failure rate data were unable to differentiate between assumptions and estimation techniques when models performed poorly.

The reasonableness analysis for the certification model was performed in two parts, first assuming perfect debugging of the software and subsequently considering the effect of introducing errors during product repair.

To perform the analysis, failure rate data exhibiting the effects of engineering changes was derived from the Adams data which was reported in terms of failure rate classes and failure counts. First $\lambda_1, \lambda_2 \ldots, \lambda_8$ were defined as the failure rates associated with the eight rate classes and n(i,k) as the number of errors in rate class i after k engineering changes. The initial failure rate for a product (before making any engineering changes) can then be expressed as

$$F_0 = \Sigma_1^8 \lambda_i \, n(i,0).$$

To introduce the effect of engineering changes, successive failure rates must be derived by simulating the occurrence of failures. This is done by expressing the probability of the first error being from rate class i as

$$\lambda_i \, n(i,0)/\lambda$$

and using a random number generator to select a rate class (designated i_0) according to these probabilities. The number of errors in each class rate after removing the first error would then be expessed as

$$n(i,1) = n(i,0) \text{ for } i \neq i_0$$
$$n(i,1) = n(i,0) - 1 \text{ for } i = i_0.$$

Since the number of errors for each rate class can be derived, the failure rate for the product after removing one error (first engineering change) can be expressed as

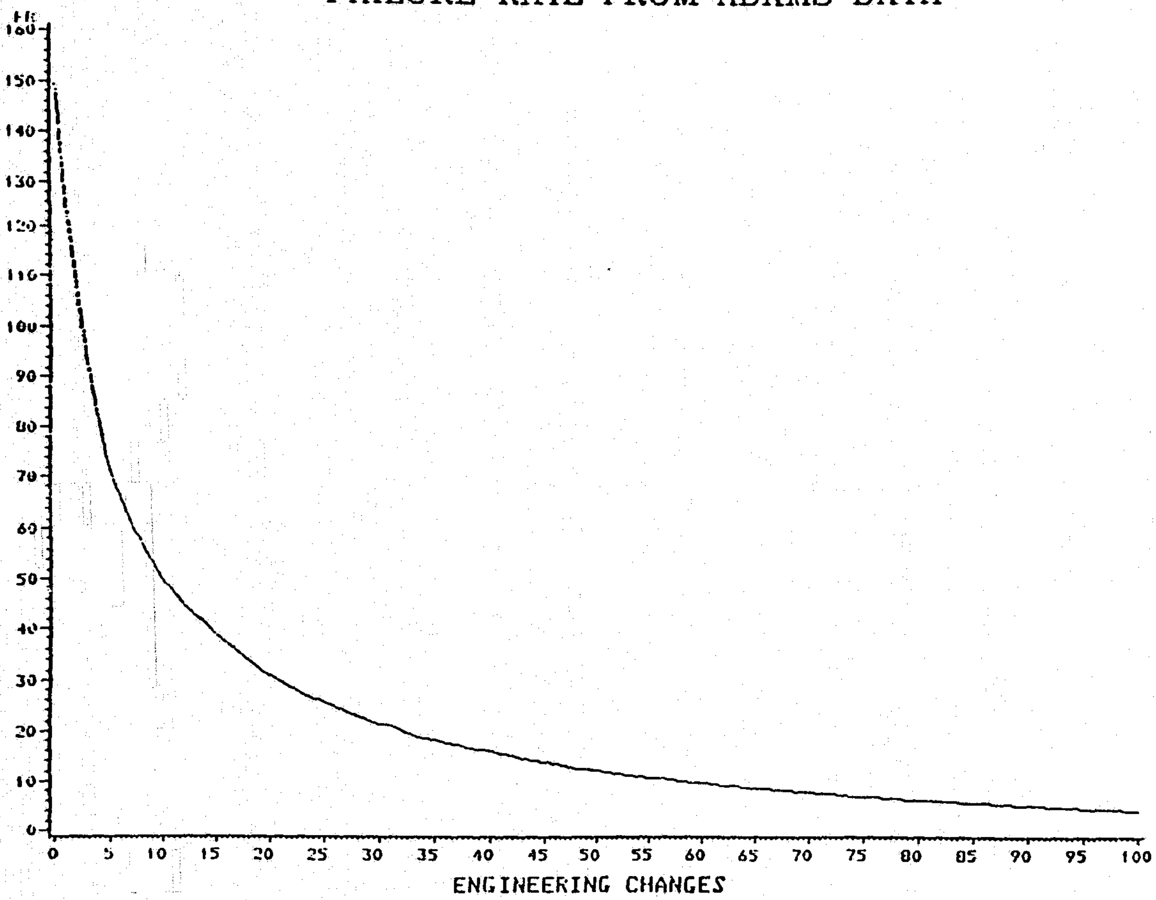
$$F_1 = \Sigma_1^8 \lambda_i \, n(i,1).$$

This process is repeatable to develop successive failure rates for the product and produces a single realization of a failure rate curve based on the Adams data.

For reasonableness analysis an expected failure rate curve obtained by averaging a number of realizations is a better tool. Figure 1 illustrates such a curve that was created by averaging 100 realizations of the Adams data assuming an initial count of 500 errors. The availability of the expected failure rate curves permits an examination of the reasonableness of the proposed certification model and a comparison of its assumptions and estimation techniques against other software reliability models.

The curve illustrated in Figure 1 is not of the form $1/MR^k$ (the reciprocal of MTTF in the certification model) since its logarithm is not linear, as shown in Figure 2. However large segments of the curve are of the desired form which suggests that the model is useful for certification but not extended prediction. Since our objective is software certification, the model satisfies this role and introducing complexity for long range prediction is not warranted.

FAILURE RATE FROM ADAMS DATA

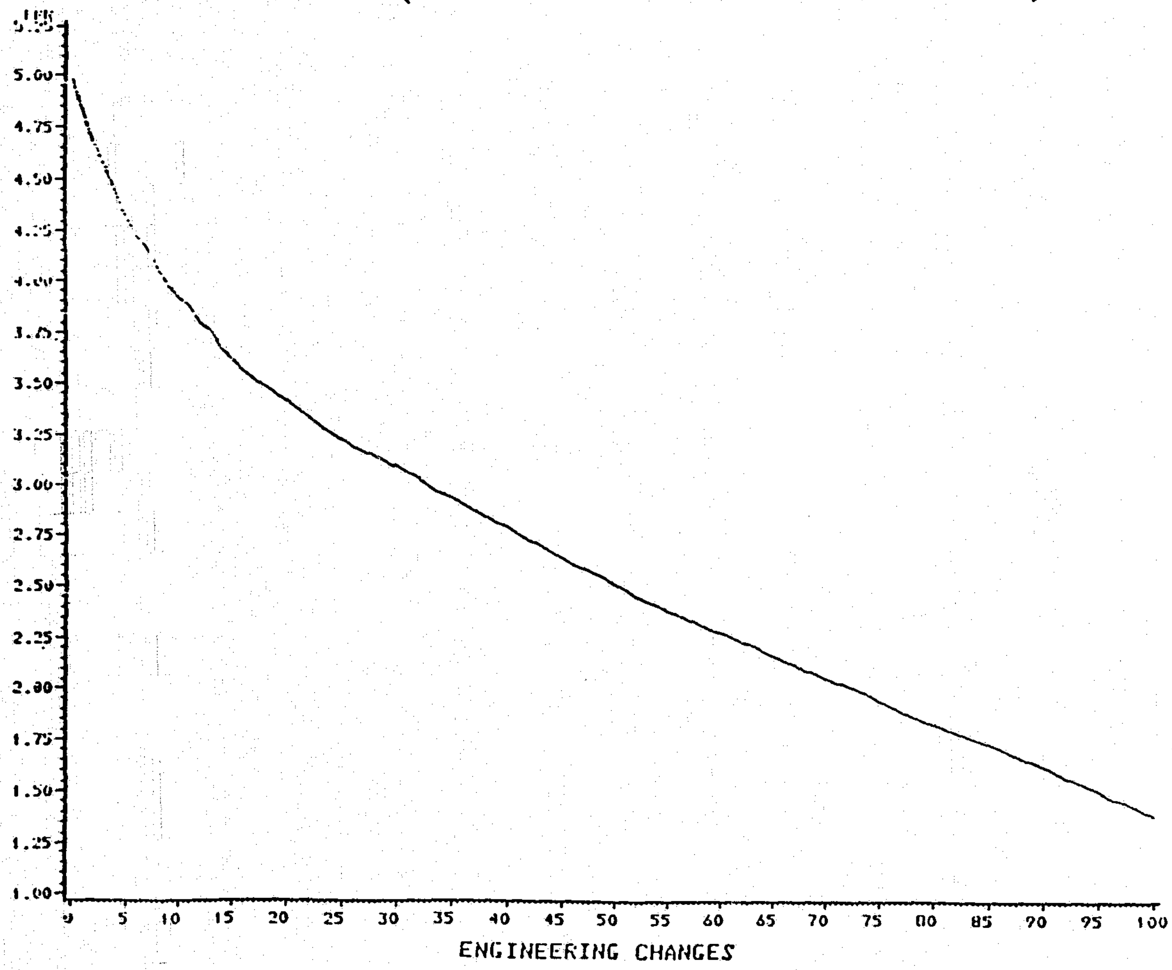


200 INITIAL BUGS GAMMA=0 50 SIMULATIONS

Figure 1

IBM 8/53

LOG(FAILURE RATE FROM ADAMS DATA)



200 INITIAL BUGS GAMMA=0 50 SIMULATIONS

Figure 2

ORIGINAL PAGE IS OF POOR QUALITY

The failure rate curve shows that failure rate does not decrease linearly with engineering changes as assumed by the Jelinski-Motanda and derivative models. The Littlewood-Verrall model has good fit with the failure rate curve and has been used by one of the authors for long range predictions.

Failure rate curves that cover the imperfect debugging case require additional knowledge of the probability (γ) that a fix creates an error and the probability (g_i) that the created error is from rate class i. A reasonable assumption is that γ should be in the range 0 to .25 and based on Adams suggestion (14) g_i can be derived by assuming the repair process is similar to the development process. The following set of g_i values have been experimentally derived:

	Rate Class of Created Error							
	1	2	3	4	5	6	7	8
Probability	.04	.08	.15	.22	.20	.16	.09	.06

Expected failure rate curves have been generated using the derived g_i data over a range of γ values. Analysis indicates that the certification model is equally useful in the imperfect debugging case where the major distinction is the appearance of fatter tails than in the perfect debugging case.

Parameter Estimation

To use the proposed model for actual software certification, methods are required for estimating the model parameters ($MTTF_0$ and R) from recorded testing data. The suggested estimation procedure differs from methods used by other reliability models and is based on a least square technique.

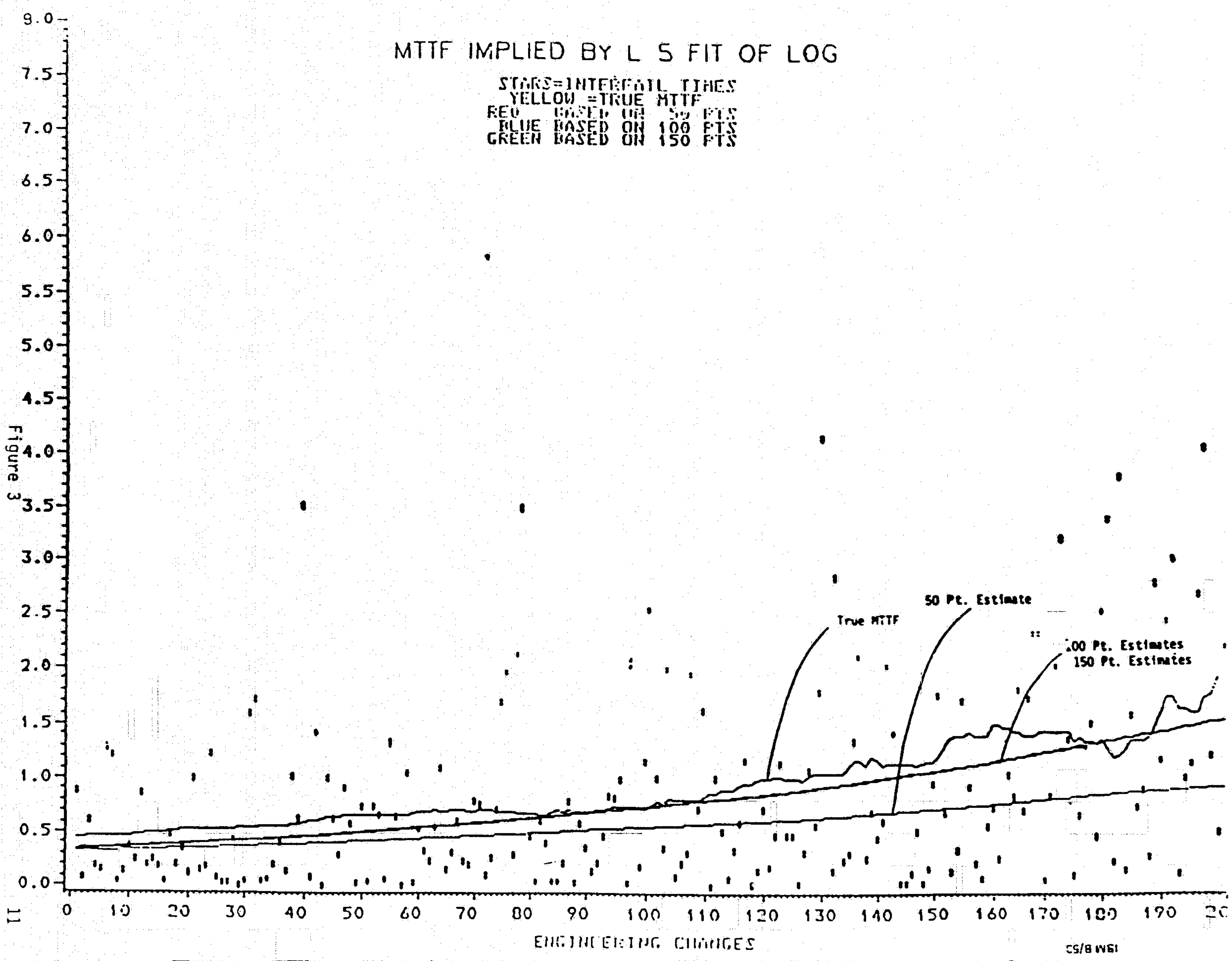
Let $t_1, t_2, \ldots, t_n$ be the successive interfail times for a product under test and certification. From time to time, engineering changes will be made to the product in response to observed failures. These changes are introduced after the failures are observed, and typically packaged in an incremental release from development to test. For each i, let c_i be the cumulative number of engineering changes made to the product after the interfail interval measured by t_i. The t_i introduce a source of randomness since the times to failure will vary about the mean. The certification model and most other models assume an exponential distribution, which seems to be corroborated by the Nagel and Skrivan work.

Properties of a model's estimators, such as unbiasedness, are as important as the underlying model assumptions. Estimators for the $MTTF_0$ and R parameters in the certification model have been developed, which are computed with a least squares analysis of the logarithms of the interfail times and a bias correction.

Most existing models rely on maximum likelihood estimators (MLE) which have a known set of problems as discussed by Forman-Singpurwalla, Sukert, and Littlewood-Verrall relative to the Jelinski-Moranda model. It has been demonstrated that for practical number of data points MLE exhibits bias and has greater variance than the estimators log least squares estimators. The bias can not be corrected because there is not a closed form MLE solution.

Figure 3 shows the MTTF curves when the logarithmic technique is used for estimating the $MTTF_0$ and R parameters. As calibration points, 50, 100, 150 and 200 data points were selected to evaluate the method using the simulation of the Adams data. Since the intent at this point was to demonstrate the effectiveness of the estimators, interfail times simulated from a curve of the form $MTTF_0 R^{c_i}$ could have been used. However using any realization, such as illustrated in Figure 3, provides a more interesting test and a closer simulation of what will actually occur. As can be seen, all curves give a good prediction of MTTF with the most discrepancy in the 50 point case when prediction is carried too far into the future.

Figure 3



ORIGINAL PAGE IS OF POOR QUALITY

P. Currit
IBM

OMIT

SOFTWARE CERTIFICATION MODEL

P. A. CURRIT
IBM Corporation
November 30, 1983

WHY MTTF?

Quality Should Be Measured From Customer's Perspective

- How often does it fail?
  - MTTF reports by large commercial customers
  - MTTF ship criteria
- What's the severity of a fail?

WHY MTTF?

Management Decisions

High MTTF is good, low MTTF is bad.

If x errors have been fixed, is that good or bad?

If x is small, either:

1) There were very few errors made

or

2) There are plenty of errors, but the testing process is ineffective

If x is large, either:

1) There were a large number of errors made

or

2) The testing process is very effective

WHY MTTF?

You Get What You Measure.

Why Not Measure What The Customer Wants?

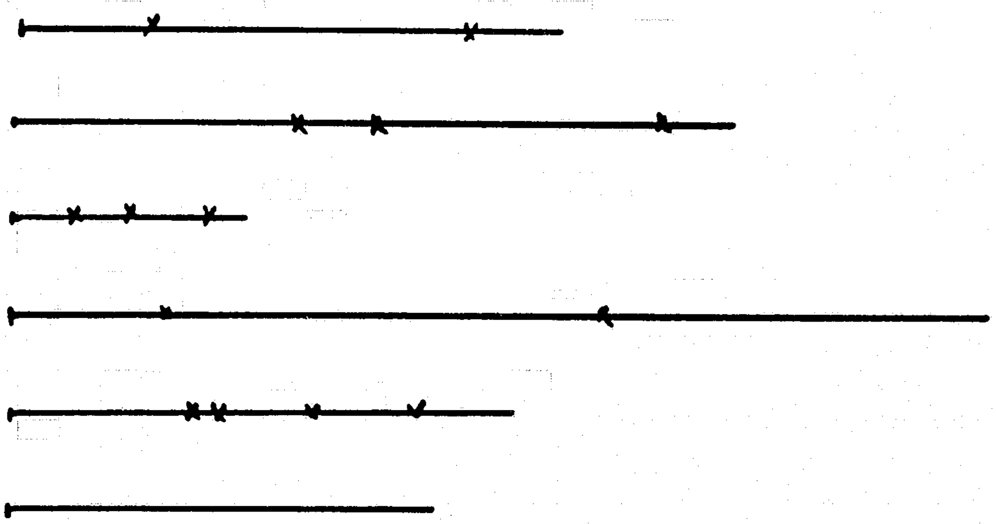
MODELING GOALS

Certify MTTF

Avoid Restrictive Assumptions

Provide Statistically Sound Estimators

Keep It Conceptually Simple



Each lifetime has a probability of occurrence.

Failure rate F of the programs is averaged over all possible lifetimes.

$$MTTF = M = \frac{1}{F}$$

Each error e has an associated failure rate Fe.

$$F = \sum_{e} Fe.$$

Let $pe = \frac{Fe}{F}$. (Relative frequency of error e)

Upon removal of error e;

$$F \longrightarrow F - Fe$$

$$M \longrightarrow \frac{1}{F-Fe}$$

$$= \frac{1}{F(1-\frac{Fe}{F})}$$

$$= M \frac{1}{1-pe}$$

In general, upon removal of errors e1, e2, ---, en,

$$M \longrightarrow M \frac{1}{1-p1 -p2 - \ldots - pn} .$$

Let $p_i = (1-\alpha)\alpha^{i-1}$.

Then, after k engineering changes,

$$M \longrightarrow \frac{M}{1-\sum_{i=1}^{k}(1-\alpha)\alpha^{i-1}}$$

$$= \frac{M}{1-(\cancel{1-\alpha})\,\dfrac{1-d^k}{(\cancel{1-\alpha})}}$$

$$= \frac{M}{\alpha^k}$$

$$= MR^k$$

where $R = \frac{1}{\alpha}$.

SOFTWARE CERTIFICATION MODEL

Mean Time To Failure After k Engineering Changes

$$= MR^k$$

where

M = Mean time to failure before any engineering changes

R = Factor for relative improvement in MTTF due to a single engineering change

SOFTWARE CERTIFICATION MODEL ASSUMPTION

Equivalent to

Ramamoorthy-Bastani

Application: Nuclear Reactors

Moranda Geometric De-Eutrophication

Special case of

Cox Proportional Hazard Rate

Application: Boeing Computer Services

MODEL REASONABLENESS

Adams data

- Large software products
  - Product usage
  - Defects found
  - Failures due to a defect
- MTTF classification of defects
- Allows independent check of
  - Model assumptions
  - Estimation procedures

Fitted Percentage Defects

Mean Time to Problem Occurrence in Years

Product	1.6	5	16	50	160	500	1600	5000
1	0.7	1.2	2.1	5.0	10.3	17.8	28.8	34.2
2	0.7	1.5	3.2	4.5	9.7	18.2	28.0	34.3
3	0.4	1.4	2.8	6.5	8.7	18.0	28.5	33.7
4	0.1	0.3	2.0	4.4	11.9	18.7	28.5	34.2
5	0.7	1.4	2.9	4.4	9.4	18.4	28.5	34.2
6	0.3	0.8	2.1	5.0	11.5	20.1	28.2	32.0
7	0.6	1.4	2.7	4.5	9.9	18.5	28.5	34.0
8	1.1	1.4	2.7	6.5	11.1	18.4	27.1	31.9
9	0.0	0.5	1.9	5.6	12.8	20.4	27.6	31.2

E. N. Adams, RC 8228, 4/11/80, p. 19, IBM Research

SIMULATED FAILURE RATE CURVES

Based on Adams Data

For Perfect Debugging

Let $n(i,k)$ = number of errors of failure rate λ_i after k fixes

Let F_k = program failure rate after k fixes

$$F_0 = \sum_{i=1}^{8} \lambda_i \, n(i,0)$$

Probability that first failure is caused by an error of rate λ_i

$$= \lambda_i \, n(i,0)/F_0$$

Randomly select i_0 according to preceding probability

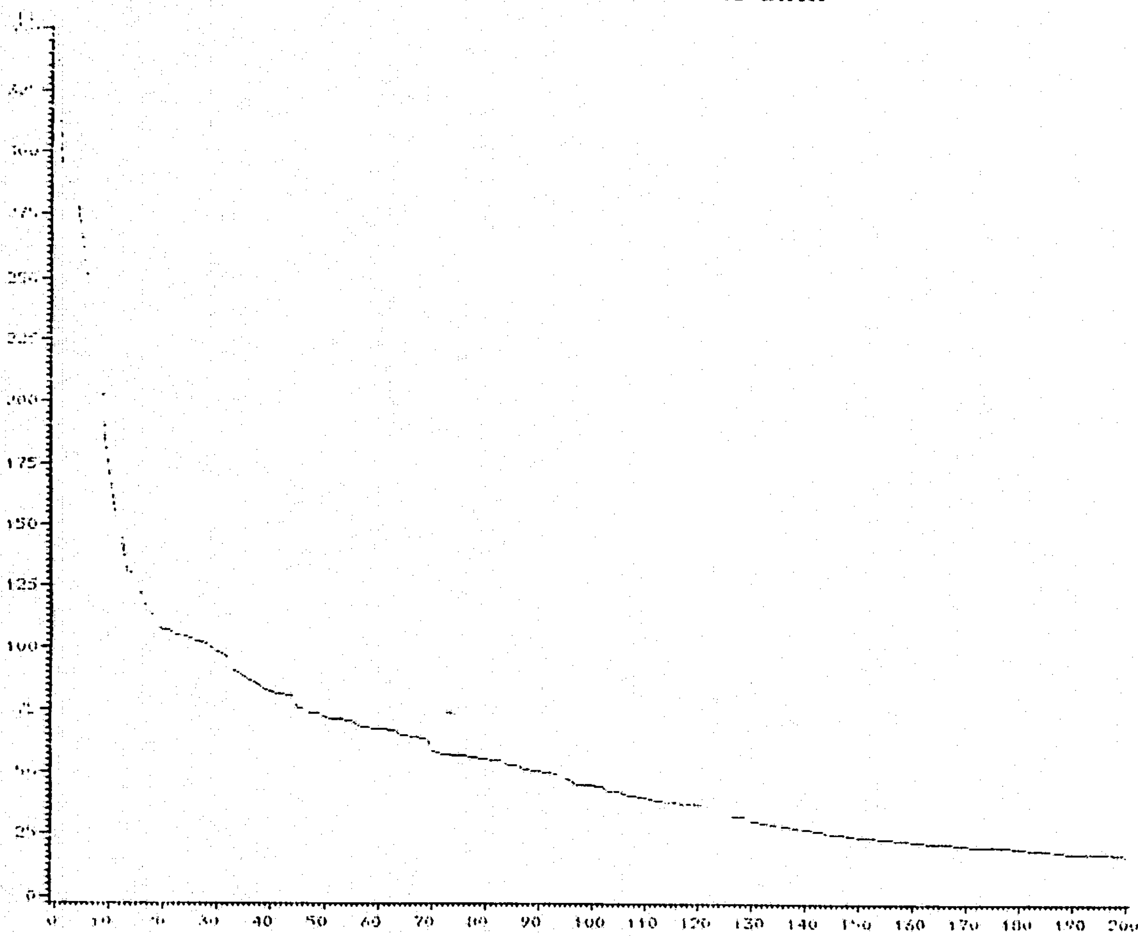
$$n(i,1) = n(i,0) \qquad i \neq i_0$$

$$n(i,1) = n(i,0)-1 \qquad i = i_0$$

$$F_1 = \sum_{i=1}^{8} \lambda_i \, n(i,1)$$

Repeat to determine F_2, F_3, ...

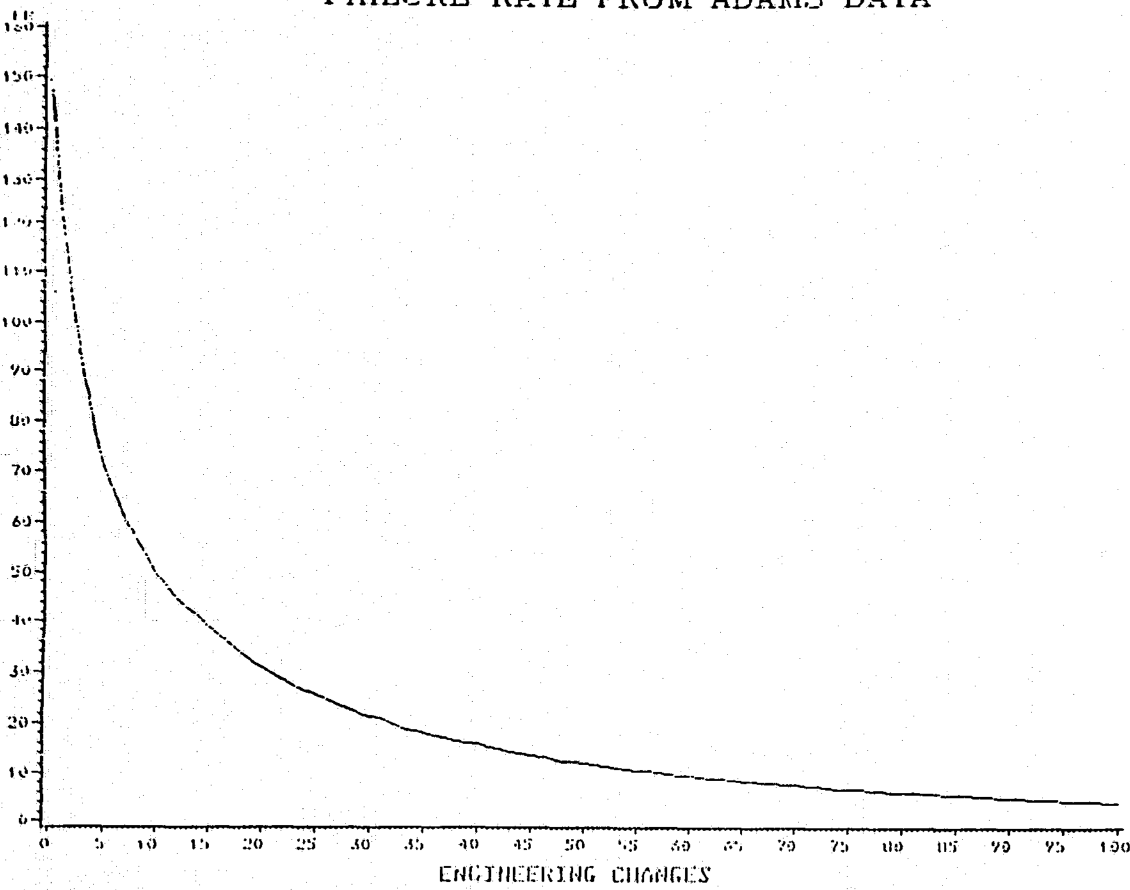
FAILURE RATE FROM ADAMS DATA



500 INITIAL BUGS GAMMA=0 1 SIMULATION

ORIGINAL PAGE IS OF POOR QUALITY

FAILURE RATE FROM ADAMS DATA



200 INITIAL BUGS GAMMA-0 50 SIMULATIONS

Figure 1

LOG(FAILURE RATE FROM ADAMS DATA)

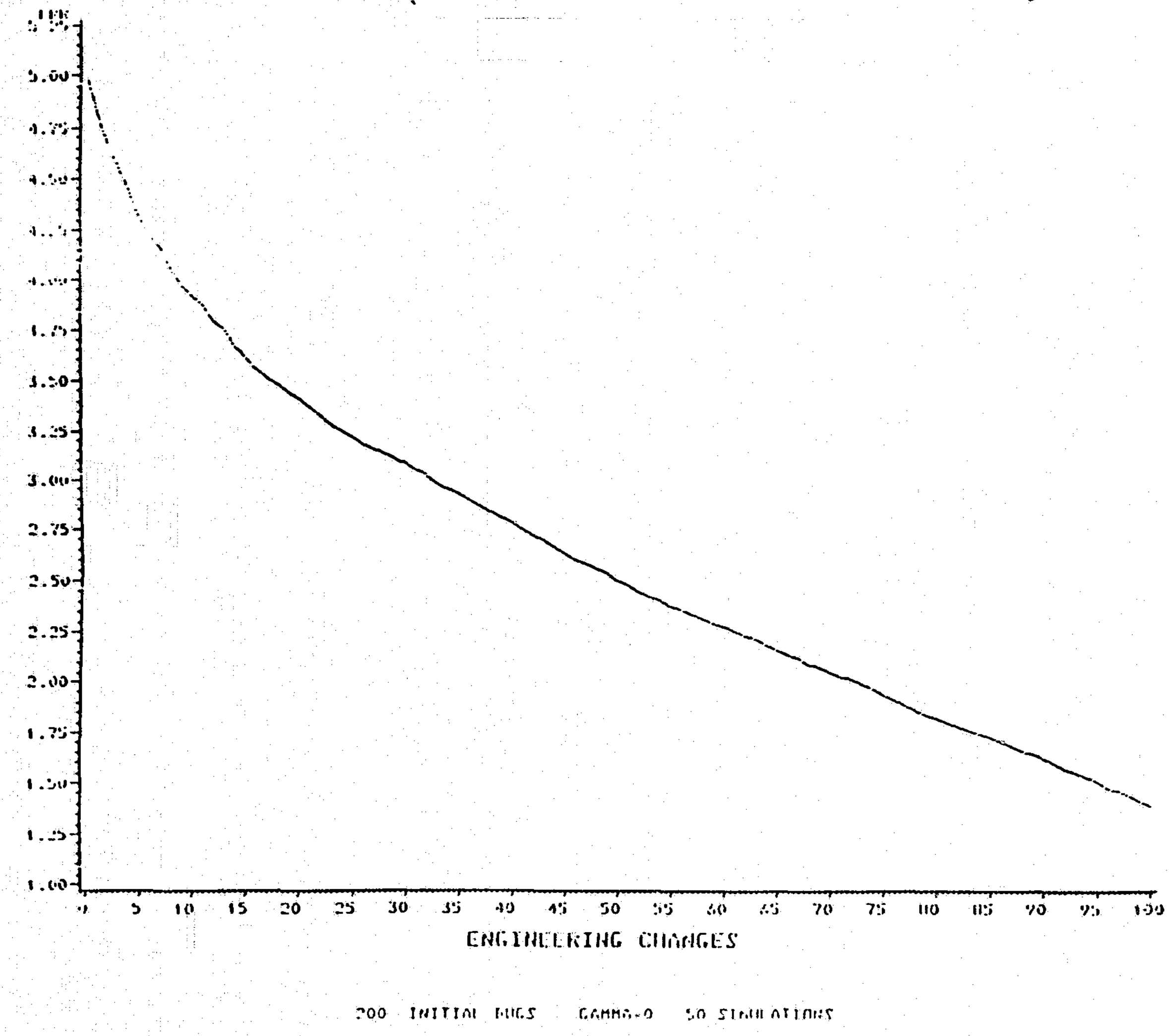
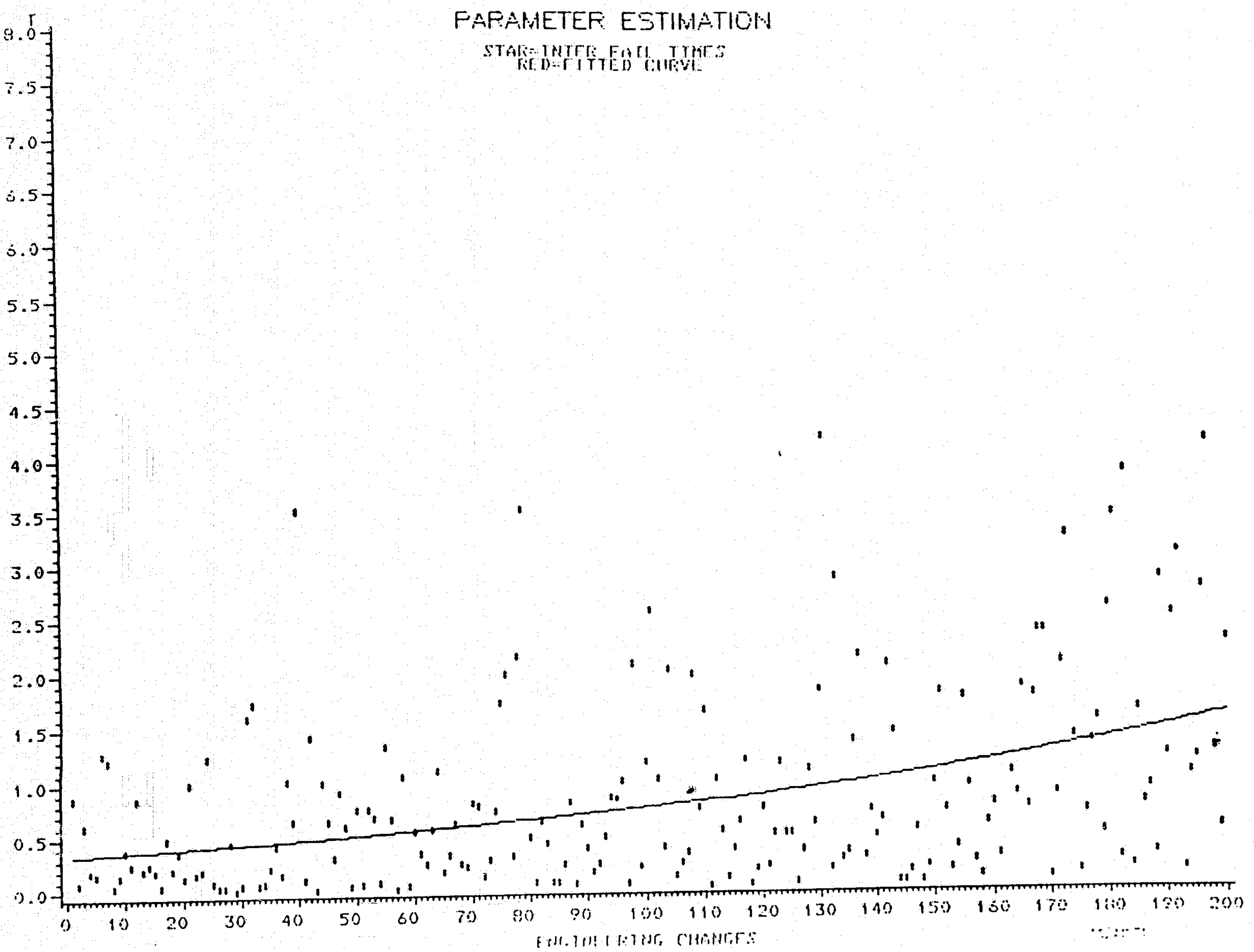


Figure 2


PARAMETER ESTIMATION
STAR=INTER FAIL TIMES
RED=FITTED CURVE
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
200
ENGINEERING CHANGES

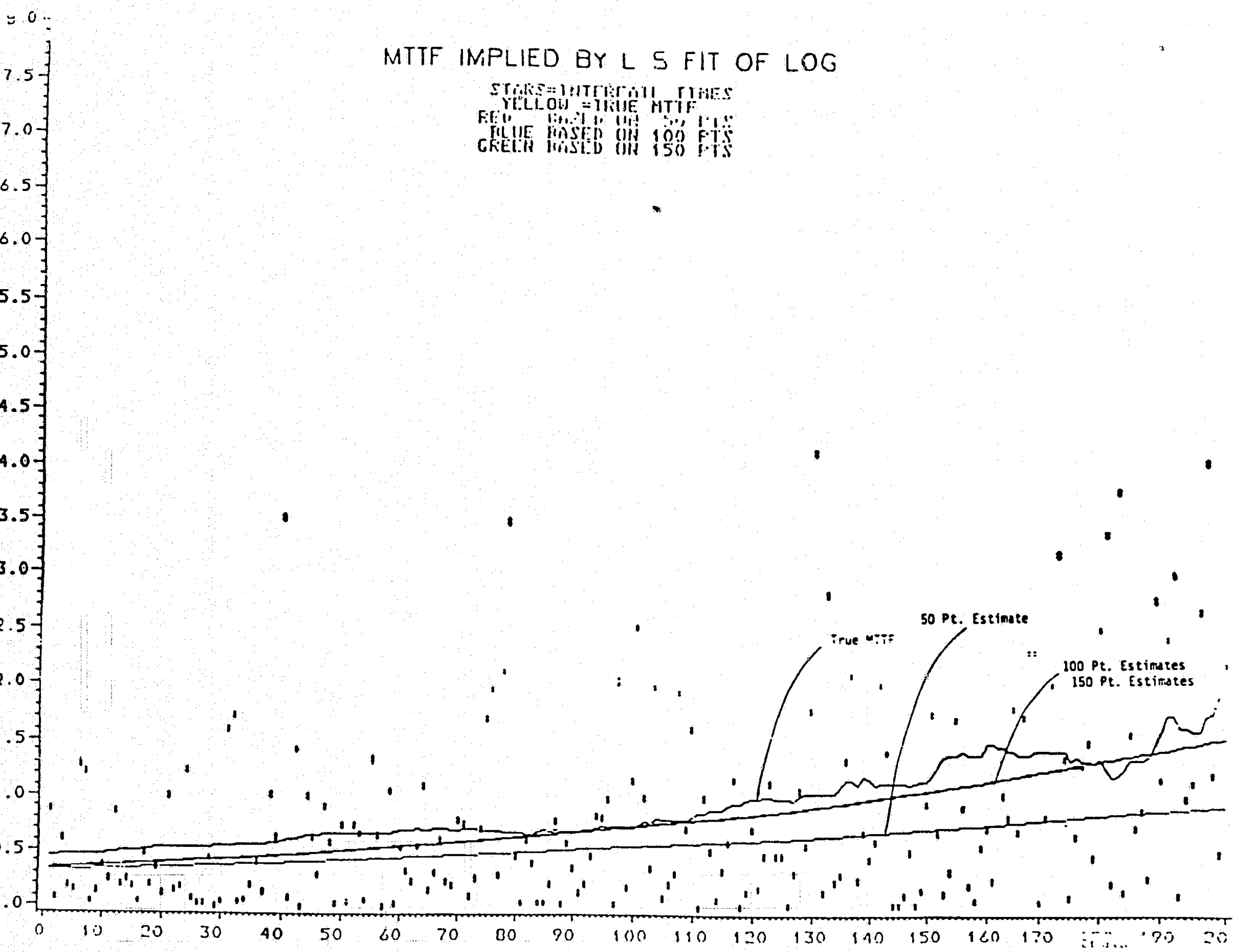
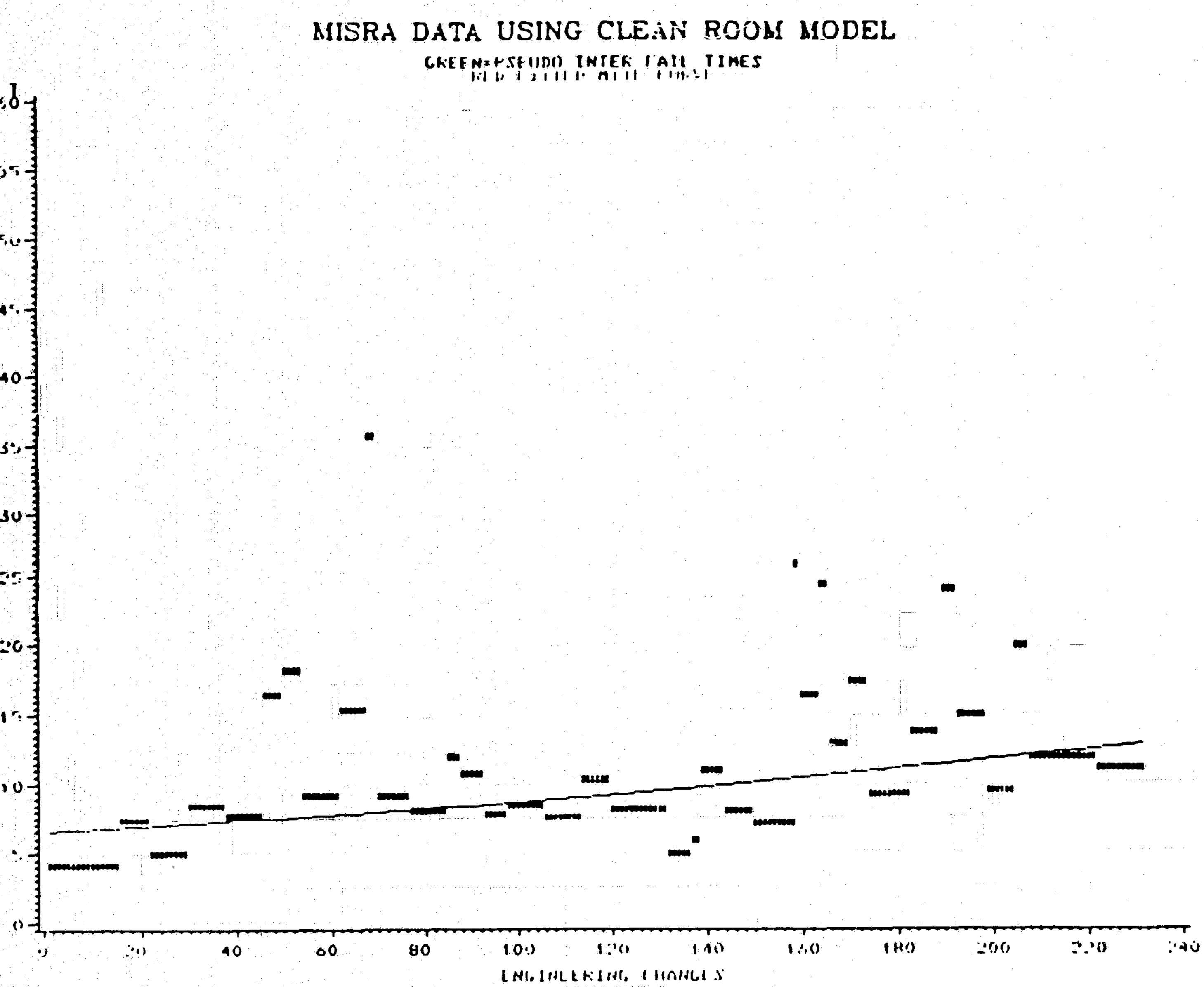

MTTF IMPLIED BY L S FIT OF LOG
YELLOW =TRUE MTTF
BLUE BASED ON 100 PTS
GREEN BASED ON 150 PTS
True MTTF
50 Pt. Estimate
100 Pt. Estimates
150 Pt. Estimates
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0 10 20 30 40 50 60 70 80 90 100 110 120 130 140 150 160 170

Figure 3

PARAMETER ESTIMATION

Comparison with Maximum Likelihood



Implies 15 expected errors in STS4

SOFTWARE CERTIFICAION MODEL

Estimators: Statistical Properties

- Unbiased
- Decreasing variance
- Relative efficiency

CONCLUSIONS

- Goals are acceptably satisfied

- Programmed version of the model is available

  (Although none is required. Any statistical package will suffice.)

D11

N84 23148

Projecting Manpower to Attain Quality

by
Kyle Y. Rone

International Business Machines Corporation
Federal Systems Division
Houston, Texas

Abstract

In these days of soaring software costs it becomes increasingly important to properly manage a software development project. One element of the management task is the projection and tracking of manpower required to perform the task. In addition, since the total cost of the task is directly related to the initial quality built into the software, it becomes a necessity to project the development manpower in a way to attain that quality. The purpose of this paper, then, is to describe an approach to projecting and tracking manpower with quality in mind.

The basic approach is to begin with a current manpower model which accurately describes the cost of developing a usable element of software. Then, based on the assumption that improving quality does not cost more over the entire life cycle, the current model is modified to reflect greater expenditure on elements of work which are known to improve initial quality. This requires a reduction in the cost of other elements since an increase in quality does not cost more. The obvious elements to reduce are those directly affected by quality. The final result of this type of analysis is the development of a manpower model designed to generate quality software.

The resulting model is useful as a projection tool but must be validated in order to be used as an on-going software cost engineering tool. A procedure is developed to facilitate the tracking of model projections and actual data to allow the model to be tuned. Finally, since the model must be used in an environment of overlapping development activities on a progression of software elements in development and maintenance, a manpower allocation model is developed for use in a steady state development/ maintenance enviornment.

Table of Contents

Introduction

In these days of soaring software costs it becomes increasingly important to properly manage a software development project. One element of the management task is the projection and tracking of manpower required to perform the task. In addition, since the total cost of the task is directly related to the initial quality built into the software, it becomes a necessity to project the development manpower in a way to attain that quality. The purpose of this paper, then, is to describe an approach to projecting and tracking manpower with quality in mind.

The basic approach is to begin with a current manpower model which accurately describes the cost of developing a usable element of software. Then, based on the assumption that improving quality does not cost more over the entire life cycle, the current model is modified to reflect greater expenditure on elements of work which are known to improve initial quality. This requires a reduction in the cost of other elements since an increase in quality does not cost more. The obvious elements to reduce are those directly affected by quality. The final result of this type of analysis is the development of a manpower model designed to generate quality software.

The resulting model is useful as a projection tool but must be validated in order to be used as an on-going software cost engineering tool. A procedure is developed to facilitate the tracking of model projections and actual data to allow the model to be tuned. Finally, since the model must be used in an environment of overlapping development activities on a progression of software elements in development and maintenance, a manpower allocation model is developed for use in a steady state development/ maintenance environment.

The Cost of Ignoring Initial Quality

In the past software projects have generated initial software relying on the usual network of functional, subsystem and system tests to find the "bugs" prior to system delivery. This is a questionable approach, however, when the overall cost of the finished (debugged) system is considered. As Figure 1 shows, software development is a pyramiding or stair-stepping group of functions each of which, when begun, continues until the project is complete. Errors found early in development when only the programmer is involved are essentially "free". That is, they can be absorbed in the normal work flow at a minimum cost. Once the code is placed on the master system, however, an error must be documented by a discrepancey report (DR) which must be eventually closed by all elements of the project. And so it goes, the later in the life cycle that a software error is discovered the more elements of the project are involved in the software and the more work must be done to correct the error. This naturally costs more. The result is that shown generically in Figure 1 and can be summarized as: The later in the software development cycle that an error is found, the more it costs.

The obvious conclusion is that steps should be taken to find errors early in the development process to minimize cost. The first step in this process is to define the positive actions required and to plan the life cycle and project appropriate manpower to accomplish those actions.

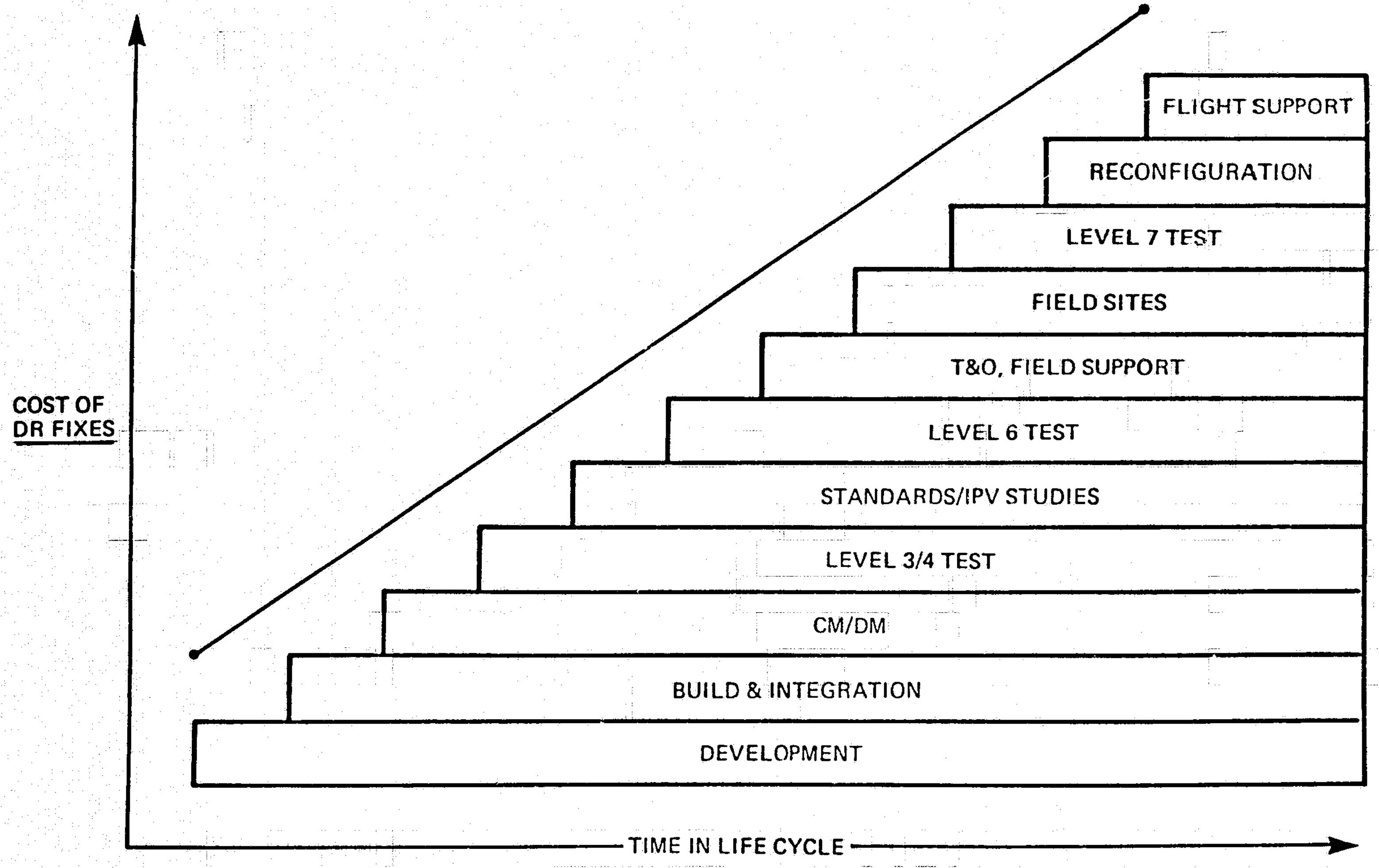


Figure 1. DR Cost vs. Time in Life Cycle

The Current Manpower Model

As mentioned in the introduction, the basic approach requires the use or generation of a manpower model which reflects the current software development environment. The complete development of the model used is described in Reference 1, however, to aid in this discussion a brief summary is included here. The development of the model was initiated by delineating all project costs with the following information:

- Type cost: direct change request (CR) cost/technical and project support costs.
- Organization: software development project organization.
- Function: purpose of the cost.
- Drivers: factors affecting the cost.
- Estimation methodology: how the item is estimated.

These project costs were placed into categories and then reordered by those categories. The categories used were as follows:

- Category I: direct CR cost
- Category II: development/verification technical support
- Category III: preprocessors
- Category IV: management and common support
- Category V: project release/schedule/reconfiguration
- Category VI: maintenance
- Category VII: project independent costs

Using the first five categories (ignoring maintenance and project independent costs for the moment) and examining Release 19 of the Shuttle onboard Primary Avionics Software System (PASS) we can express the cost of that release with the percent model shown in Figure 2.

CATEGORY	AREA	FUNCTION	R19 Manmonths	%
I	DEV.	Direct CR Est.	197	16
	VERIF.	Direct CR Est.	173	14
II	DEV.	Requirements Analysis (R.A.)	13	1
		Level 3 Test (L3)	26	2
		Systems Analysis (SA)	9	1
		Systems Architecture (SAr)	27	2
II	VERIF.	Studies and Audits (ST/AU)	19	2
		Common Function Tests (CF)	8	1
		Systems Measurement (S Meas)	14	1
		Level 7 Test (L7)	139	11
		Level 6 DR Support	38	3
III	DEV.	Preprocessors (PREP)	30	3
IV	DEV, VERIF, P.O.	Management and Support (M&S)	135	11
		Common Support (CS)	70	6
V	SFO	Build and Integration (B&I)	140	11
		Resource Management (RES MGT.)	100	8
		Configuration Management/ Data Management (CM/DM)	80	7
TOTALS			1218	100

ORIGINAL PAGE IS OF POOR QUALITY

Figure 2. Percent Model for PASS Release 19

By examining these costs by category, it can be seen that a factor can be developed which will relate the total cost through Category IV to the direct costs contained in Category I. This is accomplished by the following calculations:

DIRECT CR COSTS = CR_D = CATEGORY I

INDIRECT CR COSTS = CR_I = CATEGORIES II - IV

$$\textbf{FACTOR} = \frac{CR_D + CR_I}{CR_D} = \frac{I+II+III+IV}{I}$$

$$\text{FACTOR} = \frac{370 + 293 + 30 + 205}{370} = 2.5$$

This factor can be used along with estimates of the direct CR costs to calculate those costs driven by CR's. However, maintenance (Category VI) costs are also driven by CR costs whereas Categories V and VII are not.

The cost to maintain a CR is given by the area of the difference between a Rayleigh curve without the CR and one which includes it evaluated over the maintenance timeframe. (Figure 3)

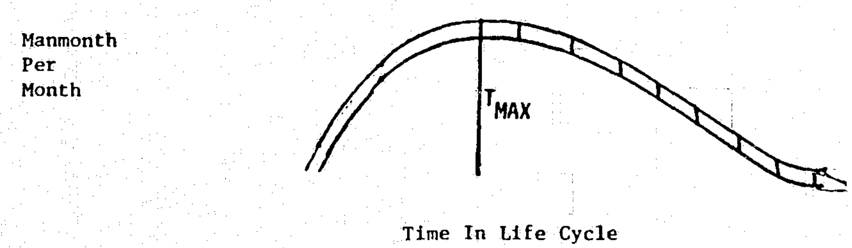


Figure 3. Maintenance Cost of a CR

ORIGINAL PAGE IS
OF POOR QUALITY

This is calculated by taking the integral of the difference between the two generalized equations of the curves and letting the time of the maximum be one year consistent with current release plans. Doing this a formula for maintenance is generated:

$$\text{Maintenance} = e^{-a_2} (K_1 - K_2)$$

But the time of the maximum is one year which implies that $a_2 = 1/2$. Thus:

$$\text{Maintenance} = e^{-1/2} (K_1 - K_2) = .6\ (K_1 - K_2)$$

This means that maintenance costs are 60% of the total cost of developing a CR. However, the maintenance timeframe does not begin for all project areas at the time of the maximum. If the time of the maximum plus .3 years is used for the beginning of the maintenance timeframe then the following equations are derived:

$$\text{Maintenance} = .56\ (K_1 - K_2)$$

$$\text{Development} = .44\ (K_1 - K_2)$$

The factor necessary to add maintenance costs to the development cost is given by:

$$\text{Maintenance} = (.56 + .44)\ /.44 = 2.25$$

Thus, we have developed a useable manpower model that can be expressed in terms of categories of cost and associated manpower, a percent model based on the categories and a generalized cost model shown in Figure 4 which uses factors to arrive at total costs driven by direct CR costs.

ACTIVITY	COST CATEGORY							ALGORITHM
	I	II	III	IV	V	VI	VII	
DEVELOPMENT THROUGH INITIAL SYSTEM RELEASE	X	X	X	X				$2.5\ (CR_D)$
DEVELOPMENT AND MAINTENANCE	X	X	X	X		X		$2.25\ (2.5(CR_D)) = 5.6(CR_D)$
TOTAL PROJECT COST	X	X	X	X	X	X	X	$5.6\ (CR_D)$ + CAT. V + CAT. VII

Figure 4. Generalized Cost Model

Development of a Manpower Model Based on Quality

As with any modeling exercise, this one is initiated by collecting data. All data is collected by releases of Shuttle onboard PASS. Since the data for Release 19 is incomplete only the data for Releases 16 and 17/18 is used.

Data Collection

All the data gathered is from the Project Development Plan and data bases which support the plan or from Project Office history files. Requirements Change Request (CR) data is collected as the total number by release. Discrepancy Report (DR) data reflects the total number by release divided into those which require a code fix and those which do not. It is important to note that the "No Fix" category includes user notes, waivers, and other categories which have the potential of becoming "Fix" DR's in the future. The largest group in the "No Fix" category, however, are the DR's which are simply not PASS problems but simulator, user or misinterpretation errors. The manpower data is divided into base work prior to system delivery and maintenance work after delivery. Each of these categories is subdivided into work performed by the development and verification groups. The data collected is then used to generate the data comparison table presented as Figure 5. The first four columns of data in the table represent the data collected from the project. The remaining five columns show relationships derived from the ratios of the data elements.

RELEASE/ AREA	CRs	DEV MM	DRs	MAINT MM	DR/ CR	MM/ CR	MM/ DR	MAINT/ DEV	DR/ MM DEV
R-16									
DEV	–	6228	–	817	–	3.6	.2	.1	–
VERIF	–	3179	–	618	–	1.9	.1	.1	–
FIX	–	–	2371	–	1.4	–	–		.4
NO FIX	–	–	2289	–	1.3	–	–	–	.4
TOTAL	1725	9407	4660	1435	2.7	5.5	.3	.2	.8
R-17/18									
DEV	–	1286	–	381	–	1.6	.2	.2	–
VERIF	–	972	–	3	–	1.2	0	0	–
FIX	–	–	951	–	1.2	–	–	–	.7
NO FIX	–	–	1449	–	1.9	–	–	–	1.1
TOTAL	782	2258	2400	384	3.1	2.8	.2	.2	2.0

Figure 5. Shuttle Onboard PASS Data Comparison Table

DR Analysis

The next step in the process is to analyze the DR and maintenance data to generate the Average Cost of "Fix" and "No Fix" DR's. Beginning with Release 16 data the initial action required is to remove the technical support manpower from the maintenance manpower by dividing by 2.25 (the technical support factor less the project office). Then knowing that, on the average, five times as much effort is spent on "Fix" DR's as "No Fix" DR's, the following equation can be written:

$$1435/2.25 = 2371\,x + 2289\,(x/5)$$

The solution of the equation renders the result that each "Fix" DR cost 4.50 mandays total or 2.25 mandays for each of development and verification. Performing the same analysis for Release 17/18, using development maintenance only since verification maintenance was not required, the equation yields 2.7 mandays of development effort for each DR. Averaging these figures the following direct impact values are derived:

- The direct impact of a DR which is fixed is:
  - 2.5 md FOR DEVELOPMENT
  - 2.5 md FOR VERIFICATION
- The direct impact of a DR which is not fixed is:
  - .5 md FOR DEVELOPMENT
  - .5 md FOR VERIFICATION

DR Prevention

In the Shuttle onboard PASS project DR's are written for a problem only after the software causing the problem has been placed on the Master System. Once a DR is written, all areas of the project become involved in its closure regardless of whether it is a problem or not. Hence, there are two possibilities for reducing the number of DR's. The first is to enhance the requirements analysis activities to give a reliable point of coordination before the DR is written. This subject will not be treated further in this study but will be the object of a later study. The second possibility, and the main object of this study, is to enhance the development process prior to the master system build. The two ways to accomplish this are to enhance requirements analysis activities and design and code reviews early in the initial development cycle. Requirements analysis should be enhanced to improve the quality of CR's before implementation begins, shepherd CR's through the development life cycle, help specify level 1 and 2 tests, review level 1 and 2 test results and support design and code reviews. Design and code reviews could be improved by allowing more time for the reviews, improving checklists and review documentation, providing for improved and dedicated review moderators and to require wider involvement from functional areas of the project.

Modifying the Current Manpower Model

The approach, then, to modifying the current manpower model is to consider which areas of the model should be increased for DR prevention, which areas of cost will benefit from having fewer DR's to deal with and which areas contain the skills required to enhance early development. These areas should be modified accordingly to create an incremental release model which assumes an enhanced early development and fewer DR's - in other words, a model which assumes and also assures quality. Figure 6 depicts the process of modifying the current manpower model which is reflected under the "Old %" column. The modifications are listed under the "Δ %" column. It should be noted that 3% is taken from each of Level 6 and 7 verification and redirected toward the early development activity. This results in no change to the overall project development model. This is consistent with the introductory assumption that quality does not cost more. The final two columns show the current and quality manpower models in terms of Release 19 manmonths.

CAT.	AREA	FUNCTION	OLD %	Δ%	NEW %	R-19 EXAMPLE	NEW EXAMPLE
I	DEV.	DIRECT EST.	16	.4(16)=+6	22	197	274
	VERIF.	DIRECT EST.	14		14	173	173
II	DEV.	RA	1		1	13	13
		L3	2		2	26	26
		SA	1		1	9	9
		SAr	2		2	27	27
		DR	-		-	0	0
II	VERIF.	RA	-		-	0	0
		ST/AU	2		2	19	19
		CF	1		1	8	8
		S MEAS	1		1	14	14
		L7	11	-3	8	139	100
		PRE CI DR's	3	-3	-	38	0
III	DEV.	PREP	3		3	30	30
IV	All	M & S	11		11	135	135
		CS	6		6	70	70
V	SFO	B & I	11		11	140	140
		RES MGT	8		8	100	100
		CM/DM	7		7	80	80
			100	0	100	1218	1218

Figure 6. Modifying the Current Manpower Model

Model Change Justification

The amount of manpower relocated in the model change is not arbitrarily selected. The direct development manpower is increased by 40%. Twenty percent is added to account for increased requirements analysis. This figure is based on early Release 16 history when a separate requirements analysis group was maintained in the development organization. This reflects a return to heavy emphasis on requirements analysis as a front end process of the project. The remaining 20% is added to the direct development manpower to account for enhanced design and code reviews. This figure is based on a comparison of the old and new review processes in terms of increased elapsed time of the reviews, broader involvement of the project in the reviews and increased documentation and tracking.

To account for the 40% increase in development a corresponding decrease must occur elsewhere since quality does not cost more. The two areas selected to sustain the reduction are Level 6 DR support and Level 7 Test. Each of these reductions is examined individually.

Savings Due to Fewer DR's

Better initial quality should be reflected in the project as fewer DR's during the development and verification process. The task then is to quantify the projected savings. To do this the following procedure is used. By examining Release 19 data it is noted that there are 235 CR's included in the release. Based on the Release 16 and 17/18 DR/CR ratios it can be projected that there will be 705 DR's during the completion of the development life cycle. Of these only 40% or 282 should be code changes. If we increase the development budget to improve the initial quality of the software going to each build, a decrease in DR's after the builds should be expected. It should also be expected that not all DR's will be eliminated. Selecting 50% of DR's as a target for elimination, a projected savings can be calculated as:

(282 DR's) (.5) (2.5 md/DR) = 18 man months

Including technical support (without the project office) a savings of 41 man months can be projected in the verification area. This amount alone justifies the reduction to the level 6 test function. However, the development area will experience a similar 41 man month decrease during the verification support time frame. This means that our model is conservative. The goal is to reduce the DR number by 50% but a 25% reduction will enable the development and verification areas to "break even".

Savings Due to Rephasing Skills

The skills necessary to enhance the requirements analysis task currently reside in the level 7 test group in the verification organization. Rephasing these skills to the requirements analysis task must therefore be justified. In the current model the level 7 task took 11% of the release manpower. Examination of Release 17/18 data, however, shows that of the 674 DR's found by verification, only 35 were found by the level 7 test group. Of these, only 12 were significant flight software problems. It appears obvious then that the recommended rephasing of skills would be not only feasible but desirable. Three percent of the release manpower would be adequate to perform the requirements analysis task which would be reflected in fewer DR's reaching the master build. This would leave 8% to perform level 7 testing which could be performed in a more stable enviornment due to fewer DR's during the performance time frame.

The completion of this exercise concludes the justification of the manpower movements to create the quality model. The % model given in Figure 6 can then be used to project manpower for Release 20 and beyond.

A Generalized Quality Model

A % model alone does not completely satisfy the project need. Consequently, the procedure used earlier in the paper to generate a generalized model is used again with the quality model. By examining the costs through category IV, the following calculations can be made:

$$\text{DIRECT CR COSTS} = CR_D = \text{CATEGORY I}$$

$$\text{INDIRECT CR COSTS} = CR_I = \text{CATEGORIES II - IV}$$

$$\text{FACTOR} = \frac{CR_D + CRI}{CR_D} = \frac{I+II+III+IV}{I}$$

$$\text{FACTOR} = \frac{447 + 216 = 30 = 205}{447} = 2.0$$

Again this factor can be used to calculate the development and verification costs directly driven by CR's.

Once again maintenance, which is not included in this factor must be considered. By using generalized equations for Rayleigh curves with and without a CR the following equations are derived:

$$\text{DEVELOPMENT} = .4\ (K_1 - K_2)$$

$$\text{MAINTENANCE} = .6\ (K_1 - K_2)$$

But part of the maintenance time frame in the development area is now devoted to verification support and maintenance begins at the completion of the verification cycle. By separating the verification support from the maintenance as shown in Figure 7 the equations are modified as follows:

$$\text{CAPABILITIES DEVELOPMENT} = .4\ (K_1 - K_2)$$

$$\text{VERIFICATION SUPPORT} = .1\ (K_1 - K_2)$$

$$\text{MAINTENANCE DEVELOPMENT} = .5\ (K1 - K_2)$$

But the verification support contains some verification costs. Correcting for this the equations become:

$$\text{CAPABILITIES DEVELOPMENT} = .43\ (K_1 - K_2)$$

$$\text{VERIFICATION SUPPORT} = .07\ (K_1 - K_2)$$

$$\text{MAINTENANCE DEVELOPMENT} = .5\ (K_1 - K_2)$$

Thus the factors necessary to add verification support and maintenance costs to the costs directly driven by CRs are:

$$\text{VERIFICATION SUPPORT} = \frac{.43+.07}{.43} = 1.15$$

$$\text{MAINTENANCE} = \frac{.5+.5}{.5} = 2.00$$

Hence, the generalized model presented in Figure 8 is complete.

ORIGINAL PAGE IS
OF POOR QUALITY

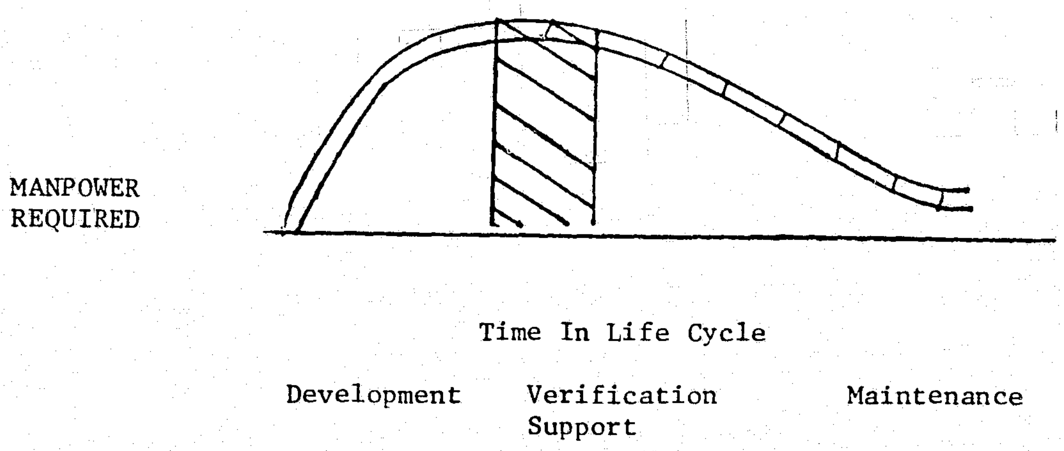


Figure 7. Division of Life Cycle Manpower

ACTIVITY		COST CATEGORY							ALGORITHM
		I	II	III	IV	V	VI	VII	
DEVELOPMENT AND VERIFICATION ROM'S		X	X	X	X				$2.0\ (CR_D)$
TOTAL ROM'S	VERIFICATION SUPPORT								$1.15(2.0(CR_D)) = 2.3(CR_D)$
	MAINTENANCE	X	X	X	X		X		$2.0(1.15(2.0(CR_D)))=2.0(2.3(CR_D))= 4.6\ CR_D$
RUNOUTS AND PROPOSALS		X	X	X	X	X	X	X	$4.6\ (CR_D)$ + CAT. V + CAT. VII

Figure 8. Generalized Quality Model

Finally, it should be noted that if better initial quality is introduced into the software system then the cost of maintenance should go down. Again, the quality model is conservative since it did not take this into account. As actuals are accrued, then the model can be tuned to reflect those actuals.

Extension to a Manpower Allocation Model

There comes a time in the life of most projects when they can no longer be viewed as one Rayleigh curve but as one curve followed by a series of smaller curves each of which represents a release. A generic graph of this time frame is reflected in Figure 9. As a steady state, uniform set of releases is reached the total manpower line tends to a steady state and the maintenance line also tends to a steady state. Studies using groups of curves in this fashion have shown that the steady state maintenance level reached is 25% of the total steady state level. Based on the study conclusion the first allocation of manpower should be:

DEVELOPMENT	75%
MAINTENANCE	25%

To allocate below this major division, a % model base on organization rather than category is required. Performing this reorganization the quality % model by organization is given in Figure 10.

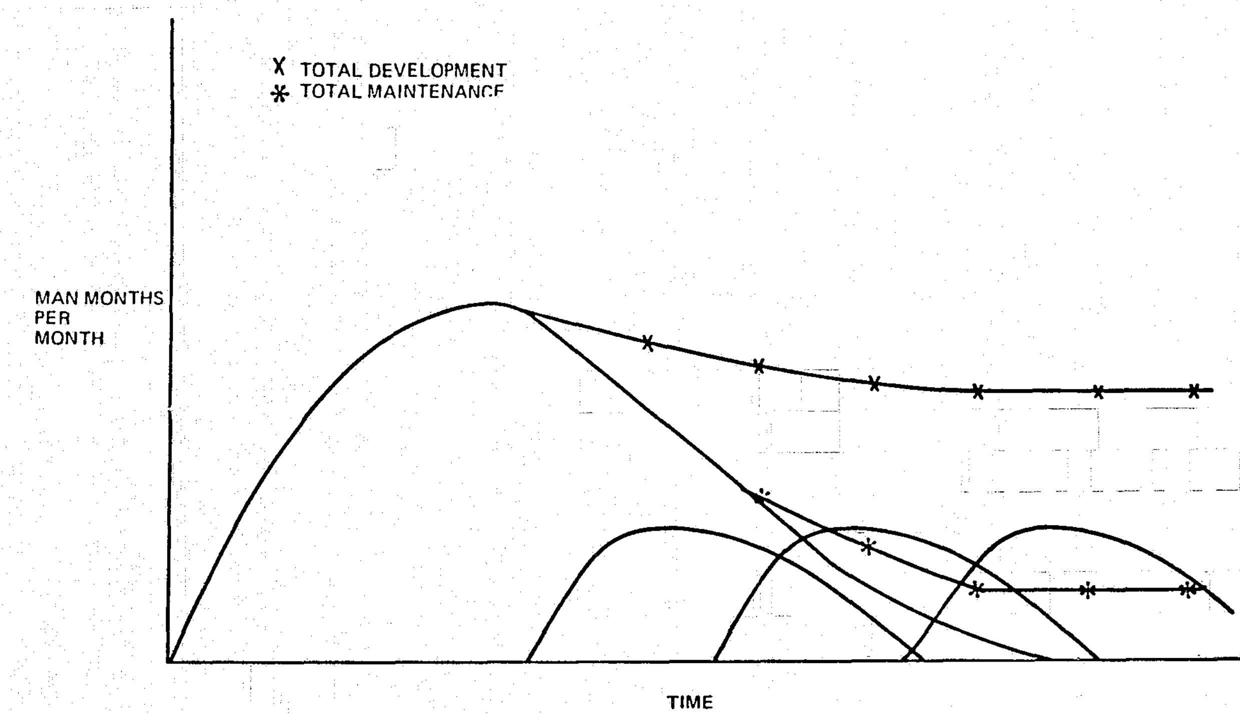


Figure 9. Manpower in the Incremental Release Timeframe

ORIGINAL PAGE IS OF POOR QUALITY

ORG.	CAT.	FUNCTION	NEW %		NEW EXAMPLE	
DEV.	I	DEV. DIRECT EST.	22	(3)	274	
DEV.	I	VERIF. DIRECT EST.	14		173	
DEV.	II	VERIF. ST/AU	2		19	
DEV.	II	VERIF. CF	1		8	
DEV.	III	PREP	3		30	
DEV.	IV	M & S (PART)	8		98	
				50		602
SE	II	RA	1		13	
SE	II	L3	2		26	
SE	II	SA	1		9	
SE	II	SAr	2		27	
SE	II	SMEAS	1		14	
SE	II	L7	8		100	
SE	IV	M & S (PART)	3		37	
				18		226
P.O.	IV	CS	6		70	
				6		70
OTHER	V	B & I	11		140	
OTHER	V	RES MGMT	8		100	
OTHER	V	CM/DM	7		80	
				26		320
			100	100	1218	1218

Figure 10. Quality % Model by Organization

Since in a incremental release time frame all activities of the project are progressing simultaneously, a time slice allocation can be made to each activity. The allocation to development activities is shown in Figure 11. The legend of this figure is the same as the % model with the following exceptions. Software development is divided into capabilities development (CD) which is ongoing development of the master system and verification support (VS) which handles error correction during the verification time frame. Half of the verification support is set aside as a buffer to handle late, mandatory CR's required by near term flights. This allows the capabilities development to be scheduled and worked without interruption. It should be noted that a comparable amount of buffer must be set aside for verification.

Including maintenance and development into one allocation model the model depicted in Figure 12 is attained.

CD	0I5	0I6	0I7	18%
VS	0I4	0I5	0I6	9%
L6	0I4	0I5	0I6	20%
L7		0I4 AND 0I5		9%
RA	0I5/0I4	0I6/0I5	0I7/0I6	5%
L3, SA, SAR	0I5/0I4	0I6/0I5	0I7/0I6	7%
CS				6%
B&I, CM/DM, RES MGT.				26%
				100%

Figure 11. Development Manpower Allocation Model

CD	OI5	OI6	OI7	14%
VS	OI4	OI5	OI6	7%
L6	OI4	OI5	OI6	15%
L7		OI4 AND OI5		7%
RA	OI5/OI4	OI6/OI5	OI7/OI6	4%
L3, SA, SAR	OI5/OI4	OI6/OI5	OI7/OI6	5%
CS				4%
B&I, CM/DM, RES MGT.				19%
MAINT				25%
				100%

Figure 12. Total Manpower Allocation Model

Buffer Management

Since the idea of CR buffers in the verification support and verification areas has been introduced, some attention must be paid to the management of those buffers. From Figure 10 it can be seen that the factor which represents organizational technical support (overhead) should be:

$$\text{DEVELOPMENT FACTOR} = \frac{\text{CRD} + \text{CRDI}}{\text{CRD}} = \frac{19 + 8}{19} = 1.4$$

$$\text{VERIFICATION FACTOR} = \frac{\text{CRV} + \text{CRVI}}{\text{CRV}} = \frac{14 + 6}{14} = 1.4$$

The buffer management factor then is 1.4 for both development and verification.

The ground rules for buffer management can now be stated as:

- Begin with defined CR buffers in the verification support and L6 verification allocations.
- Adjust the buffers to account for the final build in each operational increment which is left open for DR corrections.
- On a continuing basis account for change by:
  - Adjusting both buffers by the direct CR estimates marked up by the buffer management factor.
  - Adjusting for actuals overruns and underruns if required.

Model Tracking

An initial model is good only for a first projection and allocation of manpower. In order to make the model useable on a continuing basis, actual data relating to the projected data should be tracked and used to validate and/or modify the model. For this model two types of data tracking are required. The first requirement is that the quality projected is attained. The most appropriate measure of quality appears to be DR's per manmonth of development. The DR count is the number of DR's which require fixes written against the builds contained in the increment or release. The manpower number is the total manpower expended during capabilities development and verification support. This measurement is initiated at the end of the first capabilities development phase and terminates at CI for a given release. As shown in Figure 13 an alert line has been established from Release 17/18 experience. If the measurement violates the alert line then an effort will be made to determine if the initial quality is not what had been projected. The maximum alert line is 75% of the Release 17/18 Fix DR per manmonth of development number found in Figure 5. A graph of this measurement is reviewed in the project and with the customer on a periodic basis. The second type of data tracking required is that expenditures by major function must be examined on a release basis and the data used to tune the model. To assist in this task a form has been developed to allow for the recording of schedule, manpower projections, buffer management, actuals and quality tracking data for a given release. This form is presented in Figure 14. Since it contains scheduling, tracking, and completion data it could be useful as a release management tool as well as a software cost engineering tool.

ORIGINAL PAGE IS
OF POOR QUALITY

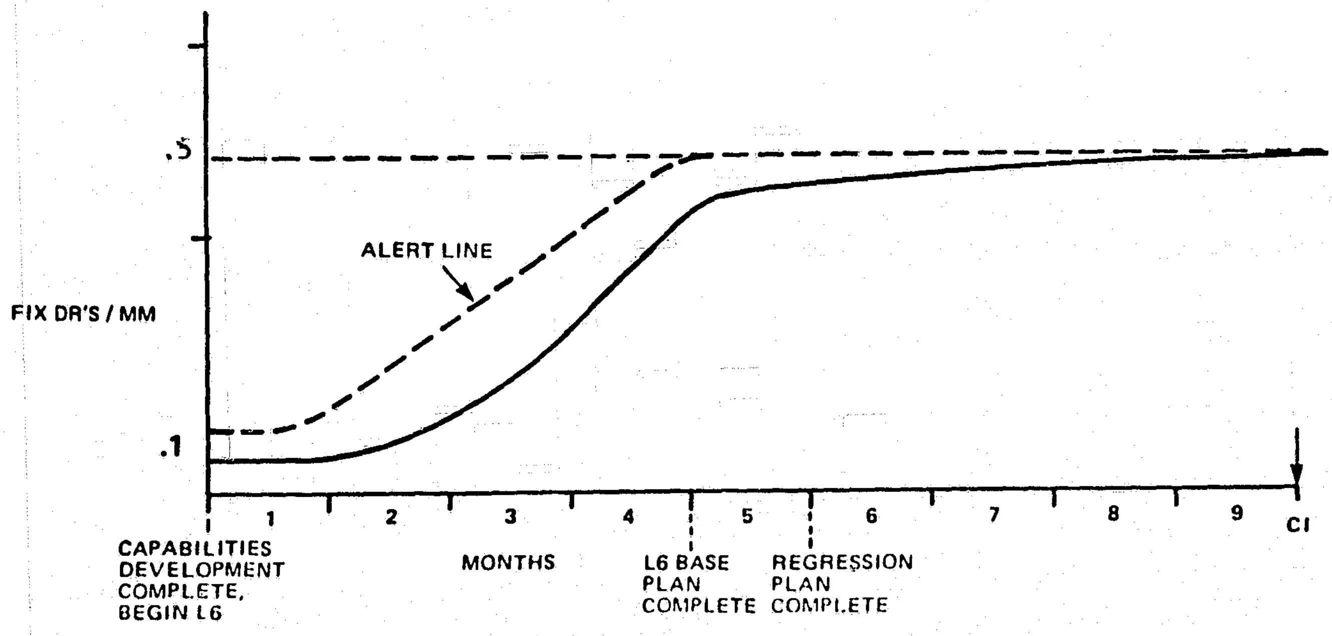
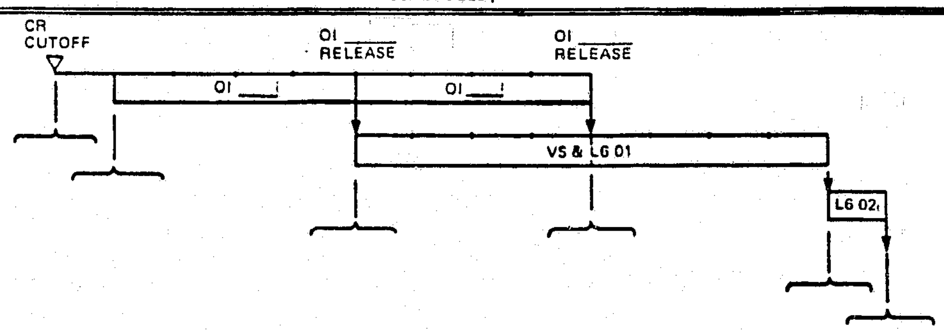


Figure 13. Software Quality Tracking

ORIGINAL PAGE IS OF POOR QUALITY

RELEASE NUM. ______

SCHEDULES:



PROJECTIONS

PROJECTION DATE ______ CR BASELINE DATE ______

NO. OF CR'S ______ DIRECT CR ESTIMATE ______ PROJECTED DR'S ______

MANPOWER ALLOCATION

FUNCTION	MM	%	FUNCTION	MM	%	FUNCTION	MM	%
CD VS L6			L7 RA L3, SA, SAR			CS B&I, CM/DM RES MGT		
TOTAL			TOTAL			TOTAL		

TOTAL RELEASE = ______

BUFFER IN VS ______ BUFFER IN L6 ______ BUFFER CUTOFF ______

BUFFER MANAGEMENT

DATE	CR'S	COST	BUFFER	DATE	CR'S	COST	BUFFER	DATE	CR'S	COST	BUFFER

CR'S ABSORBED ______ BUFFER USED ______ BUFFER REMAINING ______

ACTUALS

ACTUALS DATE ______ CR BASELINE DATE ______ DR BASELINE DATE ______

NO. OF CR'S ______ DIRECT CR EST. ______ NO. OF DR'S ______

MANPOWER ACTUALS

FUNCTION	MM	%	FUNCTION	MM	%	FUNCTION	MM	%
CD VS L6			L7 RA L3, SA, SAR			CS B&I, CM/DM RES MGT		
TOTAL			TOTAL			TOTAL		

TOTAL RELEASE ______ DELTA FROM PROJECTION ______

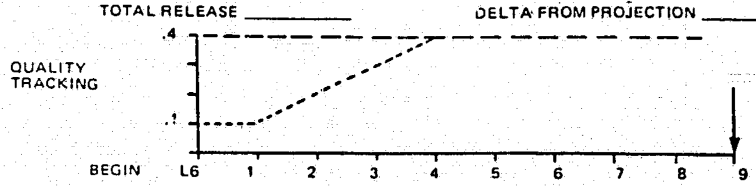


Figure 14. Release Tracking Form

Model Sensitivities

It is important to point out that the generalized model is sensitive to the direct CR estimates. The accuracy of this basic building block of most cost models is important to any project which has a significant amount of ongoing change. Also, since the quality model emphasises early development, the increased impact due to enhanced design and code reviews, requirements analysis and pre-build testing need to be included in the direct CR costs. The model is also sensitive to the number and cost of the DR's generated during development and test of a release. The tracking of release quality will keep a proper project focus on this sensitivity.

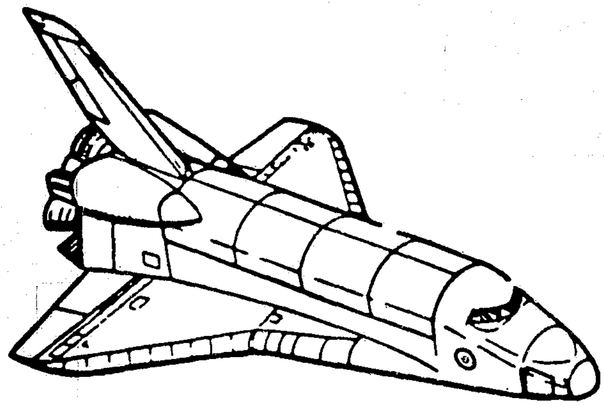
Summary

Since the total cost of a software development project is directly related to the initial quality built into the software, it becomes a necessity to project manpower to attain that quality. The basic approach takes a current manpower model and, by reflecting greater expenditure on elements which are known to improve initial quality, generates a new manpower model designed to generate quality software. Since the model must be used in an ongoing incremental release environment, an allocation model is developed to allocate manpower across a project's organization. Finally, a procedure is developed to allow the tracking of data for model validation and modification.

REFERENCES

1. Rone, K. Y., "General Project Cost Procedures", ____________ 1982, FSD Houston.

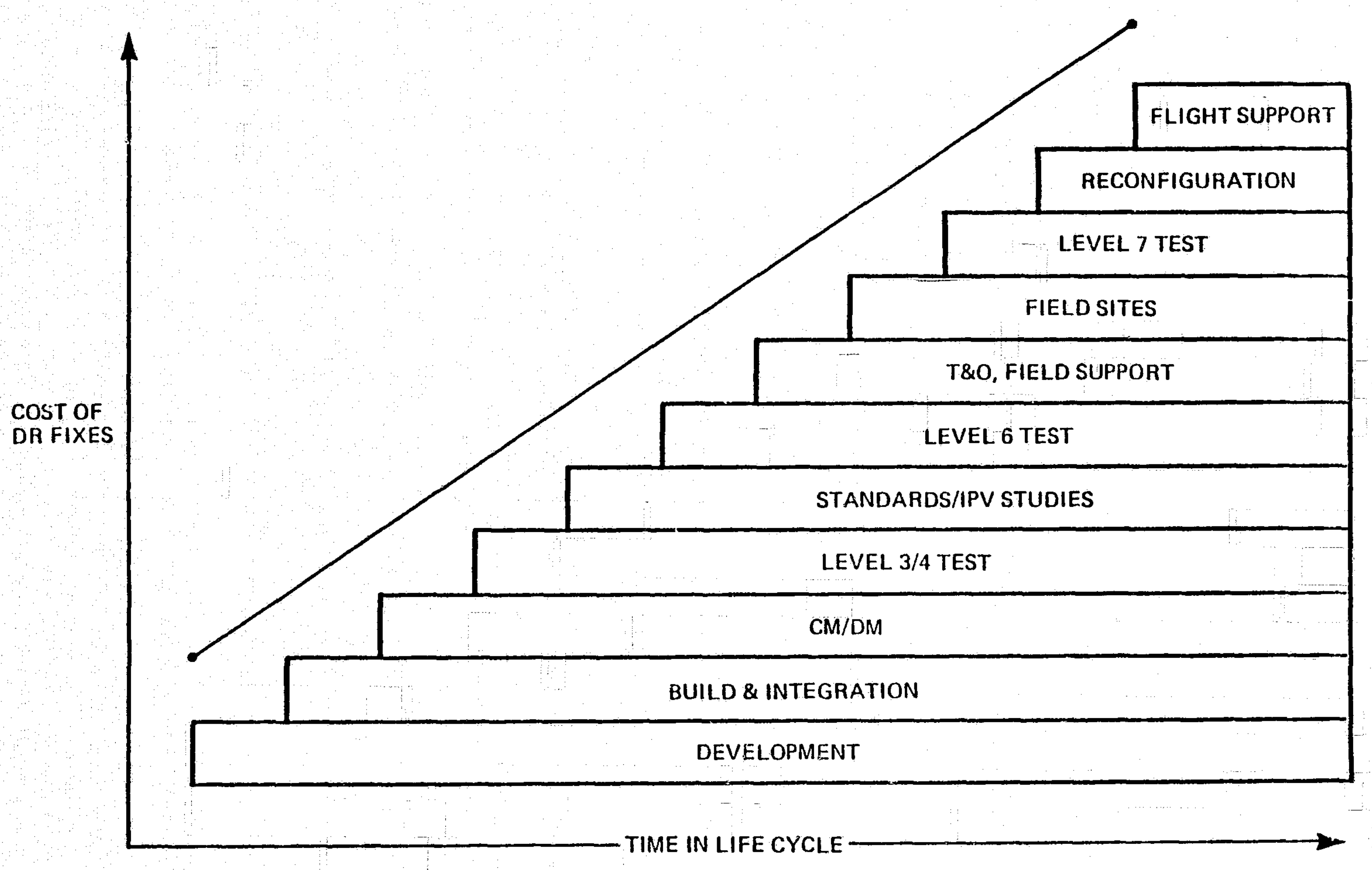
PROJECTING MANPOWER
TO ATTAIN QUALITY

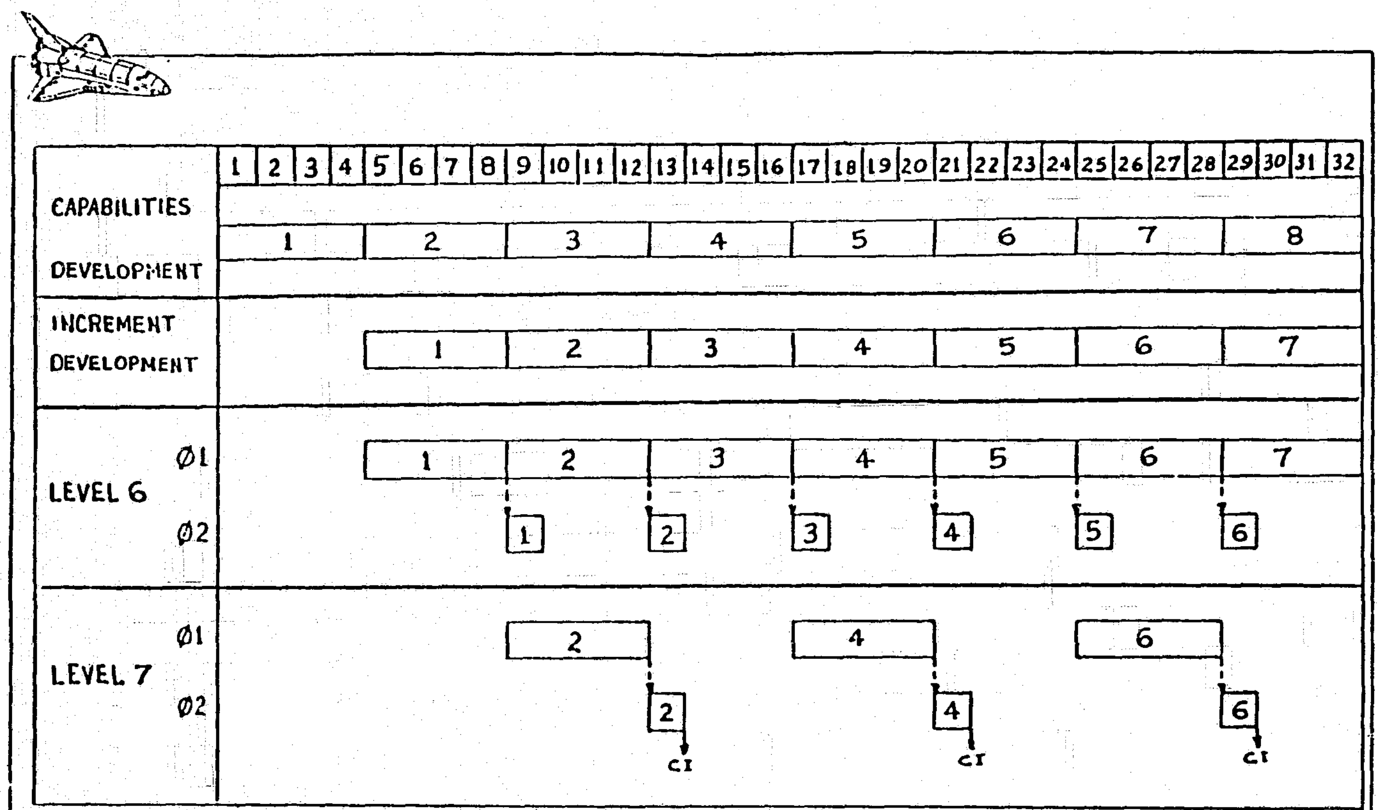


SPACE SHUTTLE PROGRAMS

K. Y. RONE
FEDERAL SYSTEMS DIVISION
HOUSTON, TEXAS

IBM
Federal Systems Division
1322 Space Park Drive, Houston 77058


COST OF DR FIXES
FLIGHT SUPPORT
RECONFIGURATION
LEVEL 7 TEST
FIELD SITES
T&O, FIELD SUPPORT
LEVEL 6 TEST
STANDARDS/IPV STUDIES
LEVEL 3/4 TEST
CM/DM
BUILD & INTEGRATION
DEVELOPMENT
TIME IN LIFE CYCLE



INCREMENTAL RELEASE PROCESS

IBM SPACE SHUTTLE PROGRAMS

ORIGINAL PAGE IS
OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

PURPOSE

- THE INCREMENTAL RELEASE PLAN IS A PROJECT PLANNING METHODOLOGY WHICH WILL RESULT IN HIGHER QUALITY FLIGHT SOFTWARE RELEASES:
  - SMALLER, MORE FREQUENT DEVELOPMENT INCREMENTS
  - MORE COMPREHENSIVE TESTING PRIOR TO FIELD DELIVERY
- MANPOWER IS BEING REPHASED ON THE PROJECT TO PLACE MORE EMPHASIS ON REQUIREMENTS ANALYSIS, DESIGN/CODE REVIEWS AND PRE-BUILD TEST
- THE PURPOSE OF THIS STUDY IS TO MODIFY THE CURRENT MANPOWER MODEL TO REFLECT THESE CHANGES AND PROVIDE A USABLE ONGOING MODEL FOR CLASS 1 WORK IN THE FUTURE

ORIGINAL PAGE IS
OF POOR QUALITY

APPROACH

- START WITH CURRENT MANPOWER MODEL
  - MW DEVELOPMENT ESTIMATE
  - ROM = 2.25(2.5(1.8(1 MW))) = 5.6(1.8(1 MW))
    = 10.1 MW

- BEGINNING WITH THE BASIC ASSUMPTION THAT IMPROVING QUALITY DOES NOT COST MORE, WE DEVELOP A MODEL WHICH
  - ADDS 40% TO DEVELOPMENT FOR BETTER REQUIREMENTS ANALYSIS AND BETTER DESIGN AND CODE REVIEWS
  - THIS IS EQUIVALENT TO ADDING 1 DAY FOR R.A. AND 1 DAY FOR REVIEWS
  - THIS WILL RESULT IN FEWER DRs SO EVEN THOUGH THE DEVELOPMENT COST IS UP THE TOTAL COST IS THE SAME
  - THE DEVELOPMENT COST WOULD BE 1.4 MW WHILE THE DIRECT L6 COST IS THE SAME (.8)
  - ROM = 2.0 (1.15(2.0(2.2 MW))) = 4.6 (2.2 MW) = 10.1 MW

- THIS IS A CONSERVATIVE APPROACH FOR THE FIRST MODEL WHICH CAN BE MODIFIED BASED ON ACTUAL DATA

- THIS MODEL IS INTENDED FOR USE WHEN THE INCREMENTAL RELEASE STRATEGY REACHES STEADY STATE

RELEASE/ AREA	CRs	DEV MM	DRs	MAINT MM	DR/ CR	MM/ CR	MM/ DR	MAINT/ DEV	DR/ MM DEV
R-16									
DEV	-	2183	-	817	-	1.3	.2	.2	-
VERIF	-	1718	-	618	-	1.0	.1	.2	-
FIX	-	-	2371	-	1.4	-	-	-	1.1
NO FIX	-	-	2289	-	1.3	-	-	-	1.1
TOTAL	1725	3901	4660	1435	2.7	2.3	.3	.4	2.2
R-17/18									
DEV	-	1286	-	381	-	1.6	.2	.2	-
VERIF	-	972	-	3	-	1.2	0	0	-
FIX	-	-	951	-	1.2	-	-	-	.7
NO FIX	-	-	1449	-	1.9	-	-	-	1.1
TOTAL	782	2258	2440	384	3.1	2.8	.2	.2	2.0

R16 AND R17/18 DR ANALYSIS

- The maintenance manpower includes technical support for both Development and Verification.

- Taking that support out of R16 we get 1435/2.25 = 638mm

- Knowing that we spend, on the average, 5 times as much effort on fix DRs as no fix DRs we can write:

638 mm	=	2371X+2289(X.5)
3190	=	11855X+2289X
3190	=	14144X
X	=	.23mm
	=	4.50md
	or	2.25md/Fix DR for each of Development and Verification

- Performing the same analysis for R17/R18 (for Development only since Verification maintenance 0 since CI is so close to flight) we get:

  2.66 md/Fix DR Development

- If we average these numbers we arrive at:

  - The direct impact of a DR which is fixed is:
    - 2.5 md for Development
    - 2.5 md for Verification
  - The direct impact of a DR which is not fixed is:
    - .5 md for Development
    - .5 md for Verification

DR PREVENTION

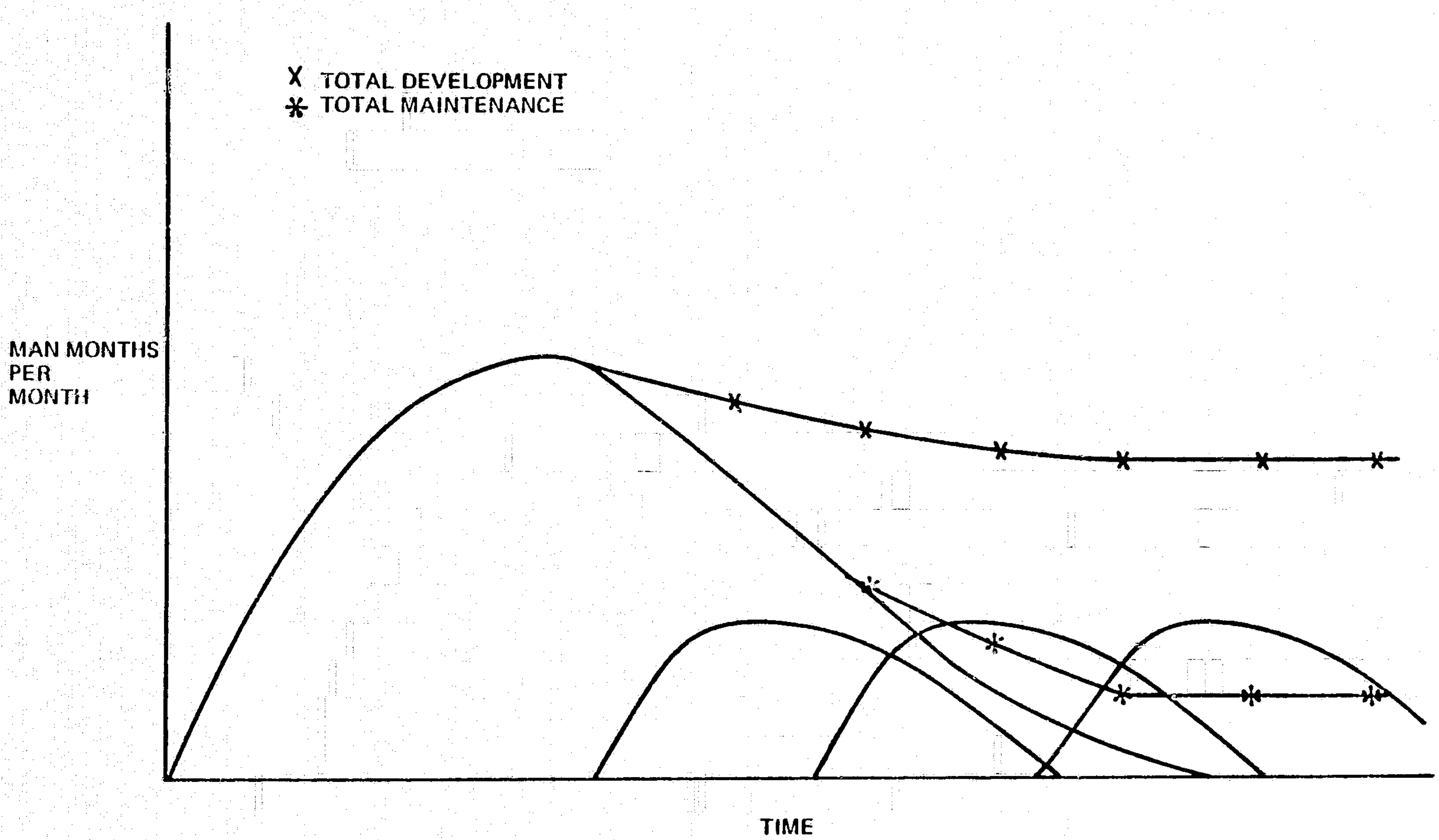
- DRs ARE WRITTEN FOR A PROBLEM ONLY AFTER THE SOFTWARE CAUSING THE PROBLEM HAS BEEN PLACED ON THE MASTER SYSTEM
- ONCE A DR IS WRITTEN, ALL AREAS OF THE PROJECT BECOME INVOLVED IN ITS CLOSURE REGARDLESS OF WHETHER IT IS A PROBLEM OR NOT
- HENCE, THERE ARE TWO POSSIBILITIES FOR REDUCING THE NUMER OF DRs:
    - ENHANCE THE REQUIREMENTS ANALYSIS ACTIVITY TO GIVE A POINT OF COORDINATION BEFORE THE DR IS WRITTEN
    - ENHANCE THE DEVELOPMENT PROCESS PRIOR TO THE BUILD FOR THE MASTER SYSTEM
        - ENHANCE REQUIREMENTS ANALYSIS
            - IMPROVE QUALITY OF CR BEFORE IMPLEMENTATION BEGINS
            - COORDINATE CRs
            - SPECIFY L1/L2 TESTS
            - REVIEW TEST RESULTS
            - SUPPORT D&C REVIEWS
        - ENHANCE DESIGN & CODE REVIEWS
            - SPEND MORE TIME ON REVIEW
            - IMPROVE CHECKLISTS, DOCUMENTATION
            - IMPROVED/DEDICATED MODERATORS
            - UNDER INVOLVEMENT ON PROJECT

ORIGINAL PAGE IS
OF POOR QUALITY

CAT.	AREA	FUNCTION	OLD %	Δ %	NEW %	R-19 EXAMPLE	NEW EXAMPLE
I	DEV.	DIRECT EST.	16	.4(16)=+6	22	197	274
	VERIF.	DIRECT EST.	14		14	173	173
II	DEV.	RA	1		1	13	13
		L3	2		2	26	26
		SA	1		1	9	9
		SAr	2		2	27	27
		DR	-		-	0	0
II	VERIF.	RA	-		-	0	0
		ST/AV	2		2	19	19
		CF	1		1	8	8
		S MEAS	1		1	14	14
		L7	11	-3	8	139	100
		PRE CI DR's	3	-3	-	38	0
III	DEV.	PREP	3		3	30	30
IV	All	M & S	11		11	135	135
		CS	6		6	70	70
V	SFO	B & I	11		11	140	140
		RES MGT	8		8	100	100
		CM/DM	7		7	80	80
			100	0	100	1218	1218

ORIGINAL PAGE IS
OF POOR QUALITY

DEVELOPMENT OF INCREMENTAL
RELEASE MANPOWER ALLOCATION MODEL


X TOTAL DEVELOPMENT
* TOTAL MAINTENANCE
MAN MONTHS PER MONTH
TIME

ORIGINAL PAGE IS
OF POOR QUALITY

MANPOWER ALLOCATION MODEL

- BASED ON THE STUDY CONCLUSION, THE FIRST ALLOCATION OF MANPOWER SHOULD BE:

DEVELOPMENT	75%
MAINTENANCE	25%

- IN ORDER TO ALLOCATE BELOW THIS MAJOR DIVISION, A % MODEL BASED ON THE NEW ORGANIZATION IS REQUIRED

C - 84

NEW ORGANIZATION % MODEL

ORG.	CAT.	FUNCTION	NEW %		NEW EXAMPLE	
DEV.	I	DEV. DIRECT EST.	22	(3)	274	
DEV.	I	VERIF. DIRECT EST.	14		173	
DEV.	II	VERIF. ST/AU	2		19	
DEV.	II	VERIF. CF	1		8	
DEV.	III	PREP	3		30	
DEV.	IV	M & S (PART)	8		98	
				50		602
SE	II	RA	1		13	
SE	II	L3	2		26	
SE	II	SA	1		9	
SE	II	SAr	2		27	
SE	II	SMEAS	1		14	
SE	II	L7	8		100	
SE	IV	M & S (PART)	3		37	
				18		226
P.O.	IV	CS	6		70	
				6		70
OTHER	V	B & I	11		140	
OTHER	V	RES MGMT.	8		100	
OTHER	V	CM/DM	7		80	
				26		320
			100	100	1218	1218

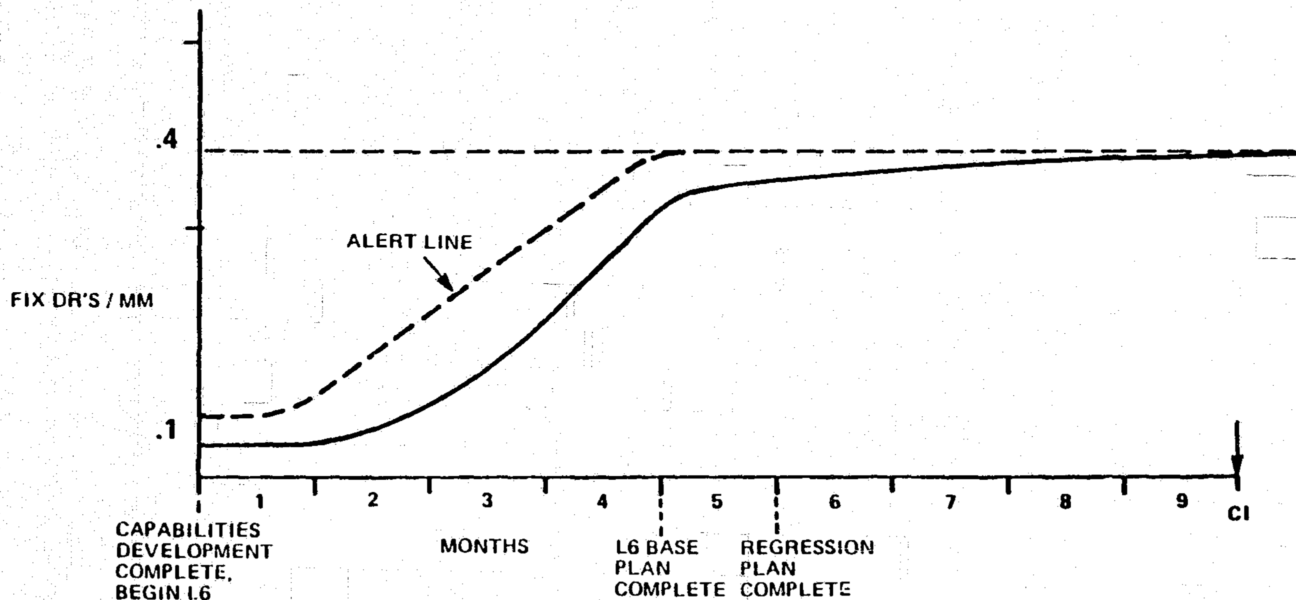
TOTAL MANPOWER ALLOCATION MODEL

CD	OI5	OI6	OI7	14%
VS	OI4	OI5	OI6	7%
L6	OI4	OI5	OI6	15%
L7		OI4 AND OI5		7%
RA	OI5/OI4	OI6/OI5	OI7/OI6	4%
L3, SA, SAR	OI5/OI4	OI6/OI5	OI7/OI6	5%
CS				4%
B&I, CM/DM,. RES MGT.				19%
MAINT				25%
				100%

SOFTWARE QUALITY TRACKING

- MOST APPROPRIATE MEASURE IS DR'S PER MAN MONTH
  —BASE MANPOWER NUMBER IS TOTAL EXPENDED DURING CAPABILITIES DEVELOPMENT AND CAPABILITIES REFINEMENT
  —DR COUNT WILL BE THE DR'S WHICH REQUIRE FIXES WRITTEN AGAINST THE BUILDS CONTAINED IN THE INCREMENT
- INITIATE MEASUREMENT AT THE END OF FIRST CAPABILITIES DEVELOPMENT PHASE, TERMINATE AT CI
- ESTABLISH "ALERT" LINE FROM R18/R19 EXPERIENCE
- REVIEW PERIODICALLY WITH SSD

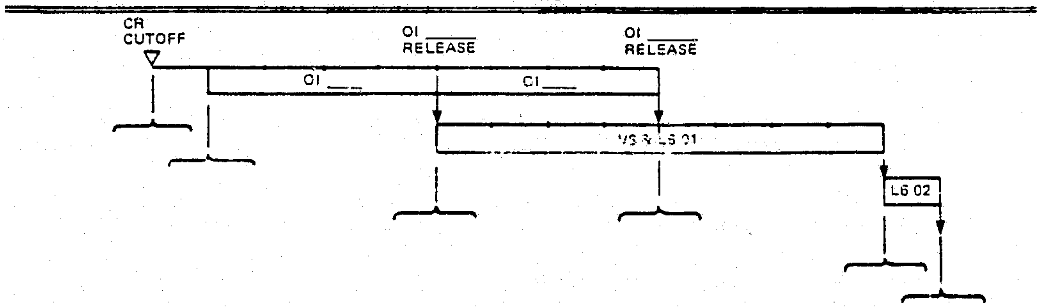
  —RELEASE STATUS MEETING



ORIGINAL PAGE IS
OF POOR QUALITY

RELEASE NO. ______ OI'S ______ RELEASE MGR. ______

SCHEDULES



PROJECTIONS

PROJECTION DATE ______ CR BASELINE DATE ______

NO. OF CR'S ______ DIRECT CR ESTIMATE ______ PROJECTED DR'S ______

MANPOWER ALLOCATION

FUNCTION	MM	%	FUNCTION	MM	%	FUNCTION	MM	%
CD VS L6			L7 RA L3, SA, SAR			CS B&I, CM/DM RES MGT		
TOTAL			TOTAL			TOTAL		

TOTAL RELEASE = ______

BUFFER IN VS ______ BUFFER IN L6 ______ BUFFER CUTOFF ______

BUFFER MANAGEMENT

DATE	CR'S	COST	BUFFER	DATE	CR'S	COST	BUFFER	DATE	CR'S	COST	BUFFER

CR'S ABSORBED ______ BUFFER USED ______ BUFFER REMAINING ______

ACTUALS

ACTUALS DATE ______ CR BASELINE DATE ______ DR BASELINE DATE ______

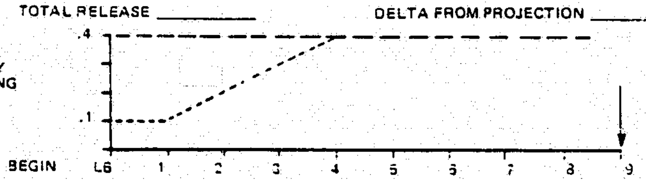
NO. OF CR'S ______ DIRECT CR EST. ______ NO. OF DR'S ______

MANPOWER ACTUALS

FUNCTION	MM	%	FUNCTION	MM	%	FUNCTION	MM	%
CD VS L6			L7 RA L3, SA, SAR			CS B&I, CM/DM RES MGT		
TOTAL			TOTAL			TOTAL		

TOTAL RELEASE ______ DELTA FROM PROJECTION ______

QUALITY TRACKING



K. Rone
IBM

D/12

N84 23149

An Approach To Software Baseline Generation

By:

Jorge Luis Romeu
IIT Research Institute
199 Liberty Plaza
Rome, New York 13440

1.0 INTRODUCTION

This paper summarizes a current Data & Analysis Center for Software (DACS) effort to develop software baselines. This baseline effort is an on-going activity; that is, the baselines are meant to be updated as new software data becomes available. The information presented and processed has been organized to make periodic updating a much simpler task.

A baseline, for this effort, will consist of an estimation of any characteristic of a software project that may be helpful to a developer, manager, or monitor to manage, control, or influence a software product. The objective of these baselines is to provide a tool for aiding software developers in their daily work. Baselines have been synthesized from an empirical dataset provided by the Software Engineering Laboratory at NASA Goddard Space Flight Center (NASA/SEL). These data have been selected because the data collection effort developed at the NASA/SEL is the most thorough and complete available to us. The characteristics of the NASA/SEL environment may not be common to most or all users. Therefore, the user is advised to calibrate our baselines with his professional judgement and experience to provide for the possible differences between his and the NASA/SEL environments.

The baseline effort, defined as an on-going activity, has been broken down into several phases. The motive for the division of the baseline effort into successive phases is two fold. The first motive is the desire to provide the practitioner with the most current information. Waiting until all variables have been analyzed to release the package incurs the risk of providing very outdated baselines. Second, and more important for the future development of this effort, the plan for producing the baselines may be subject to modifications. The practitioner may require different/additional information or the same information presented in another form. Therefore, any comments and suggestions to adapt, modify or change the present baselines in order to improve their practical use is not only welcome but is considered to be an integral part of this effort.

2.0 METHODOLOGY

The importance of breaking down software projects into smaller and more homogeneous subgroups was an insight gained from previous analysis tasks. Project heterogeneity was caused by the presence of very different factors which were not possible to isolate in different software projects. A solution was provided by breaking down the set of software projects into more homogeneous subgroups.

We have selected the current version (February 1983) of the NASA/SEL dataset as the empirical base from which to develop software baselines. This set contains the latest version of the comprehensive and thorough data collection effort performed by the NASA/SEL staff. It exhibits two interesting features. First, the data was collected at both the project and module or component levels. Second, these components are classified according to their function (See Table I). This classification will allow us to characterize each component by the function it performs within the project. These module functions also provide the scheme for breaking the data into homogeneous subgroups.

TABLE I: DESCRIPTION OF THE MODULE/COMPONENT FUNCTIONS ANALYZED DURING THE PRESENT EFFORT

NASA/SEL Encoding Dictionary

Code	Module/Component Name	Function
1	Include	Include Statements
2	Control	Control Statements (JCL, Overlay)
3	System	System Statements (ALC)
4	Gess	Graphics Statements (GESS)
5	Data	Data Statements
7	CDR	FORTRAN Control/Driver Module
8	C COMP	FORTRAN Control/Computational Statements
9	DTRANS	FORTRAN Data Transfer Module
10	IO	FORTRAN Input/Output Module
17	IOCDR	FORTRAN Control/Driver Module with I/O
18	IOCCOMP	FORTRAN Control/Computational Module W
19	IODTRANS	FORTRAN Data Transfer Module with I/O

It is useful to perform a simultaneous analysis as a triple function of data analysis (baseline generation), data quality assurance, and research where a large dataset such as the NASA/SEL is being studied. The first, baseline generation, is the primary objective of this effort. The baselines are designed to statistically address the following question:

> What does a module or component of a given function look like? In other words, how can I describe a "typical" module that performs a given function in terms of its size, effort, runs necessary to develop this module, origin, complexity and type of specification? How does this situation vary, if any, from the moment this module is given to a programmer until the moment this module is ready (i.e., from the NEW to the COMPLETED stage)?

The second, quality assurance, is inherent to statistical analysis. A statistician carefully logs in his quality assurance notebook all observed inconsistencies during the process of data reduction and analysis. The analysis of these isolated inconsistencies provide insight to the process being studied and/or improvements to the data collection process. These insights often prove useful for both the data collector and the analyst in future efforts. Finally, the research function in the baseline generation follows a sequence of activities:

(1) look at large numbers of software components or modules grouped by common function to try to isolate the similarities and differences stemming from this grouping

(2) try to determine, given that we are dealing with empirical data, whether these similar and different behavioral patterns are arising by chance

(3) if (2) is not true, to determine if there is sufficient statistical proof to state that these patterns are an inherent characteristic of these groupings of modules/components

This type of information is useful to both the theoretical software engineering researcher and active practitioner, the software developer. It may be possible in future efforts to uncover a correlation that enables the practitioner to obtain one element (module/component) from another or to monitor one element while fixing the other, once it can be established that a relationship exists between two integral elements of the dataset.

In addition, certain relationships are known to hold from theory or experience. If it is found that they also hold in the data, it provides an indicator of the quality of the information; if they don't, it may either mean that the quality of the data is suspect or that there may be some special characteristics about this situation that deserve further investigation. This indicator becomes a useful working tool. This is the framework in which the present research effort has been conducted. Baseline results should follow this line of thought.

We have worked exclusively with the variables shown in Table II in the current phase of the baselines task.

TABLE II: DATA DEFINITION FOR VARIABLES USED IN PHASE I

	Variable	Format
1.	Project - code	I(2)
2.	Component - code	I(3)
3.	Module - function	I(2)
4.	System/subsystem	I(2)
5.	Origin-	I(2)
6.	Precision of specifications	I(2)
7.	Complexity	A(2)
8.	Num-comp-called	I(2)
9.	Num-calling-components	I(2)
10.	Primary-language	I(2)
11.	% of Primary	I(3)
12.	Secondary-language	I(2)
13.	% of Secondary	I(3)
14.	Total - runs	I(4)
15.	Total - time	I(4)
16.	Total - effort	I(4)
17.	Total-source-for-components	I(8)
18.	Development status	A(2)

There were several reasons for selecting from all of the possible variables existing in the NASA/SEL dataset these 18 variables. The main thrust of the current phase of the baselines task is the characterization of project components by some type of useful grouping. Project code combined with component code (1 and 2) provides identification for each module/component, and the module function (3) provides the required subgrouping to produce more homogeneous subsets. The variable System/subsystem (4) was not used in this phase of the baselines task. The qualitative variables origin (5), precision of the specifications (6), and perceived complexity (7), were selected to assess whether the subjective appraisal of a module/component by a programmer primarily reflects the module's characteristics or rather the programmer's characteristics. The number of components called (8) and the number of calling components (9) were selected as an indicator of the module complexity with respect to its interface within the whole system structure.

Data on the variables (8) and (9) was not available throughout the entire twelve module functions defined in Table I in quantities large enough to support analysis. These variables, therefore, were analyzed only in the last four and most numerous, module function groups.

It was observed that each component was written in a single language and that only two languages, FORTRAN and Assembler, were utilized throughout the dataset. The variables total runs (14), total time (15), total effort (16) and total source for components (17) refer to computer runs, time spent by a programmer in computer work, programming effort and module size respectively. These last 4 variables will provide quantitative characterization baselines in subsequent phases of this effort.

The variable development status (18) provides two qualitative classes: New and Completed. This variable provided a key analysis tool since it is possible to compare the state of the module, represented by all of the above variables, before and after its implementation. This type of analysis will yield a valuable management tool since it will allow assessing the accuracy of the estimates provided by the programmers at the beginning of their tasks.

Preliminary results are presented in Tables IIIA, IIIB, IVA, and IVB as examples of the type of output obtained from these analyses by module functions.

TABLE IIIA: CORRELATION SIZES/SIGNIFICANCES

NEW (PRE-DEVELOPMENT ESTIMATES)

MODULE FUNCTION	SIZE VS EFFORT	SIZE VS TIME	SIZE VS RUNS	EFFORT VS TIME	EFFORT VS RUNS	TIME VS RUNS
Include	26/***	25/*	25/NS	25/**	25/**	25/**
Control	N/A	N/A	N/A	N/A	N/A	N/A
System	10/NS	9/NS	8/NS	9/*	8/NS	7/NS
Gess	85/***	84/***	84/***	88/***	88/***	88/***
Data	135/***	108/***	114/***	112/***	118/***	112/***
CDR	48/***	46/***	46/***	47/***	47/**	46/**
C COMP	28/**	22/**	26/NS	24/***	28/**	24/***
DTRANS	53/**	40/NS	43/**	42/***	45/*	42/**
IO	129/***	110/***	119/***	111/***	121/***	111/***
IOCDR	255/***	206/***	212/***	209/***	215/***	208/***
IOCCOMP	189/***	160/***	163/***	163/***	166/***	163/***
IODTRANS	128/***	101/***	103/***	105/***	107/***	105/***

Legend:
- N/A sufficient data not available to support test
- NS non-significant
- * significant at level $\alpha = 0.05$
- ** significant at level $\alpha = 0.01$
- *** significant at level $\alpha = 0.001$

Footnote to Table IIIA

This table provides in each cell the number of pairs used in the estimation of the τ correlation coefficient and the significance level attained by this coefficient. For example, "26/***" in the first cell, indicates that the τ correlation coefficient was computed for 26 "Include" modules and results were that effort and size were correlated at the .001 level of significance. The table is useful for

i) directing the analyst in successive phases of this effort

ii) evaluating the quality of the data

iii) proposing new and assessing old research questions

TABLE IIIB: CORRELATION SIZES/SIGNIFICANCES

COMPLETE (ACTUAL)

MODULE FUNCTION	EFFORT	SIZE VS TIME	RUNS	EFFORT VS TIME	RUNS	TIME VS RUNS
Include	25/NS	25/*	25/**	25/***	25/***	25/***
Control	N/A	N/A	N/A	N/A	N/A	N/A
System	7/NS	N/A	N/A	N/A	N/A	N/A
Gess	73/***	31/**	32/**	31/***	32/***	31/***
Data	142/***	86/***	87/***	86/**	87/***	86/***
CDR	51/***	46/*	46/**	46/***	46/***	46/***
C COMP	36/***	27/NS	27/*	27/*	27/**	27/***
DTRANS	60/NS	39/NS	39/**	39/***	39/***	39/***
IO	119/***	84/***	86/***	84/***	86/***	84/***
IOCDR	254/***	197/***	199/***	198/***	200/***	201/***
IOCCOMP	180/***	135/***	137/***	136/***	138/***	136/***
IODTRANS	124/**	89/***	92/**	89/***	92/***	89/***

Legend:
N/A sufficient data not available to support test
NS non-significant
* significant at level $\alpha = 0.05$
** significant at level $\alpha = 0.01$
*** significant at level $\alpha = 0.001$

Footnote to Table IIIB

This table provides in each cell the number of pairs used in the estimation of the τ correlation coefficient and the significance level attained by this coefficient. The interpretation of this table is the same as for Table IIIA. The table is useful for

i) directing the analyst in successive phases of this effort

ii) evaluating the quality of the data

iii) proposing new and assessing old research questions

TABLE IVA: CONTINGENCY TABLE RESULTS

COMPLEXITY VS CODE ORIGIN

MODULE FUNCTION	NEW	COMPLETE
Include	N/A	N/A
Control	N/A	N/A
System	**	NS
Data	NS	NS
CDR	NS	NS
C COMP	NS	NS
DTRANS	NS	NS
IO	***	NS
IOCDR	NS	*
IOCCOMP	NS	NS
IODTRANS	NS	NS

Legend:
- N/A sufficient data not available to support test
- NS non significant at level $\alpha = 0.05$
- * significant at level $\alpha = 0.05$
- ** significant at level $\alpha = 0.01$
- *** significant at level $\alpha = 0.001$

Footnote to Table IVA

The degree of association between the two qualitative variables complexity and code origin was established through contingency tables at the NEW and COMPLETED development phases. The results are tabulated here and may provide

i) an evaluation of the quality of the data

ii) new research questions

TABLE IVB: CONTINGENCY TABLE RESULTS

COMPLEXITY VS PRECISION OF SPECIFICATION

MODULE FUNCTION	NEW	COMPLETE
Include	N/A	N/A
Control	N/A	N/A
System	*	***
Data	***	***
CDR	NS	NS
C COMP	*	NS
DTRANS	**	NS
IO	***	***
IOCDR	**	***
IOCCOMP	NS	*
IODTRANS	NS	***

Legend:

N/A	sufficient data not available to support test
NS	non-significant at level $\alpha = 0.05$
*	significant at level $\alpha = 0.05$
**	significant at level $\alpha = 0.01$
***	significant at level $\alpha = 0.001$

Footnote to Table IVB

The degree of association between the two qualitative variables complexity and code origin was established through contingency tables at the NEW and COMPLETED development phases. The results are tabulated here and may provide:

i) an evaluation of the quality of the data

ii) new research questions

CONCLUSIONS

The baselines effort is an on-going activity. It has barely started and some elementary baselines will be generated to include a subset of the variables that characterize a module from a functional perspective. The next phase in this effort will contemplate completing this characterization process by looking at other variables and exploiting some of the functional relations that have been explored and established during the present phase.

It will become necessary to begin a study of the performance measures of these same modules after the characterization process is sufficiently explored. This activity will include the study of productivity and the process of changes (both error correction and enhancements). Eventually, this will lead to the study of different methodologies and other production factors and their influence on the behavior of the above-mentioned performance measures and other measures suggested by this research.

ACKNOWLEDGEMENT

The author thanks Frank McGarry (NASA/SEL) for furnishing the data to perform the present analyses and Rocco Iuorno (IITRI) and John Palaimo (RADC/COEE) for their help in defining the Software Engineering baseline needs.

Bibliography

A - Software Engineering

1. Thayer et al., Software Reliability Study, RADC-TR-76-238, August 1976.

2. Evaluation of Management Measures of Software Development, Vols. 1 & 2, SEL-82-001, September 1982.

3. Dekker, G.J., Wilt and F.J. Bosch, Functional Requirements for a Software Cost Database, National Aerospace Laboratory, The Netherlands, 1982.

4. Wallston, C.E., and C.P. Felix, "A Method of Programming Measurement and Estimation," IBM Journal, 16(1), 1977, pp 54-73.

5. Byrne, W.E., A Military Standard for Software related Technical Data, MITRE Corporation, MTR-8556, January 1982.

6. Basili, V., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235.

7. McCall, J.A., and M.T. Matsumoto, Software Quality Metrics, September 1979. Final Report for the Metrics Enchancement Study, contract No. F30602-78-C-0216.

8. McCall, J., et. al., Factors in Software Quality, RADC-TR-77-369, Vols. I, II, and III, November 1977.

9. Iuorno, R., et al., "The DACS Software Engineering Data Collection Package", DACS draft report, June 1983.

B - Statistics

BOX73 Box, G.E.P. and G.C. Tiao, Bayesian Inference in Statistical Analyses, Addison Wesley, 1973, Chapter 3: "Effects of Non-Normality on inferences about a population mean with Generalization."

KEND70 Kendall, M.G., Rank Correlation Methods, Griffin, London 1970 (4th edition).

LEHM75 Lehman, E.L., Non-Parametrics Statistical Methods Based on Ranks, Holden Dey, 1975.

NEL79 Nelson, Richard, "Software Data Collection and Analysis," Draft Report, DACS, 1979.

ROHA76 Rohatgi, V.K., An Introduction to Probability Theory and Mathematical Statistics, Wiley, 1976.

ROME82 Romeu, J.L., and C. Turner, "Parametric vs Non-Parametric Techniques in the Analysis of Software Productivity Data," Draft DACS Report, December 1982.

ROME83b Romeu, J.L., and S.A. Gloss-Soler, "Some Measurement Problems Detected in the Analysis of Software Productivity Data and Their Statistical Consequences," Proceedings of COMPSAC '83.

SEN71 Puri, M.L., and P.K. Sen, Non-Parametric Methods in Multivariate Analysis, Wiley, 1971.

SIEG56 Siegel, S., Non-Parametric Statistics for the Behavioral Sciences, McGraw-Hill, 1956.

DACS

DATA & ANALYSIS CENTER FOR SOFTWARE

Rome Air Development Center

IIT RESEARCH INSTITUTE

ORIGINAL PAGE IS
OF POOR QUALITY

J. Romeu
IITRI
14 of 27

AN APPROACH TO SOFTWARE BASELINE GENERATION

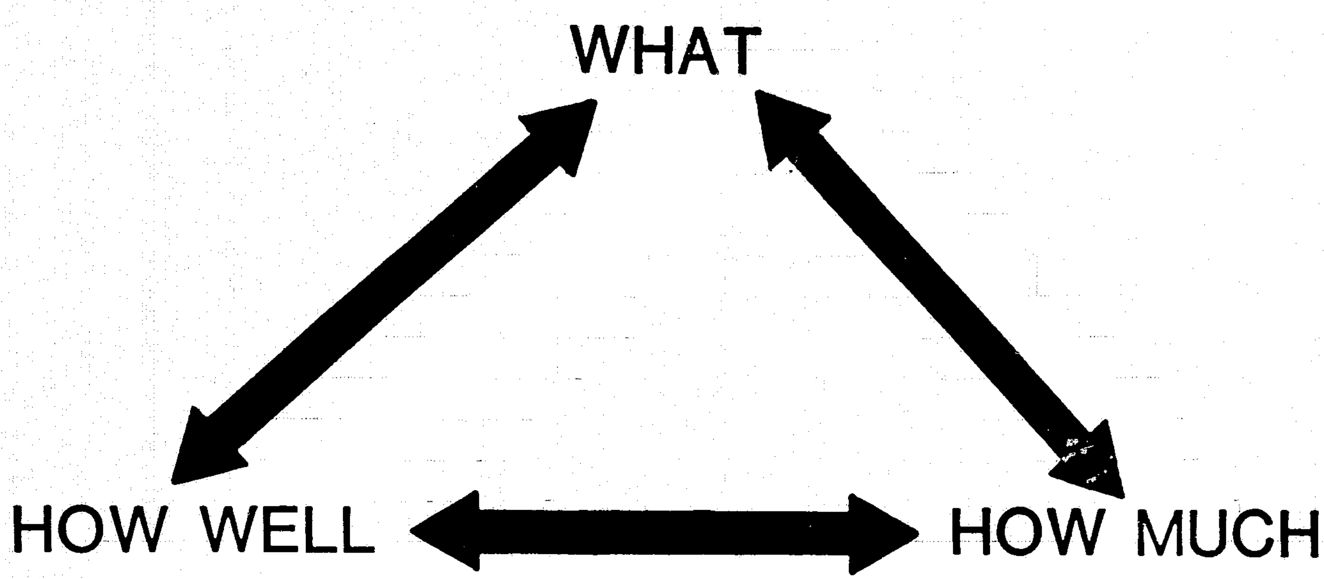
JORGE LUIS ROMEU

IIT Research Institute
199 Liberty Plaza
Rome, New York 13440

BASELINE

An estimation of any characteristic of a software project that may be helpful to the developer, manager or monitor to manage, control or exert any conscious influence over a software project.

SOFTWARE PROJECT



FRAMEWORK

1) DISAGREGATE

2) CHARACTERIZE

3) PATTERN SEARCH

4) STATISTICAL TESTS

5) INFERENCES

Description of the Module/Component Functions Analyzed During the Present Effort

NASA/SEL Encoding Dictionary

Code	Module/Component Name	Function
1	Include	Include Statements
2	Control	Control Statements (JCL, Overlay)
3	System	System Statements (ALC)
4	Gess	Graphics Statements (GESS)
5	Data	Data Statements
7	CDR	FORTRAN Control/Driver Module
8	C COMP	FORTRAN Control/Computational Statements
9	DTRANS	FORTRAN Data Transfer Module
10	IO	FORTRAN Input/Output Module
17	IOCDR	FORTRAN Control/Driver Module with I/O
18	IOCCOMP	FORTRAN Control/Computational Module W
19	IODTRANS	FORTRAN Data Transfer Module with I/O

REASONS FOR VARIABLE SELECTION

1) IDENTIFICATION

2) CLASSIFICATION

3) CHARACTERIZATION

4) SUBJECTIVE EVALUATION

5) TIME MILESTONES

DATA DEFINITION

VARIABLE

1. PROJECT - CODE
2. COMPONENT - CODE
3. MODULE - FUNCTION
4. SYSTEM/SUBSYSTEM
5. ORIGIN
6. PRECISION OF SPEC
7. COMPLEXITY
8. NUM-COMP-CALLED
9. NUM-CALLING-COMP
10. PRIMARY-LANG
11. % OF PRIMARY
12. SECONDARY-LANG
13. % OF SECONDARY
14. TOTAL - RUNS
15. TOTAL - TIME
16. TOTAL - EFFORT
17. TOTAL-SOURCE-FOR-COMP
18. DEVELOPMENT STATUS

CORRELATION OBJECTIVES

1) Directing the analyst in successive phases

2) Evaluating the quality of the data

3) Exploring new and assessing old research questions

Correlation Sizes/Significances

New

Module Function	Effort	Size vs Time	Size vs Runs	Effort vs Time	Effort vs Runs	Time vs Runs
Include	26/***	25/*	25/NS	25/**	25/**	25/**
Control	N/A	N/A	N/A	N/A	N/A	N/A
System	10/NS	9/NS	8/NS	9/*	8/NS	7/NS
Gess	85/***	84/***	84/***	88/***	88/***	88/***
Data	135/***	108/***	114/***	112/***	118/***	112/***
CDR	48/***	46/***	46/***	47/***	47/**	46/**
C COMP	28/**	22/**	26/NS	24/***	28/**	24/***
DTRANS	53/**	40/NS	43/**	42/***	45/*	42/**
IO	129/***	110/***	119/***	111/***	121/***	111/***
IOCDR	255/***	206/***	212/***	209/***	215/***	208/***
IOCCOMP	189/***	160/***	163/***	163/***	166/***	163/***
IODTRANS	128/***	101/***	103/***	105/***	107/***	105/***

Correlation Sizes/Significances

Complete

Module Function	Effort	Size vs Time	Size vs Runs	Effort vs Time	Effort vs Runs	Time vs Runs
Include	25/NS	25/*	25/**	25/***	25/***	25/***
Control	N/A	N/A	N/A	N/A	N/A	N/A
System	7/NS	N/A	N/A	N/A	N/A	N/A
Gess	73/***	31/**	32/**	31/***	32/***	31/***
Data	142/***	86/***	87/***	86/**	87/***	86/***
CDR	51/***	46/*	46/**	46/***	46/***	46/***
C COMP	36/***	27/NS	27/*	27/*	27/**	27/***
DTRANS	60/NS	39/NS	39/**	39/***	39/***	39/***
IO	119/***	84/***	86/***	84/***	86/***	84/***
IOCDR	254/***	197/***	199/***	198/***	200/***	201/***
IOCCOMP	180/***	135/***	137/***	136/***	138/***	136/***
IODTRANS	124/**	89/***	92/**	89/***	92/***	89/***

Complexity vs Code Origin

Module Function	New	Complete
Include	N/A	N/A
Control	N/A	N/A
System	**	NS
Data	NS	NS
CDR	NS	NS
C COMP	NS	NS
DTRANS	NS	NS
IO	***	NS
IOCDR	NS	*
IOCCOMP	NS	NS
IODTRANS	NS	NS

COMPLEXITY VS. PRECISION OF SPEC

MODULE FUNCTION	NEW	COMPLETE
INCLUDE	N/A	N/A
CONTROL	N/A	N/A
SYSTEM	**	NS
DATA	NS	NS
CDR	NS	NS
C COMP	NS	NS
DTRANS	NS	NS
IO	***	NS
IOCDR	NS	*
IOCCOMP	NS	NS
IODTRANS	NS	NS

CONCLUSIONS

1) BASELINE METHODOLOGY

2) INITIAL BASELINES

3) FURTHER WORK/RESEARCH

ORIGINAL PAGE IS
OF POOR QUALITY

PEOPLE WHO ATTENDED THE WORKSHOP ON NOVEMBER 30, 1983

NAME	ORGANIZATION
ADELSON,B	YALE UNIVERSITY
AGRESTI,B	C.S.C.
AKERS,F	NASA/GSFC
ALBRIGO,F	C.S.C.
ARDANUY,P	RESEARCH & DATA SYSTEMS,INC
ASTILL,P	SIGMA DATA
AYERS,E	ARINC RESEARCH CORPORATION
BABST,T	C.S.C.
BARKSDALE,J	NASA/GSFC
BARRETT,C	NASA/GSFC
BASILI,V	UNIVERSITY OF MARYLAND
BELL,J	PEACE CORPS
BIGWOOD,D	U.S. DEPT. OF AGRICULTURE
BLAND,R	U.S. DEPT. OF AGRICULTURE
BOEHM-DAVIS,D	GENERAL ELECTRIC
BOGARD,L	JHU-APL
BOND,R	N.S.A.
BONK,H	NASA/GSFC
BRACKEN,P	NASA/GSFC
BREDESON,R	CAD CORPORATION
BREDESON,M	I.I.T.R.I.
BROOKS,G	E.G. & G.
BROWN,M	FED HOME LOAN MORTGAGE CORP
BRUBAKER,M	INTELLIMAC INC
BUTE,E	C.S.C.
BURNS,T	MITRE CORPORATION
BURTON,J	U.S. DEPT. OF AGRICULTURE
CALLAHAN,J	NASA/GSFC
CAMPBELL,J	E.P.A.
CARO,D	C.S.C.
CARPENTER,L	NASA/GSFC
CERMAS,A	NASA/GSFC
CHERRIT,C	NASA/GSFC
CHEUVRONT,S	C.S.C.
CHURCH,V	C.S.C.
CUBS,S	INCO,INC
CONNELLY,E	PERFORMANCE MEASUREMENTS ASSOC
COOK,J	NASA/GSFC
COOK,L	GENERAL SOFTWARE CORPORATION
COOPER,R	U.S. DEPT. OF AGRICULTURE
COPPERTHITE,R	ACTION
CRAWFORD,S	BELL LABS
CRUICKSHANK,R	IBM CORPORATION
CURRIT,P	IBM CORPORATION
CURTIS,R	U.S. DEPT. OF AGRICULTURE

DECKER,N	C.S.C.
DICKSON,C	U.S. DEPT. OF AGRICULTURE
DIECKHANS,J	F.C.C.
DISKIN,D	U.S.BUREAU OF THE CENSUS
DOUGLAS,F	PROFESSIONAL SOFTWARE SERVICES
DOZIER,J	ARINC RESEARCH CORPORATION
DUNHAM,J	RESEARCH TRIANGLE INSTITUTE
DUNIHO,M	N.S.A.
EDWARDS,F	NASA/GSFC
ELIAS,D	G.T.E.
ENG,E	NASA/GSFC
ESTARIA,P	SPACE TELESCOPE SCIENCE
FAGAN,M	IBM CORPORATION
FETIG,A	NASA/GSFC
FRAZIER,P	BENDIX
FRENKEL,Y	C.S.C.
FREUND,A	BURROUGHS CORPORATION
FRIEDMAN,D	RAM CORPORATION
GAFFNEY,J	IBM CORPORATION
GANNETT,M	N.S.A.
GARDINER,B	DEPARTMENT OF AGRICULTURE
GARRICK,J	NASA/GSFC
GARY,P	NASA/GSFC
GIDDINGS,V	MITRE CORPORATION
GILBERT,J	U.S.ARMY OPER TEST & EVAL CTR
GILL,K	C.S.C.
GLICK,N	N.S.A.
GOEL,A	DEPT OF IND ENG & OPER RESEARCH
GOODSON,A	NASA/GSFC
GRAFTON,R	OFFICE OF NAVAL RESEARCH
GRAY,L	INTELLIMAC INC
GREEN,A	C.S.C.
GREEN,S	NASA/GSFC
GREENLAND,A	IIT RESEARCH INSTITUTE
GRIEF,S	BENDIX
GUION,W	NASA/GSFC
HAMHA,M	INTERNAL REVENUE SERVICE
HILLMER,D	U.S.BUREAU OF THE CENSUS
HOLLOWAY,R	NASA LANGLEY RESEARCH CENTER
HOLMES,B	GENERAL SOFTWARE CORPORATION
HOLMES,R	NASA/GSFC
HOUSER,A	GSA/ORIM/KTA
HULL,L	NASA/GSFC
HWANG,P	NASA/GSFC
IDELSON,N	I.I.T.R.I.
JAMIESON,L	NASA/GSFC
JAWORSKI,R	BURROUGHS CORP.
JOHNSON,T	SIGMA DATA
JOHNSON,A	OFFICE OF SOFTWARE DEVELOPMENT
JORDAN,L	C.S.C.

KATZ,E	UNIVERSITY OF MARYLAND
KATZ,[illegible]	DEFENSE INTELLIGENCE AGENCY
KATZ,S	DEFENSE INTELLIGENCE AGENCY
KAZLAUSKI,F	NAVDAK
KELLY,K	JET PROPULSION LAB
KIDD,L	NASA/GSFC
KINGSTON,[illegible]	U.S. DEPT. OF AGRICULTURE
KLEIK,J	MITRE CORPORATION
KNOX,R	NASA/GSFC
KOO,D	WESTINGHOUSE INC.
KRAMER,N	PRC/GIS
KUHN,J	N.S.A.
KURIHARA,T	U.S.DEPT.OF TRANSPORTATION
LATOS,S	NASA/GSFC
LEE,C	DOTY ASSOCIATES
LEVINE,D	INTERMETRICS, INC
LIU,K	C.S.C.
LU,P	C.S.C.
LOMAS,J	FEDERAL CONVERSION SUPPORT CTR
LYTTON,H	U.S. DEPT. OF AGRICULTURE
MACDONALD,M	INTERNAL REVENUE SERVICE
MACK,H	NASA/GSFC
MARKS,D	GRIFFISS AFB
MAROOF,N	NASA/GSFC
MASON, D.	BURROUGHS CORPORATION
MANTEUFEL, T.	NASA/GSFC
MCCOY,W	NAVAL SURFACE WEAPONS CENTER
MCGARRY,M	IITRI
MCGARRY,F	NASA/GSFC
MCGARRY,P	GENERAL ELECTRIC
MCHUGH,J	RESEARCH TRIANGLE INSTITUTE
MCKENNA,J	N.S.A.
MCKENZIE,M	JET PROPULSION LAB
MCPHEE,J	BUREAU OF INDUSTRIAL ECONOMICS
MEESON,R	COMPUTER TECHNOLOGY ASSOC
MENDENHALL,V	NAVAL SURFACE WEAPONS CTR
MERWARTH,P	NASA/GSFC
MILES,T	BUREAU OF INDUSTRIAL ECONOMICS
MILLER,F	PRC SYSTEMS SERVICES
MILLER,W	C.S.C.
MILLIGAN,W	FEDERAL SOFTWARE TESTING CTR
MILLNER,D	IIT RESEARCH INSTITUTE
MOWBAY,B	GENERAL DYNAMICS
MURPHY,B	NASA/GSFC
MYERS,P	C.S.C.
NADOLNA,D	N.S.A.
NELSON,R	NASA/GSFC
NEUMAN,J	NATIONAL BUREAU OF STANDARDS
NG,E	JET PROPULSION LABS
NORMAN,K	UNIVERSITY OF MARYLAND
O'KEEFE,S	U.S. CUSTOM SERVICE
O'NEIL,L	BELL LABS

OBRISKAT,K	U.S. DEPT. OF AGRICULTURE
[illegible],P	NASA/GSFC
PACKARD,C	NASA/GSFC
PAGE,J	C.S.C.
PANITGTU-YAP,[illegible]	UNIVERSITY OF MARYLAND
PARCOVER,G	DEPT. OF HOUSING & URBAN DEV.
PARKER,D	NASA/GSFC
PARKER,J	N.S.A.
PASSALACQUA,T	CENSUS BUREAU
PENNY,L	PENNY ASSOCIATES
PFEIFER,D	NASA/GSFC
PIETRAS,J	MITRE CORPORATION
PLEIT,[illegible]	C.S.C.
POSTAK,J	DOTY ASSOCIATES
POWELL,J	DEPT OF LABOR
QUANN,J	NASA/GSFC
RAMSEY,J	UNIVERSITY OF MARYLAND
RATCLIFFE,V	INTERNAL REVENUE SERVICE
RATIE,C	U.S. DEPT. OF AGRICULTURE
REDDY,R	NASA/GSFC
REDAINE,S	MITRE CORPORATION
RHEE,H	NASA/GSFC
RIDDLE,T	NASA/GSFC
RIZZARDI,G	AFDSC/IRS PENTAGON
ROBASKY,[illegible]	GENERAL ELECTRIC
ROBBINGS,D	N.S.A.
ROBINSON,J	N.S.A.
ROHLEDER,[illegible]	WESTINGHOUSE ELECTRIC CORP
ROMAN,I	IIT RESEARCH INSTITUTE
ROME,K	TRW CORPORATION
ROSE,S	GTE SYSTEMS
ROSENTHAL,D	SPACE TELESCOPE SCIENCE
RUSSIN,R	GENERAL ELECTRIC
ROSTAIN,A	YALE UNIVERSITY
RUPERT,F	FED HOME LOAN MORTGAGE CORP
RUPOLO,V	BANKERS TRUST COMPANY
RYKOWSKI,T	NASA/GSFC
SANBORN,I	NASA/GSFC
SAVOLAINE,C	BELL LABS
SCHNEIDERMAN,B	UNIVERSITY OF MARYLAND
SCHOONMAKER,P	MCDONNELL DOUGLAS
SCHULTZ,D	C.S.D.
SCOTT,L	N.S.A.
SELAY,P	UNIVERSITY OF MARYLAND
SHAMES,P	SPACE TELESCOPE SCIENCE INSTIT
SHEPPARD,S	C.S.C.
SIMS,L	NAVAL SURFACE WEAPONS CENTER
SINGH,D	ARMY COMPUTER SYSTEMS COMMAND
SMITH,C	NASA/GSFC
SMITH,J	NAVAL SURFACE WEAPONS CTR
SMITH,O	E.G. & G.
SMITH,R	NAVAL SURFACE WEAPONS CTR

ORIGINAL PAGE IS
OF POOR QUALITY

SNYDER,G	C.S.C.
SOLOMON,D	C.S.C.
SOLOWAY,E	YALE UNIVERSITY
SOPKOWITZ,A	DEPT. OF HUD
SPRAVREGEN,L	SIGMA DATA
STACY,G	FORD AEROSPACE
STANTON,M	U.S. DEPT. OF AGRICULTURE
STARK,M	NASA/GSFC
STEPHENS,[illegible]	NASA/GSFC
STONE,P	PRC SYSTEMS SERVICES
SUDDITH,S	GENERAL SOFTWARE CORPORATION
SUKRI,J	UNIVERSITY OF MARYLAND
SZULEWSKI,P	CHARLES STARK DRAPER LAB
TASAKI,K	NASA/GSFC
THOMPSON,R	NASA/GSFC
TIDD,R	N.S.A.
TOM,K	ARINC RESEARCH CORP
TOTH,G	NASA/GSFC
TRUSS,V	IIT RESEARCH INSTITUTE
TURNER,R	NAVDAC
TURNROSE,B	C.S.C.
USHER,G	U.S. DEPT. OF AGRICULTURE
VALARDO,J	RESEARCH & DATA SYSTEMS INC
VALOIS,P	I.S.S.I.
VALETT,J	NASA/GSFC
VANOVITCH,G	ARINC RESEARCH CORP
WALIGORA,S	C.S.C.
WALLACE,D	NATIONAL BUREAU OF STANDARDS
WATSON,B	NASA/GSFC
WATSON,R	I.I.T. RESEARCH INSTITUTE
WEATHERBEE,J	USAF/ESDSIM
WEISS,D	COMSAT
WILDES,W	NASA/GSFC
WILLIAMSON,D	IIT RESEARCH
WILLIS,R	INTELLIMAC INC
WOLBER,R	SPERRY CORPORATION
WRIGHT,F	C.S.C.
WYNN,V	NASA/GSFC
YANT,[illegible]	NATIONAL SECURITY AGENCY
YORK,E	AFDSC/IRS
YOUMAN,C	MITRE CORPORATION
YOUNG,J	SIGMA DATA
ZELKOWITZ,M	UNIVERSITY OF MARYLAND
ZYGIELBAUM, A.	JET PROPULSION LAB

TOTAL = 243

BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu and D. S. Wilson, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

[1]SEL-78-002, FORTRAN Static Source Code Analyzer (SAP) User's Guide, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-102, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), W. J. Decker and W. A. Taylor, September 1982

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

[1]SEL-80-004, System Description and User's Guide for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, W. J. Decker, J. G. Garrahan, et al., October 1980

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

[1]SEL-81-001, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

[1]SEL-81-002, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, G. Page, and F. E. McGarry, September 1981

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

[1]SEL-81-003, Data Base Maintenance System (DBAM) User's Guide and System Description, D. N. Card, D. C. Wyckoff, and G. Page, September 1981

SEL-81-103, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo and D. Card, July 1983

[1]SEL-81-004, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., September 1981

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

[1]SEL-81-005, Standard Approach to Software Development, V. E. Church, F. E. McGarry, G. Page, et al., September 1981

[1]SEL-81-105, Recommended Approach to Software Development, S. Eslinger, F. E. McGarry, and G. Page, May 1982

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-81-006, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, December 1981

[1]SEL-81-007, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, E. J. Smith, A. L. Green, et al., February 1981

SEL-81-107, *Software Engineering Laboratory (SEL) Compendium of Tools*, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-008, *Cost and Reliability Estimation Models (CAREM) User's Guide*, J. F. Cook and E. Edwards, February 1981

SEL-81-009, *Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation*, W. J. Decker and F. E. McGarry, March 1981

[1]SEL-81-010, *Performance and Evaluation of an Independent Software Verification and Integration Process*, G. Page and F. E. McGarry, May 1981

SEL-81-110, *Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics*, G. Page and F. McGarry, December 1983

SEL-81-011, *Evaluating Software Development by Analysis of Change Data*, D. M. Weiss, November 1981

SEL-81-012, *The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems*, G. O. Picasso, December 1981

SEL-81-013, *Proceedings From the Sixth Annual Software Engineering Workshop*, December 1981

SEL-81-014, *Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL)*, A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-82-001, *Evaluation of Management Measures of Software Development*, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-002, *FORTRAN Static Source Code Analyzer Program (SAP) System Description*, W. A. Taylor and W. J. Decker, August 1982

SEL-82-003, *Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description*, P. Lo, September 1982

SEL-82-004, *Collected Software Engineering Papers: Volume 1*, July 1982

[1]SEL-82-005, *Glossary of Software Engineering Laboratory Terms*, M. G. Rohleder, December 1982

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

[1]SEL-82-006, Annotated Bibliography of Software Engineering Laboratory (SEL) Literature, D. N. Card, November 1982

SEL-82-106, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, T. A. Babst, and F. E. McGarry, November 1983

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-83-001, Software Cost Estimation Experiences, F. E. McGarry, G. Page, D. N. Card, et al., November 1983

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., November 1983

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-004, SEL Data Base Retrieval System (DARES) User's Guide, T. A. Babst and W. J. Decker, November 1983

SEL-83-005, SEL Data Base Retrieval System (DARES) System Description, P. Lo and W. J. Decker, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-RELATED LITERATURE

[2]Agresti, W. W. , F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[3]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

Banks, F. K., "Configuration Analysis Tool (CAT) Design," Computer Sciences Corporation, Technical Memorandum, March 1980

[3]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1979

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: Computer Societies Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[2]Basili, V. R., and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, University of Maryland, Technical Report TR-1195, August 1982

[3]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

[2]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[3]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[3]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: Computer Societies Press, 1978

[3]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

Card, D. N., and V. E. Church, "Analysis Software Requirements for the Data Retrieval System," Computer Sciences Corporation Technical Memorandum, March 1983

Card, D. N., and V. E. Church, "A Plan of Analysis for Software Engineering Laboratory Data," Computer Sciences Corporation Technical Memorandum, March 1983

Card, D. N., and M. G. Rohleder, "Report of Data Expansion Efforts," Computer Sciences Corporation, Technical Memorandum, September 1982

[3]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: Computer Societies Press, 1983

Freburger, K., "A Model of the Software Life Cycle" (paper prepared for the University of Maryland, December 1978)

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

Hislop, G., "Some Tests of Halstead Measures" (paper prepared for the University of Maryland, December 1978)

Lange, S. F., "A Child's Garden of Complexity Measures" (paper prepared for the University of Maryland, December 1978)

McGarry, F. E., G. Page, and R. D. Werking, Software Development History of the Dynamics Explorer (DE) Attitude Ground Support System (AGSS), June 1983

Miller, A. M., "A Survey of Several Reliability Models" (paper prepared for the University of Maryland, December 1978)

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (proceedings), March 1980

Page, G., "Software Engineering Course Evaluation," Computer Sciences Corporation, Technical Memorandum, December 1977

Parr, F., and D. Weiss, "Concepts Used in the Change Report Form," NASA, Goddard Space Flight Center, Technical Memorandum, May 1978

Reiter, R. W., "The Nature, Organization, Measurement, and Management of Software Complexity" (paper prepared for the University of Maryland, December 1976)

Scheffer, P. A., and C. E. Velez, "GSFC NAVPAK Design Higher Order Languages Study: Addendum," Martin Marietta Corporation, Technical Memorandum, September 1977

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

Weiss, D. M., "Error and Change Analysis," Naval Research Laboratory, Technical Memorandum, December 1977

Williamson, I. M., "Resource Model Testing and Information," Naval Research Laboratory, Technical Memorandum, July 1979

[3]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: Computer Societies Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]This document superseded by revised document.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.