NUREG/CR-6293
Vol. 1

Verification and Validation Guidelines for High Integrity Systems

Main Report

Manuscript Completed: December 1994
Date Published: March 1995

Prepared by
H. Hecht, M. Hecht, G. Dinsmore, S. Hecht, D. Tang

SoHaR Incorporated
8421 Wilshire Boulevard
Beverly Hills, CA 90211-3204

Under Contract to:
Harris Corporation
Information Systems
P.O. Box 9800
Melbourne, FL 32902

Prepared for
Division of Systems Technology
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001
NRC Job Code L2448

and

Nuclear Power Division
Electric Power Research Institute
3412 Hillview Avenue
Palo Alto, CA 94303

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

ABSTRACT

High integrity systems include all protective (safety and mitigation) systems for nuclear power plants, and also systems for which comparable reliability requirements exist in other fields, such as in the process industries, in air traffic control, and in patient monitoring and other medical systems. Verification aims at determining that each stage in the software development completely and correctly implements requirements that were established in a preceding phase, while validation determines that the overall performance of a computer system completely and correctly meets system requirements. Volume I of the report reviews existing classifications for high integrity systems and for the types of errors that may be encountered, and makes recommendations for verification and validation procedures, based on assumptions about the environment in which these procedures will be conducted. The final chapter of Volume I deals with a framework for standards in this field. Volume II contains appendices dealing with specific methodologies for system classification, for dependability evaluation, and for two software tools that can automate otherwise very labor intensive verification and validation activities.

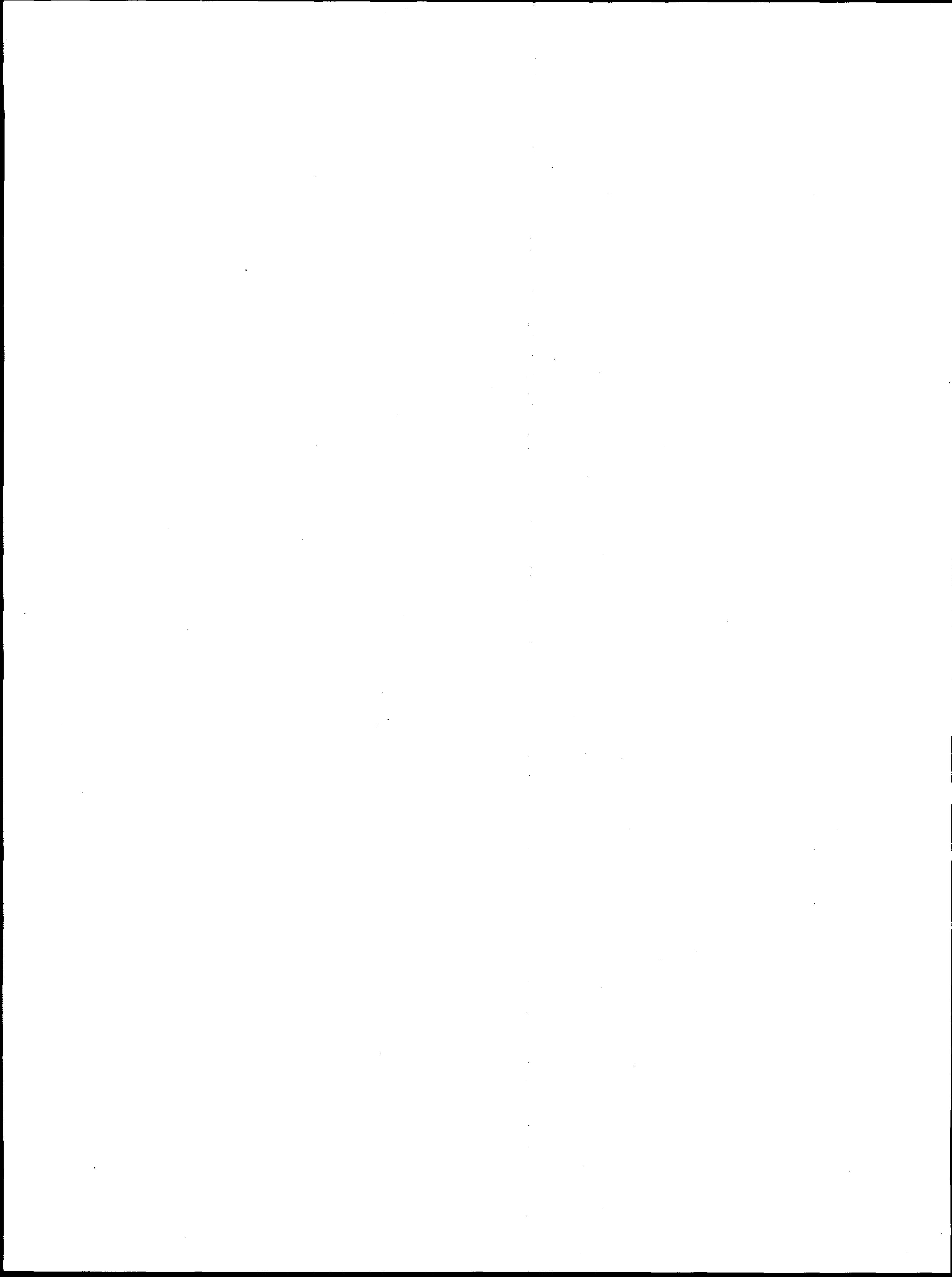


TABLE OF CONTENTS

LIST OF FIGURES ix
ABBREVIATIONS xi
GLOSSARY xiii
EXECUTIVE SUMMARY xix

SECTION 1 - INTRODUCTION 1-1

SECTION 2 - CLASSIFICATION OF HIGH INTEGRITY SYSTEMS 2-1
2.1 OVERVIEW 2-1
2.2 REGULATORY REQUIREMENTS AND CURRENT NUCLEAR STANDARDS 2-1
2.2.1 Requirements in the Code of Federal Regulations 2-1
2.2.2 Current and Pending Nuclear Standards 2-2
2.2.3 Evaluation of Current and Pending Standards 2-4
2.3 CLASSIFICATION IN OTHER STANDARDS 2-7
2.3.1 Process Industry Classifications 2-7
2.3.2 Classification in Military Standards 2-12
2.3.3 Discussion of Classifications from Other Fields 2-16
2.4 QUANTITATIVE RELIABILITY ASSESSMENT 2-16
2.5 CONCLUSIONS AND RECOMMENDATIONS 2-17

SECTION 3 - ERROR CLASSIFICATIONS 3-1
3.1 OVERVIEW 3-1
3.1.1 Motivation for Error Classification 3-1
3.1.2 Nomenclature and General Concepts 3-2
3.2 REVIEW OF PRIOR WORK 3-5
3.2.1 Administrative Error Classifications 3-5
3.2.2 U. S. Air Force Software Fault Classifications 3-6
3.2.3 Classifications in Connection with Specific Cause Hypotheses 3-10
3.2.4 Classifications in Connection with Fault Tolerance 3-13
3.2.5 Classifications from Recent SoHaR Projects 3-14
3.2.6 NASA Space Shuttle Avionics Failure Classification 3-20
3.3 REQUIREMENTS AND RECOMMENDATIONS FOR ERROR CLASSIFICATION 3-21
3.3.1 General Requirements 3-21
3.3.2 Fault Classification File 3-23
3.3.3 Failure Classification 3-24
3.3.4 Environment Classification 3-26
3.3.5 Cost Considerations 3-27
3.4 APPLICATION METHODOLOGY 3-29
3.4.1 Site Specific Applications 3-29
3.4.2 Global Applications 3-30
3.4.2 Application of Cost Data 3-33

3.5 CONCLUSIONS AND RECOMMENDATIONS 3-35

SECTION 4 - VERIFICATION AND VALIDATION OBJECTIVES 4-1
4.1 Overview 4-1
4.2 FREEDOM FROM FAILURE IN OPERATION 4-1
4.2.1 Definition of Normal Service 4-3
4.2.2 Failure Modes, Error Detection and Fault Tolerance Requirements . 4-3
4.2.3 Unsafe Actions 4-5
4.2.4 Human Interfaces 4-5
4.2.5 Isolation 4-6
4.2.6 Test 4-6
4.2.7 Attributes 4-7
4.3 DISTINCTIVE ROLES OF METRICS, VERIFICATION AND VALIDATION 4-7
4.4 CONCLUSIONS AND RECOMMENDATIONS 4-9

SECTION 5 - QUALITY METRICS 5-1
5.1 OVERVIEW 5-1
5.2 MAJOR FRAMEWORKS FOR QUALITY METRICS 5-4
5.2.1 Rome Laboratory Software Quality Measurement Methodology ... 5-4
5.2.2 SEI Capability Maturity Model 5-11
5.3 SURVEY OF SPECIFIC SOFTWARE METRICS 5-15
5.3.1 Line of Code Measure 5-16
5.3.2 Halstead's Software Science Measures 5-16
5.3.3 McCabe Cyclomatic Complexity Metric 5-19
5.3.4 Henry and Kafura's Information Flow Metric 5-19
5.3.5 Measure of Oviedo 5-20
5.4 NEW QUALITY METRIC 5-20
5.4.1 Metrics Requirement for High Integrity Software 5-21
5.4.2 Implementation of Extensions to the Halstead Metric 5-22
5.4.3 Metrics Evaluation 5-23
5.5 CONCLUSIONS AND RECOMMENDATIONS 5-29

SECTION 6 - VERIFICATION GUIDELINES 6-1
6.1 OVERVIEW 6-1
6.2 ORGANIZATION AND PLANNING OF VERIFICATION 6-4
6.2.1 Requirements for Independence 6-4
6.2.2 Discussion of Requirements 6-5
6.2.3 Verification Plans 6-7
6.2.4 Interfaces with QA and Configuration Management 6-9
6.3 VERIFICATION METHODOLOGIES 6-11
6.3.1 Reviews and Audits 6-11
6.3.2 Independent Equivalent Activity 6-12
6.3.3 Backward Reconstruction 6-13

6.3.4 Algebraic Methods ... 6-14
6.4 VERIFICATION IN THE LIFE CYCLE ... 6-15
6.4.1 The Ontario Hydro Life Cycle Activities ... 6-16
6.4.2 Evaluation of the Ontario Hydro Verification Methodology ... 6-24
6.4.3 Life Cycle Activities for the U. S. Environment ... 6-25
6.4.4 Verification of Isolated Non-Critical Segments ... 6-26
6.4.5 Verification by Reverse Engineering ... 6-27
6.5 SPECIAL VERIFICATION CONCERNS ... 6-28
6.5.1 Commercial and Reused Software ... 6-28
6.5.2 Compilers ... 6-29
6.5.3 Tools ... 6-30
6.5.4 Process Audits ... 6-31
6.6 CONCLUSIONS AND RECOMMENDATIONS ... 6-32

SECTION 7 - VALIDATION GUIDELINES ... 7-1
7.1 OVERVIEW ... 7-1
7.1.1 Motivation ... 7-1
7.1.2 Structure of this Chapter ... 7-2
7.2 TEST METHODOLOGY ... 7-3
7.2.1 Functional Testing ... 7-3
7.2.2 Structural Testing ... 7-6
7.2.3 Statistical Testing ... 7-9
7.2.4 Relative Evaluation of the Test Methodologies ... 7-12
7.2.5 Validation of Requirements ... 7-14
7.2.6 Validation of Diagnostics ... 7-15
7.3 TEST TERMINATION CRITERIA ... 7-16
7.3.1 Test Termination for Functional Test ... 7-16
7.3.2 Test Termination for Structural Test ... 7-18
7.3.3 Test Termination for Statistical Testing ... 7-19
7.4 VALIDATION OF COMMERCIAL SOFTWARE ... 7-24
7.5 CONCLUSIONS AND RECOMMENDATIONS ... 7-25

SECTION 8 - STANDARDS FRAMEWORK ... 8-1
8.1 OVERVIEW ... 8-1
8.2 TOP LEVEL OF THE FRAMEWORK ... 8-2
8.2.1 Requirements for the Top Level ... 8-2
8.2.2 Recommended Structure ... 8-3
8.2.3 Additional Information to be Supplied ... 8-5
8.3 LOWER LEVEL STANDARDS FRAMEWORK ... 8-5
8.3.1 Life Cycle Phases ... 8-5
8.3.2 System and Software Requirements ... 8-6
8.3.3 Software Development or Procurement ... 8-6
8.3.4 Licensing Activities ... 8-6
8.3.5 Acceptance Testing ... 8-7

8.3.6 Other Activities 8-7

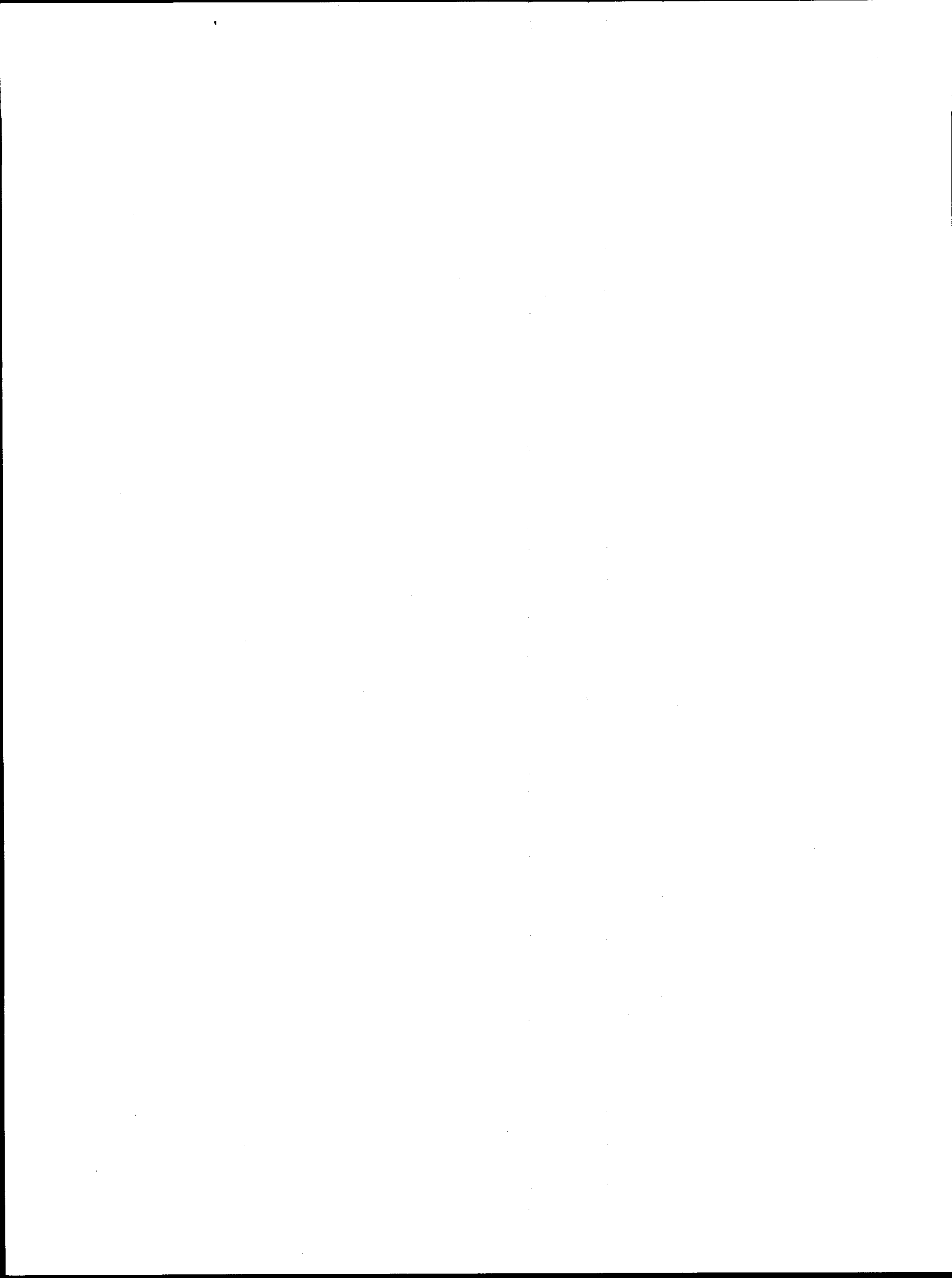
SECTION 9 - SUMMARY CONCLUSIONS AND RECOMMENDATIONS 9-1

REFERENCES REF-1

STANDARDS REFERRED TO STANDARDS-1

LIST OF FIGURES

FIGURE	TITLE	PAGE
Figure 2.3-1	Recommended Verification practices in IEC65A(Sec)123	2-10
Figure 2.3-2	Recommended Black Box Testing in IEC65A(Sec)123	2-11
Figure 2.3-3	Risk Definition in MIL-STD-1629	2-13
Figure 3.1-1	Generic Failure Model	3-3
Figure 3.1-2	Propagation of Failure Effects	3-4
Figure 3.1-3	Failure Model for Fault Tolerance Provisions	3-4
Figure 3.2-1	Use of TRW Classification	3-9
Figure 5.2-1	Software Quality Model	5-5
Figure 5.2-2	Reliability Factor	5-7
Figure 5.2-3	Metric Worksheet Part 1	5-9
Figure 5.2-3	Metric Worksheet Part 2	5-10
Figure 5.4-1	Failure Data of the Evaluated Software	5-26
Figure 5.4-2	Regression on Halstead Metric	5-27
Figure 5.4-3	Regression on AR Halstead Metric	5-28
Figure 6.4-1	Development and Verification Interfaces in the OH Environment	6-17
Figure 7.2-1	Branch coverage as a function of test cycles	7-11
Figure 7.3-1	Partitioning a program	7-18
Figure 7.3-2	Reliablility Model	7-20
Figure 7.3-3	Progression of failure types during test	7-22



ABBREVIATIONS

ANS American Nuclear Society

ANSI American National Standards Institute

ASME American Society of Mechanical Engineers

CATS Code Analyzer Tool Set

DID Design Input Documentation

EPRI Electric Power Research Institute

ESA European Space Agency

ECT Enhanced Condition Table

FAT Factory Acceptance Test

FIPS Federal Information Processing Standards

FRACAS Failure Reporting, Analysis and Corrective Action System

FRB Failure Review Board

IEC International Electro-technical Commission

IEEE The Institute of Electrical and Electronics Engineers, Inc.

ISA Instrument Society of America

I/O Input/Output

METBF Mean Execution Time Between Failures

MTBF Mean Time Between Failures

MIL-STD Military Standards

MOD Ministry of Defense (United Kingdom)

NIST National Institute of Standards and Technology

NRC Nuclear Regulatory Commission

NUREG/CR Nuclear Regulatory Commission Contractor Report

SAT Site Acceptance Test

SDD Software Design Description

SP Special Publication

SQA Software Quality Assurance

SRS Software Requirements Specification

V&V Verification and Validation

GLOSSARY

Acceptance test A series of system tests are performed on the delivered software and usually the acceptance of the software is made contingent upon the successful completion of these tests. This term is also used for fault-tolerant software or defensive programming in which acceptance test is the means of checking computational results for on-line error detection.

Accuracy Characteristics of software which provide the required precision of calculations and output.

Assessment Activity of an independent person or body with the responsibility for assessing whether all safety and other requirements for the software have been achieved.

Anomaly management Ability to provide for continuity of operations during and in recovering from non-nominal conditions.

Autonomy Independence from implementation of interfaces and functions.

Availability Dependability with respect to the readiness for usage. Measure of correct service delivery with respect to the alteration of correct and incorrect service.

Benign failure Failure whose penalties are small compared to the benefit provided by correct service delivery.

Class 1 E The safety classification of electric equipment and systems that are essential to emergency reactor shutdown, containment isolation, reactor core cooling, and containment and reactor heat removal, or are otherwise essential in preventing significant release of radioactive material to the environment.

Code A uniquely identifiable sequence of instructions and data which is part of a module (e.g., main program, subroutine, and macro).

Common Mode Failure Simultaneous failures in multiple components due to a common cause

Completeness Provision for full implementation of the required function.

Consistency Uniformity of design and implementation techniques and notation.

Correctness Extent to which a program conforms to its specifications and standards.

Criticality This term is used in the DoD reliability standards (e. g., MIL-STD-1629) for what is here referred to as *Risk*. See Risk.

Defense in depth Multiple provisions to protect against, or to mitigate, failures.

Design Fault A design (coding, specification) fault results from a (human) mistake during the design of a system. A design fault causes an error, residing undetected within a (sub)system, until the input values to that (sub)system are such that the produced result does not conform to the specification. This constitutes the failure of that (sub)system. If the same input values appear again, the same erroneous results will be produced.

Diversity Existence of different means of performing a required function, for example, other physical principles, others ways of solving the same problem. See also *Functional Diversity* and *Software Diversity.*

Document accessibility Ease of access to software and documentation, particularly to permit selective use of components.

Efficiency Relative extent to which a resource is utilized (i.e. storage space, processing time, communication time).

Environment The environment in which a *system* operates, including exposure to weather, the electric supply, communication lines, heating, ventilation and air conditioning, etc.

Error A discrepancy between a computed, observed, or measured value or condition and the true, specified or theoretically correct value or condition. See also Chapter 3.

Expandability Relative effort to increase the software capability or performance by enhancing current functions or by adding new functions or data.

Fail-Safe The built-in capability of a system such that predictable (or specified) equipment (or service) failure modes only cause system failure modes in which the system reaches and remains in a safe fall-back state.

Failure The termination of the ability of a functional unit to perform its required function. See also Chapter 3.

Fault An accidental condition that causes a functional unit to fail to perform its required function. See also Chapter 3.

Fault avoidance The use of design techniques which aim to avoid the introduction of faults during the design and construction of the system.

Fault tolerance Methods and techniques aimed at providing a service complying with the specification in spite of faults.

Flexibility Relative effort for changing the software missions, functions, or data to satisfy other requirements.

Functional diversity Implementation of a single protection requirement by two or more independent systems, operating on different plant parameters and using different algorithms, e. g., a trip system that is actuated on the basis of sensed neutron flux and on the basis of sensed temperatures or pressures.

Independence Ability to operate in changed environments (computer system, operating system, utilities, libraries).

Independent Department An Independent Department is a department which should be separate and distinct by ways of finance, management and other resources from the main development and maintenance of safety-related software and should not have direct responsibility for these main activities.

Independent faults Faults attributed to different causes.

Independent organization An Independent organization is one which is separate and distinct by ways of finance, management and other resources from the organization responsible for development and maintenance of safety-related software and shall not have direct responsibility for these main activities.

Independent person An Independent Person is a person who should be separate from the main development and maintenance of safety-related software and should not have direct responsibility for these activities.

Independent systems Systems that will not fail due to a common cause within the plant design basis.

Independently developed programs Programs that have been designed and developed by independent organizations or departments with the aim of minimizing the probability of common cause failures.

Intermittent fault Temporary internal fault. Faults whose conditions of activation cannot be reproduced or which occur rarely enough.

Integrity Extent to which the software will perform without failures due to unauthorized access to the code or data within specified time period.

Interoperability Relative effort to couple the software of one system to the software of another system.

Maintainability The ability of an item under given conditions to be retained in or restored to a state, in which it can perform the required function.

Measure A measure is a function of metrics which can be used to asses or predict more complex attributes like cost or quality.

Metric Metrics numerically characterize simple attributes like length, number of decisions, number of operators (for programs), or number of bugs found, cost, and time (for processes).

Mistake A human action that produces an unintended result.

Modularity Provisions for a structure of highly cohesive components with optimum coupling.

Operational fault Faults which appear during the system's operation.

Plant The entity monitored, serviced, or controlled by the *system*; typically a power plant.

Portability Relative effort to transport the software for use in another environment (hardware configuration and/or software system environment).

Programmable Logic Controller (PLC) A solid-state control system which has a user programmable memory for storage of instructions to implement specific functions.

Redundancy Provision of additional elements or systems so that any one can perform the required function regardless of the state of operation or failure of any other. Redundancy can be implemented by identical elements (identical redundancy) or by diverse elements (diverse redundancy).

Regression testing Systematic repetition of testing to verify that only desired changes are present in the modified programs.

Reliability Extent to which the software will perform without any failures within a specified time period.

Reusability Relative effort to convert a software component for use in another application.

Risk A relative measure of the consequences of a failure mode and its frequency of occurrence. Equivalent to *Criticality* as used in some DoD standards.

Safe State A state of a defined system in which there is no danger to human life: limb and health, economics or environment under certain assumptions and specified conditions.

Safety The expectation that a system does not, under defined conditions, lead to a state in which human life: limb and health, economics or environment are endangered.

Safety critical software Software that falls into one or more of the following categories: (1) software whose inadvertent response to stimuli, failure to respond when required, response out-of-sequence, or response in combination with other responses can result in an accident; (2) software that is intended to mitigate the result of an accident; (3) software that is intended to recover from the result of an accident [IEEE 1228]. Except in quotations from other documents, this report does not distinguish between safety critical and safety related software (or functions or equipment).

Safety Integrity The likelihood of a plant protection system achieving its safety functions under all stated conditions within a stated period of time.

Self-descriptiveness Explanation of the implementation of functions (associated with the source code).

Service The function or operation which the *system* furnishes to the *plant*; examples are reactor trip and emergency core cooling.

Software Intellectual creation comprising the programs, procedures, rules and any associated documentation pertaining to the operation of a data processing system.

Software diversity Implementation of a single protection requirement by two or more independently developed programs that operate from the same plant parameters, e. g. N-version programming or recovery blocks. These different versions are frequently coded in different languages.

Software lifecycle The activities occurring during a period of time that starts when software is conceived and ends when the software is no longer available for use. The software lifecycle typically includes a requirements phase, development phase, test phase, integration phase, installation phase and a maintenance phase.

Software quality The degree to which software possesses attributes desired by the user.

Software safety integrity The likelihood of software on a Programmable Electronic System achieving its safety functions under all stated conditions within a stated period of time.

System as used in this report refers to a high integrity system that furnishes an essential *service* to a *plant*. The system typically includes computer hardware, software, display interfaces, and output devices, such as relays, but not major plant equipment such as pumps or control rod actuators. The physical parts of operator interfaces are included (e. g., trip buttons), but human action or reaction are not.

System accessibility Control and audit of access to the software and data.

System clarity Clarity of program description, particularly with regard to program structure.

Testability Effort required to test a program to insure it performs its intended function, and the absence of unintended functions.

Traceability Ability to provide a thread from origin to implementation of requirements with respect to the specified development and operational environment.

Training Facility for providing familiarization and easing transition from previous operations.

Trigger An event or condition that precipitates a failure, such as an incorrect operator command or a noisy communication line.

Usability Relative effort for using software or the system containing software (training and operation).

User Another system (physical, human) interacting with the considered system.

Validation The test and evaluation of the integrated computer system to ensure compliance with the functional, performance, and interface requirements.

Verifiability Relative effort to verify the specified software operation and performance.

Verification The process of determining whether or not the product of each phase of the software development process fulfills all the requirements imposed by the previous phase

Visibility Provision for status monitoring of the development and operation.

EXECUTIVE SUMMARY

The major conclusion of this report is that verification and validation (V&V) are open-ended activities without natural completion criteria. Where V&V of a safety critical system is specified as comprising a limited set of tasks, this specification is necessarily based on the experience or subjective evaluation of the decision makers, on resource limitations, or on a combination of these. The research reported on here has not found generally applicable data that show a useful relation between verification methodology or resource expenditure and the reliability of the resulting product. While methodologies and tools are probably available to verify the absence of any one cause of safety impairment, there is no practicable set that will cover all possible causes. The risk arising from these limitations can be minimized by keeping the safety critical part extremely simple and well isolated, by using two or more functionally diverse programs or complete systems, and by research into the nature and causes of failures.

Requirements for nuclear safety systems are derived from Title 10 of the Code of Federal Regulations, Part 50 (10CFR50). This document does not address specific concerns that arise from the application of digital technology. A key issue in developing guidance for V&V is the classification of safety systems (Section 2 of this report). A multi-level classification permits less intensive V&V for less critical applications, and this would be beneficial where functionally diverse programs are provided, or where support functions (diagnostics, report generation) are included in a critical application but are isolated as far as failure effects are concerned. 10CFR50 neither precludes nor mandates multi-level safety classifications, but the predominant current practice for electrical and electronic systems (either analog or digital) is based on a single safety classification, 1E.

Multi-level safety classifications are widely employed in the process industries and in the military services, and a multi-level classification has now been adopted for nuclear instrumentation and control systems in IEC 1226, "Classification of Instrumentation and Control Systems Important to Safety of Nuclear Power Plants". Multi-level classifications provide a systematic approach to reducing the verification requirements where diverse software or functional diversity are employed, and they can also reduce the requirements for well isolated non-mainline components in critical programs. Support for acceptance of these standards is therefore recommended.

Most safety standards outside the nuclear field now employ a risk-based classification, where risk is defined as a function of probability of occurrence of an event and the severity of consequences associated with the event. These are inherently multi-level and therefore provide the benefits mentioned above. In addition, they permit experience relative to frequency of failure and severity of failure that is accumulated in use to be factored into the safety requirements. This motivates the design and deployment of highly reliable systems, and also promotes objective trade-offs of diversity vs. quality, and of increasing quality at one layer vs. adding another layer of plant protection. Appendix A of the report provides guidelines for a risk based classification of digital equipment in protection systems for nuclear reactors.

Because only very limited data from nuclear power plants are available, the chapter on error classification (Section 3) has utilized data from other sources to draw some inferences, but the limitations of that process had to be recognized. Examples of findings that are particularly relevant are:

- software involved in redundancy management and for fault tolerance was a leading cause of serious failures in at least one environment

- incorrect response rather than complete loss of computational capability was the leading error manifestation in another environment (this reduces the credibility of error detection provisions that only respond to complete cessation of computation)

- inability to handle multiple rare conditions that were encountered in close time proximity was the leading cause of failures in a third environment. Note that this finding is consistent with the first one mentioned above since fault tolerance management is frequently involved in multiple rare conditions

Because data on the nature and causes of failure in software for nuclear applications is so essential to the development of a verification and validation policy, and the evaluation of supporting methodologies, it is suggested that mechanisms be explored for obtaining failure data on new and existing digital systems in nuclear plants. An error classification is proposed that can be used in a database for the formulation of V&V policies and methodologies, and that can contribute to the objective evaluation of software for high integrity applications.

Metrics (Section 5) are desirable because they may furnish a quantitative and, it is hoped, objective assessment of software attributes important for safety that can at present only be characterized in a qualitative and subjective manner. A potentially important use of metrics is to identify troublesome software segments so that corrective action can be taken by the developer or very concentrated auditing can be applied by the licensing agency. Among available metrics none were found capable of meeting these objectives. Many current metrics are primarily intended for control or improvement of the software development process; these may be very beneficial for high integrity software in an indirect way, but recent literature notes a lack of evidence for a "return on investment" for process improvements. There is no evidence that process control is detrimental to product attributes important to safety.

The greatest difficulty encountered in the metrics area is the lack of metrics that can be obtained early in the development and that have demonstrated high correlation with relevant later metrics such as fault density or failure rate. While the quest for metrics valid during early life cycle phases should be continued, the aim of early correction can also be achieved by use of a spiral development approach, particularly when executed in accordance with the paradigm "build a little, test a lot." This will not only provide early indications of problem areas for a given software product but may also serve as a testbed for validating metrics in a specific environment.

While there is no lack of publications on verification methodologies (Section 6), there is an absence of conclusive evidence of how effective these methodologies have been in reducing failure rates to the level required for the high integrity systems addressed in this report. In addition, the administration of verification and validation in the U. S. nuclear industry differs sharply from that of the aerospace and defense industries where most verification practices originated. In the latter environments the user or customer contracts separately with an independent organization to verify the software products of the developer, whereas in the U. S. nuclear industry the developer is frequently responsible for the conduct of verification (or of substantial portions of it). This makes some widely practiced and standardized verification procedures inappropriate or of limited value.

In these circumstances the verification practices adopted by Ontario Hydro (OH) in connection with the licensing of the Darlington reactor protection system (and further developed for other safety systems) offered the best basis for recommendation for the current U. S. nuclear power environment. There is as yet little experience with the reliability of the software developed and verified under the OH procedures, and there is also a significant administrative concern in that OH had continuous and open access to the software products from the earliest development stages, whereas U. S. utilities as well as the NRC typically have access only after most of the development is complete. In spite of these reservations, the OH procedures offer these benefits:

- they have been reviewed by nuclear and software professionals, and are open for examination by any interested party; no significant objections to the procedures are known

- no negative experiences have been reported in the operation of the Darlington plant

- they are specifically tailored for the nuclear power environment.

The labor required for verification can be considerably reduced by the use of tools, and these have additional benefits in enforcing a systematic approach and in reducing the possibility of mistakes. Thus, tool use should be encouraged. However, a number of caveats must be recognized:

- tools are frequently language dependent, and selection of some languages may severely restrict the availability of tools.

- tools may themselves contain faults and must therefore be verified (see Section 6.5.3)

- to further reduce the possibility of faults introduced by tools, the verifiers should use tools that are different from those used in the development.

Of particular interest are tools that can be applied to the delivered software (either the source code or the machine code), and that permit some verification activities to be carried out completely independent of the developer. Two such tools, ECT and CATS, are described in Appendices C and D of this report. Both tools are just emerging from research and currently

have limitations that preclude general use, but they offer an avenue of largely automated verification that is particularly suitable for the nuclear safety system environment.

Validation (Section 7), conducted at the system level (computer system or higher), with end-to-end testing being the major activity, is the last bulwark against placing an inadequate or faulty system into operation. Validation is a comparison of system capabilities against the requirements, and therefore complete and correct requirements are a prerequisite for successful validation. Validation uses the products of the verification process to establish that the system development has been carried out in accordance with an acceptable process, and that discrepancies discovered during reviews and pre-system testing have been corrected.

A combination of functional, structural, and statistical testing is recommended. Preferably all tests are carried out with a test harness that permits measurement of structural coverage and that identifies untested paths in critical portions of the program and at least branches and conditions in non-critical parts. Functional testing is primarily aimed at establishing that all requirements are implemented, structural testing identifies paths or branches that have not been accessed during functional test (and that could lead to unwanted actions), and statistical testing is conducted to establish the reliability of the system, and as further safeguard against unintended functions.

The most significant issue in validation is to determine how much test is required, i. e. to identify a criterion for test termination. The implicit termination criteria for functional and structural test (e. g., to access every requirement or every branch) are not sufficient for high integrity computer systems because they do not include testing for coincident requirements or combinations of branch conditions. To overcome these limitations, statistical testing in an environment that generates test cases corresponding to multiple rare conditions has been recommended, and a test termination criterion for this type of test has been developed. While not rigorous, this provides an objective means of establishing that the goals of validation have been attained. Further research and experimentation on the criteria and on the integrated approach to use of the three test methodologies is recommended.

Since regulators may rarely be in a position to conduct tests themselves, the key activities are

- review of test plans: provision for functional, structural, and statistical testing
- establish test termination criteria consistent with the recommendations of Section 7
- approve test reports: compliance with the plans and test specifications, use of appropriate tools, identification of difficulties encountered and explanation of their potential effect on plant safety, assurance of adequate retest after modification of any part of the software (including requirements through code and documentation).

It is reasonable to insist that all documentation furnished in connection with validation be understandable by a person not familiar with the specific development and test techniques or tools used by the performing organization.

The purpose of the review of standards (Section 8) is to investigate the feasibility of a framework that clearly propagates the statutory and operational safety requirements into verification and validation practices. The most desirable outcome of this investigation is to identify a few standards of broad scope that in turn reference more detailed standards for individual activities and documents. This goal was not attained, and, on the contrary, the finding is that current standards represent a patchwork with considerable gaps, overlaps, and inconsistencies. Recommendations are made for one or more top level standards that

- implement the statutory provisions from 10CFR50
- conform to the best prevailing software practices as represented by standards of the field
- recognize the need for economical procurement and operation of digital protection systems on the part of the user (utilities).

A summary of lower level standards is provided that may be referenced in the top level standard(s).

SECTION 1 - INTRODUCTION

This is the final report on a study of Verification and Validation Guidelines for High Integrity Digital Systems. This work was performed under USAF Task Order Contract F30602-89-D-0099 between USAF Rome Laboratory and Harris Corporation. SoHaR Incorporated received a subcontract from Harris Corporation for Task 8 of the prime contract which includes all of the work reported on here. The effort was jointly sponsored by the Nuclear Regulatory Commission and the Electric Power Research Institute.

The term *High Integrity Systems* in the title of this effort is intended to include all protective (safety and mitigation) systems for nuclear power plants, and also systems for which comparable reliability requirements exist in other fields, such as the process industries, air traffic control, and aerospace control and monitoring systems. Depending on the needs of the application, high integrity implies high reliability (freedom from failures, regardless of consequence and duration), high safety (freedom from failures that produce severe consequences), high availability (freedom from failures that cause long outages), or high security.

The Statement of Work describes the background for this effort as follows:

> As analog hardware for safety systems in nuclear power plants becomes 20 to 30 years old, component failure and maintenance costs increase. The obsolete safety grade components are difficult to replace with similar analog components. [This motivates a process in which] analog based safety systems are being replaced with digital safety systems. This replacement with digital systems also results in functional improvements, such as reduction in system calibration drift due to continuous on-line calibration.
>
> A major difference between analog hard-wired systems and digital systems is the logic stored in the digital computer's memory. The design, development, and test of this logic is an error-prone process. One logic error common to all redundant channels may result in the loss of the safety function. The use of design verification and validation methods enhances the quality of the software through independent reviews of the development process and the product. Design methods and design verification and validation methods are changing rapidly because of technological advances. For example, object oriented design and the use of formal methods for design are new techniques. While the NRC has some guidelines for the verification and validation of safety grade software, they are out of date regarding advancing technology. The purpose of this research is to upgrade existing guidelines and improve on them to reflect current technology.

Verification and validation have been used for well over twenty-five years in providing assurance of fault-free operation in computer based defense systems. As programmable digital systems were introduced into other applications demanding very high reliability, the verification and validation techniques developed in the defense sector were carried over into these. A primary

benefit of verification and validation is that a technically competent party other than the developer performs a systematic and critical review of the software products and, sometimes, also of the development process. At the very least this assures that the software is reviewable, i. e., there is demonstrable traceability from requirements to design, to code, and to test conditions. But the degree to which V&V provides assurance that the software is free of faults is not so readily assessed because of the following circumstances:

- high integrity software should not fail at all, but even if this latter condition is relaxed to allow failure probability of 10^{-6} or 10^{-7} per year it precludes the opportunity to observe attainment of this goal in the operational environment

- the conditions most likely to induce software failure are combinations of unusual states or events, and exhaustive testing for these is impossible

- too little is known about the causes of failure in high integrity software, and particularly in software for nuclear plant protection systems, to formulate specific review and test procedures to target these (the recommendations of Chapter 3 partially address this problem).

The authors of this report are aware of the need for procurement and regulatory guidance in spite of these difficulties, and of the existence of standards and related documents that provide such guidance in response to these needs. These documents present a majority opinion or consensus of knowledgeable participants, but they do not necessarily represent the only conclusions that can be drawn from the available facts. Examples of the subjectivity that can be found in recently issued guidance documents are the following:

RTCA DO-178B Airborne systems, 1992	IEC 1226 Nuclear power plants, 1993
Assignment of software integrity requirements based on consequences and specifically not related to failure rate (par. 2.2.3).	Integrity requirements based on "either a quantitative probabilistic assessment of the Nuclear Power Plant, or by quantitative engineering judgement" (par. 8.2.1).
Higher integrity requirements are satisfied by increased independence of the review process (Annex A).	Higher integrity requirements are satisfied by increased redundancy and functional independence (par. 8.2.2).

Absent a guarantee of fault-free operation, and faced with divergent approaches in existing guidance documents, this research focused on establishing how the probability of failure in an unsafe manner could be verified to be sufficiently low to be acceptable in specific situations. The following defense-in-depth scenarios were postulated:

1. A safety function backed up by another (safety or non-safety) function

2. A very simple safety function (e. g., one having a single input and a single output), providing a service the failure of which can be mitigated by other plant functions.

3. A safety function that is neither very simple (as in the above example) nor backed up by another function.

For the first two scenarios the verification and validation methodologies described in Chapters 6 and 7 will be able to distinguish between products that do and do not have an acceptably low failure probability. For scenario 3 this capability does not yet exist, but by using two functionally diverse implementations that scenario can be converted into scenario 1.

The body of this report is organized into the following chapters:

2 .	Classification of High Integrity Systems
3 .	Error Classifications
4 .	Verification and Validation Objectives
5 .	Software Metrics
6 .	Verification Methodology
7 .	Validation Methodology
8 .	Standards Framework
9 .	Summary Conclusions and Recommendations

Appendices are furnished as a separate volume and comprise:

Appendix A	Risk Based Classification Guidelines
Appendix B	Measurement Based Dependability Evaluation
Appendix C	Description of the Enhanced Condition Table (ECT) Tool
Appendix D	Description of the Code Analyzer Tool Set (CATS)

The organization within most chapters is to present an overview, followed by discussion of current practices in the nuclear industry and in related fields, and then to assess these against requirements that lead to a technically effective assessment of candidate systems, and that also

promote design and application of systems that will render the desired protective (or other) service without excessive resource requirements.

This is a report on research in a very difficult field. The difficulty arises from the lack of clear completion criteria for verification and validation, activities which are in many ways similar to a medical check-up. The physician can spend a half-hour, two hours, or a full day in examining a patient, and the number of concerns identified will probably increase with the thoroughness of the examination, but no check-up can provide a complete assurance of freedom from disease. Therefore decisions about the scope of verification and validation necessarily involve some subjectivity; yet some methodologies are clearly superior to others, and some steps of verification and validation should never be omitted. In order to determine which are superior or essential activities it was necessary to examine many which were found not to fit into these categories. Documentation of the selection process (and reasons for rejection of some techniques) was considered an essential part of the task. Thus, considerable portions of some chapters contain material that is not directly connected to the selected techniques. These headings have been marked with an asterisk (*) and can be omitted by readers willing to accept our recommendations at face value. The essential material for the conduct of verification can be found in Sections 6.3 and 6.4, and similarly for validation in Sections 7.2 and 7.3.

References and a listing of significant standards are found at the end of the report.

SECTION 2 - CLASSIFICATION OF HIGH INTEGRITY SYSTEMS

2.1 OVERVIEW

The safety and availability requirements for high integrity systems depend on the expected loss upon failure of the specific function being analyzed. An effective classification should take account of this variation so that the safety requirements are set neither too high nor too low. The expected loss referred to depends on (a) the consequences of a complete failure in the absence of any protective measures (other than those which may be inherent in the function, such as self-quenching. (b) the availability of protective measures and mitigation outside the system under consideration (defense in depth), and (c) the probability of failure of the system under consideration. This chapter describes and evaluates classification formats in current use. Detailed classification guidelines, e. g., assessing the degree of protection afforded by manual overrides, are not provided.

Section 2.2 reviews the regulatory basis for classification and current or emerging standards in the nuclear field. It is found that these do not address significant issues for software based protection systems, notably diversity (software and functional), and the provision of support functions (such as diagnostics) in programs serving safety critical needs. Section 2.3 covers classification standards in the non-nuclear field, primarily in process control and the military services. It is found that risk based classifications are widely used in these fields, a practice not currently used in the nuclear field but which can provide significant benefits once obstacles, such as the lack of failure data, are overcome.

Section 2.4 introduces quantitative classification criteria for plant protection systems and discusses their relations to the qualitative ones proposed in the preceding section. Conclusions and Recommendations are presented in Section 2.5. A guideline for risk based classification in the nuclear field is contained in Appendix A.

2.2 REGULATORY REQUIREMENTS AND CURRENT NUCLEAR STANDARDS

2.2.1 Requirements in the Code of Federal Regulations

The following excerpts from the Title 10 of the Code of Federal Regulations (10CFR) imply or establish a need for classification of safety critical systems.

10CFR50.34 invokes Appendix A - *General Design Criteria for Nuclear Power Plants*. Criterion 1 of the Appendix is titled *Quality Standards and Records* and states "Structures, systems, and components important to safety shall be designed, fabricated, erected and tested to quality standards commensurate with the importance of the safety functions to be performed." This statement forms a basis for classification based on function served but does not establish the

format for classification. It permits classification on a dual level (important to safety vs. not important) as well as multiple levels (ranked in order of their importance to safety).

Appendix B - *Quality Assurance Criteria for Nuclear Power Plants and Fuel Reprocessing Plants* is also invoked in 10CFR50.34. The introduction to Appendix B states in part "The pertinent requirements of this appendix apply to all activities affecting the safety-related functions of those structures, systems, and components; these activities include ..." This wording can be interpreted as leaning more decisively toward a dual level classification, since all pertinent requirements will have to be met by all safety-related functions. However, in Section II - *Quality Assurance Program*, it is stated: "The quality assurance program shall provide control over activities affecting the quality of identified structures, systems, and components to an extent consistent with their importance to safety." This appears to allow for differentiating safety requirements into a number of categories.

Appendix E - *Emergency Planning and Preparedness for Production and Utilization Facilities* is invoked by 10CFR50.34(a) and (b). Part C of this Appendix is titled *Activation of Emergency Organization* and reads in part "The emergency classes defined shall include: (1) notification of unusual events, (2) alert, (3) site area emergency, and (4) general emergency. These classes are further discussed in NUREG-0654;FEMA REP-1." This report, jointly issued by the Nuclear Regulatory Commission and the Federal Emergency Management Agency, cites specific systems and conditions under which each of the emergency classes are to be activated. This classification differs from those discussed above in that it is based on severity of effects (or potential effects) rather than on functions served. It implies a five level classification scheme (no action required plus the four listed action levels). The distinction between effects related classifications and those based on function served is quite significant as the former directly supports risk based classifications.

2.2.2 Current and Pending Nuclear Standards

IEC 1226 "Classification of Instrumentation and Control Systems Important to Safety of Nuclear Power Plants" is an issued standards document. The target of the classification is identified as *FSE* (functions, systems and equipments) and in most cases no distinction is made among these subdivisions. It establishes three classes of FSEs important to safety, and by implication a fourth class that is not important to safety, thus being a four level classification scheme. Category A denotes an FSE which plays a principal role in the achievement or maintenance of safety; Category B denotes FSEs that play a complementary role to Category A in the achievement or maintenance of safety; and Category C denotes FSEs that play an auxiliary or indirect role in the achievement or maintenance of safety. The document includes assignment criteria, and it levies specific requirements for each category in the areas of functionality, performance, reliability, environmental durability, and QA/QC. The standard is intended for I&C systems, and the requirements are stated in terms that are meaningful to such systems. The document does not explicitly provide reduced requirements for diverse implementation of critical functions, although some interpretations may permit this. Members of the working group have indicated that further

work in this area is pending. As a matter of interpretation, reduced classification may be assigned to well isolated calibration and diagnostic functions within Category A primary functions (by declaring these separate functions to which category B or C can be assigned), but explicit guidance is lacking at present. There are no known uses of the standard in the United States.

ANS-58.14, Draft 9 (as of late 1993), "Safety and Pressure Integrity Classification Criteria for Light Water Reactors". This document is based on 10CFR50 Appendix B and claims to implement these provisions. It recognizes safety-related (Q), supplemented grade (S), and non-safety (N) categories. Category S is applied to items which do not perform a safety-related function but for which significant licensing requirements or commitments exist. In most cases the requirements levied on grade Q and S parts are identical, and therefore the standard represents a dual classification (safety grade and non-safety). Significant distinctions are drawn between requirements that apply at the function, system, component and part levels. Most of the provisions are directed at fluid and structural components.

ANS-50.1, Draft 6, "Nuclear Safety Design Criteria for Light Water Reactors". At the time of this report it was not known how close this draft is to becoming a standard. It refers to classification criteria in ANS-58.14 (see above) but adds reasoning based on probability of occurrences in translating the safety classification into design criteria. The content is organized by function served rather than by the method of implementation (analog vs. digital). While the basic design criteria cover any implementation, the standard provides no guidance for specific software processes such as code verification. The consideration of the probability of occurrence (of events that challenge the protection system or of failures in the system) is an important concept that will be referred to in later discussion.

IEEE/ANS Std. 7-4.3.2-1993,"Standard Criteria for Digital Computers in Safety Systems of Nuclear Reactors". The document does not contain classification guidelines but implicitly addresses Class 1E requirements. The foreword acknowledges the benefits of "graded" requirements but leaves this topic for future consideration. The function of "barriers" between safety-related and non-safety-related software executing on the same computer is mentioned in par. 5.6, but no specific requirements are established. The foreword acknowledges the need to address this topic in future efforts.

A working group on classification has been formed within the Electric Power Research Institute (EPRI) with representatives from vendors, utilities, and the government. It uses existing systems as indicators for classification[1]. A comparison of its categories with those of existing documents is shown below. The EPRI working group is recommending "adjustment factors" to account for diversity and product quality.

[1] The effort has published an EPRI "Verification and Validation Handbook"

Table 2.2-1. Classification under Evaluation by EPRI Working Group

EPRI Classification Working Group	ANS 58.14	IEC 1226	IEEE 603
Reactor Protection Systems	Q	A	1E
Engineered Safety Features	Q	A	1E
REGGUIDE 1.97 Type A	Q	A	1E
Aux. Syst. for RPS & ESFAS	Q/S/N	A/B	1E
Other Aux. Systems*	Q/S/N	A/B	1E
Supplementary REGGUIDE 1.97	S	B	NON-1E

* whose failure can affect the operation of the first three groups

Although the above table shows agreement between ANS Class Q, IEC Class A, and IEEE Category 1E in several rows, this does not imply that these categories are equivalent in all instances.

2.2.3 Evaluation of Current and Pending Standards

The service based classifications in the existing standards are comparatively easy to apply, although there are borderline cases that have caused some utilities to employ classification specialists. None of the standards discussed in the preceding section deals adequately with issues that arise from the use of programmable (software controlled) computers in safety critical functions. One such issue is the use of identical software in all channels of a hardware redundant plant protection system. While the incidence of software failures can be reduced by rigorous control of the development and by extensive testing, it cannot be eliminated altogether.

Developers of nuclear systems and the utilities have attempted to overcome this difficulty by careful documentation of the development and test processes, by using internal verification and validation teams, and by subjecting these activities to audits by representatives of the Nuclear Regulatory Commission. This approach has been only partly successful (e. g., see Appendix B) because it is highly dependent on subjective evaluations. It has always been lengthy and costly to all parties involved so that it is sometimes considered to be a deterrent to the use of otherwise desirable digital technology.

An alternative to claiming that the common software meets the single failure requirement is to employ defense in depth in the form of diverse software or functional diversity. The latter will be more expensive in procurement but have the advantage of diverse requirements (thereby permitting some relaxation of requirements verification, a very labor intensive process) and may, as an alternative, utilize diverse hardware (thereby overcoming the potential single mode failure

due to design defects). The use of functional diversity makes the evaluation much less subjective than that of equivalent implementations based on single string elements, and it should therefore lead to more certain, faster, and less costly licensing. Yet, none of the classification documents discussed above, with the possible exception of that generated by the EPRI working group, provides explicit guidance for the use of diverse software or systems, either in defining what degree of diversity is suitable, or in indicating the appropriate modification (relaxation) of safety assessments.

Much of the expense of functional diversity arises from hardware components, where diagnostic and maintenance training and spare parts inventories need to be considered in addition to the original procurement. These additional outlays can be minimized if the computer and control functions are implemented in equipment types that are already installed at the plant. Note that the above recommendation regarding functional diversity permits, but does not require, hardware diversity. Even if the same design flaw is present in the platforms serving diverse functions, it is highly unlikely that it will cause a simultaneous failure because they will be executing different programs. Similarly, there is no evidence in the current literature that the reliability of functionally diverse programs is significantly increased by use of different languages and compilers. The probability of the same language difficulty affecting all programs simultaneously is exceedingly small. The major disadvantage associated with the use of different languages is that the different compilers and tools will have to be maintained throughout the operational life.

In the discussion of the role of diversity, members of the standards working groups as well as regulatory staff have pointed out that accommodation to diversity, or exceptions from specific provisions, is possible or has in some instances been practiced in the past. However, unique accommodation or exceptions do not remove the uncertainty faced by either the developer or the user, nor do they motivate the design of systems that incorporate desirable diversity provisions. It is therefore recommended that high priority be assigned to the development of standards and/or regulatory guidelines that (a) define levels of diversity, and (b) assign safety classification or requirements consistent with the highly reduced dependence on the operation of one of the diverse implementations. This problem is partly addressed in Supplement 1 to IEC Publication 880 (45AWG-A3(Sec)47, March 1993) which states[2]:

> When diversity is required it shall be planned and documented. The balance between using one computer-based system which is of the highest quality in two different ways, and two systems of which one may be of poorer quality is of special importance and should be considered and analyzed.

The use of diversity may also be an effective means of dealing with the increasingly important classification of commercial or reused software in connection with programs developed specifically for nuclear plant protection systems. Commercial or reused software will in general

[2] A similar position has been taken by the NRC in SECY-93-087 which permits a safety related system to be backed up by a high quality non-safety system.

not meet the full verification and validation procedures applicable to developed software. To make this software acceptable it has been argued that (1) its reliability has been established by extensive general use, or (2) there is very limited interaction of the non-developed software with the safety functions. The first argument is weak because (a) general users may not report failures consistently and (b) the general use may not involve combinations of features that will be utilized in a given nuclear application. To support the second argument it is necessary to demonstrate that the lack of interaction with the safety functions holds under conditions of intermittent power and severely corrupted data that may be encountered during a seismic event or other plant emergency.

Another classification problem introduced by the use of software is the presence of many features not directly connected with the primary operation. These include initialization, sensor calibration, concurrent self-test and diagnostics, and on-demand or condition-dependent test and diagnostics. These features make the digital system more useful and dependable than the analog version, reduce dependence on personnel skills, and lower operating costs. Under present classification practices the code (and other software products) associated with these features is subject to the same provisions that apply to the operational part of the program, and this increases the cost considerably. If these non-mainline sections, which frequently comprise more than one-half of the total code, are well isolated (to be defined later), they should be assigned a lower classification because their failure is much less likely to affect plant safety than a failure in the operational part of the program, e. g., a failure in the diagnostics may cause a good component to be declared failed (leading to unnecessary maintenance), or a failed component to be declared good. The latter is the more serious condition, but it will affect the plant protection function only if there are simultaneous failures in one or more redundant parts and these are required for dealing with the particular plant condition that creates a demand. The program deficiency in detecting the failure will be noted at the next higher level system test, at which time the failed component will be replaced and the diagnostic code corrected.

The isolation postulated above must prevent the non-mainline code from interfering with the operation of the mainline code. The only known mechanisms for this interference are: (1) exceeding the allocated execution time, thus reducing the time available for execution of the mainline program, (2) usurping input/output channels, thus preventing communication with the mainline program, (3) altering the computer operating mode, and (4) writing into memory areas used by the mainline program. Highly effective measures (described in Section 4.2.5) are available to preclude the occurrence of such interference, and guidance on their use should be a part of future classification documents. Once such guidance is provided, the classification of the mainline program should apply to the verification of the isolation provisions, while verification of the non-mainline code to reduced requirements should be acceptable.

The classifications discussed in the above paragraphs imply multiple levels of safety grades. It is not advisable to declare the less critical features or components as non-safety grade (Class N by ANS 58.14 or non-1E in the IEEE standards) because of the absence of any regulatory review over these. Current standard that accommodate the multi-level classification are IEC 1226 and the EPRI V&V Handbook. In all instances cited here, a multi-level classification will relax the

verification requirements for some features or components compared to the current practice. It is therefore not likely that it will evoke the objection that it brings currently unregulated components under review which has previously been raised against multi-level classifications. Multi-level safety classifications for nuclear applications have also been advocated by the International Atomic Energy Agency [IAEA84], the Idaho National Engineering Laboratory [ADAM84], and Siemens KWU [FISC91]. Other related work includes the analysis of levels of risk and the frequency of initiating events performed by Ontario Hydro [BROW91].

It has been shown that existing standards in the nuclear field pertaining to classification do not address several issues specific to the use of programs in digital computers or processors, such as alternatives of dealing with the single failure mode requirement, the inclusion of support features (self-test, diagnostics, calibration), and the use of non-developed components. Technically desirable solutions (such as the risk based approach described in the following section) differ too widely from the current practice in the nuclear field to be adopted soon. As an interim step relaxed verification requirements may be accepted in situations where this appears technically warranted, and suggestions for these are included in later chapters. It is clearly recognized that the presence of these procedures does not imply their endorsement by any of the sponsors of this work.

*2.3 CLASSIFICATION IN OTHER STANDARDS

This heading investigates classification practices for safety critical systems or components in other areas with particular emphasis on the process industry and the military services.

2.3.1 Process Industry Classifications

Classification practices in the process industry are of significance to the nuclear power field because (1) the plant parameters being monitored are similar (though not identical), (2) there is widespread public concern about the consequences of possible failures, and (3) the equipment used for implementation of the protective functions is frequently similar or even identical. The principal standards activity in this field in the U. S. is being carried out by the Instrument Society of America (ISA) as standards project (SP) 84 which is working on standard 84.01 "Programmable Electronic Systems (PES) in Safety Applications" (draft 13 in August 1993). There is largely parallel international activity under IEC SC65 which is working on two pertinent documents, SC65A(Sec)122 "Software for Computers in the Application of Industrial Safety-Related Systems", and SC65A(Sec)123 "Functional Safety of Programmable Electronic Safety-Related Systems: Generic Aspects". The basis for classification is found in the latter draft standard and employs system integrity levels based on the tolerable probability of failure as shown in Table 2.3-1.

Table 2.3-1. Definition of System Integrity Levels

System Integrity Level	Target Failure Probability	
	Dangerous Failures/hr	Failures/Demand
4	$\geq 10^{-9}$ to $<10^{-8}$	$\geq 10^{-5}$ to $<10^{-4}$
3	$\geq 10^{-7}$ to $<10^{-6}$	$\geq 10^{-4}$ to $<10^{-3}$
2	$\geq 10^{-6}$ to $<10^{-5}$	$\geq 10^{-3}$ to $<10^{-2}$
1	$\geq 10^{-5}$ to $<10^{-4}$	$\geq 10^{-2}$ to $<10^{-1}$
0	No requirement	

The first probability column pertains to continuously operating (control) systems, while the last column is applicable to protection systems. The standard provides only the following very general guidance on the circumstances that lead to assignment of integrity levels (Clause 8.4.2.9):

- the specific hazards and the consequences
- the application sector and its accepted good practices
- the legal and safety authority regulatory requirements
- the public perception of the risks
- the risk associated with the equipment under control
- availability of accurate data upon which the hazard and risk analysis is to be based.

Annex A (Normative), *Risk and System Integrity Levels: General Concepts*, establishes a correlation between risk and the system integrity levels defined in the body of the standard. Risk is defined as a function of the probability of an event and the severity of its consequences, and the classification is essentially identical to that used in U. S. and U. K. military standards discussed below.

IEC65A(Sec)123 implements the previously discussed classifications specifically for software in safety critical systems. It establishes recommendations based on the specified system integrity level that include the following areas:

- hazards analysis

- personnel and responsibilities
- life cycle issues and documentation
- software requirements specification
- software architecture
- design and development
- maintenance
- design and coding standards
- static analysis
- dynamic analysis and testing
- programming languages

Examples of the recommendations for verification and black-box testing (a recommended validation technique) are reproduced in Figures 2.3-1 and 2.3-2. The column headings (IL0, etc.) designate the integrity levels to which the recommendations contained in the columns apply. The "Ref." column contains references to appendix clauses that briefly describe the techniques and also provide a bibliography.

The entries in the body of the table are: R = recommended, HR = highly recommended, and - = recommended neither for or against. The symbol NR is used to designate techniques that are positively not recommended.

Clause 12 : Verification

TECHNIQUE/MEASURE	Ref	IL0	IL1	IL2	IL3	IL4
1. Formal Proof	B.31	-	-	R	R	HR
2. Probabilistic Testing	B.47	-	-	R	R	HR
3. Static Analysis	D.9	R	R	HR	HR	HR
4. Dynamic Analysis and Testing	D.2	R	R	HR	HR	HR
5. Metrics	B.42	R	R	R	R	R

NOTE:
1. One or more of these techniques shall be selected to satisfy the Integrity Level being used.

DEGREE OF INDEPENDENCE	IL0	IL1	IL2	IL3	IL4
1. Independent Company	-	R	R	HR	HR
2. Independent Department	R	R	HR	HR	HR
3. Independent Persons	HR	HR	HR	R	R

Figure 2.3-1 Recommended Verification practices in IEC65A(Sec)123

D.3 Functional/Black Box Test

Referenced by Clauses 11, 13 and 14

TECHNIQUE/MEASURE	Ref	IL0	IL1	IL2	IL3	IL4
1. Test Case Execution from Cause Consequence Diagrams	B.6	NR	-	-	R	R
2. Prototyping/Animation	B.49	NR	-	-	R	R
3. Boundary Value Analysis	B.4	R	R	HR	HR	HR
4. Equivalence Classes and Input Partition Testing	B.19	R	R	HR	HR	HR
5. Process Simulation	B.48	R	R	R	R	R

Notes :

1. The analysis for the test cases is at the software system level and is based on the specification only.

2. The completeness of the simulation will depend upon the extent of the integrity level, complexity and application.

Figure 2.3-2 Recommended Black Box Testing in IEC65A(Sec)123

The U. S. draft standard, ISA SP84.01, uses the system integrity levels defined in IEC65A(Sec)123. It provides more definitions of "layers" that may be involved in achieving protection against accidents, such as:

- the process itself (which may be self-quenching, etc.)
- the process control system
- independent alarms (indicating the approach to an unsafe state)
- safety interlock system
- pressure relief valves
- emergency response procedures.

- community awareness and emergency response.

The action of the upper layers of this model should preclude the demand on the lower levels, such that the probability of demand on the last two can be shown to be extremely low. In determining the required system integrity level for a given layer, the effectiveness of the preceding layers is taken into consideration, so that even for very high consequence applications the system integrity level for the lower layers may be low (level 1 or 2). No detailed guidelines for the application of this concept are provided.

2.3.2 Classification in Military Standards

Safety classifications used in military standards have been in existence for many years and have been applied in practice, whereas the process industry standards are still in draft form and have seen very little use. Thus, with full recognition of significant differences between most military applications and those in nuclear power plants, a review of the military safety classification practices is considered pertinent. Three standards have been selected for this purpose: MIL-STD-1629, *Failure Mode, Effects and Criticality Analysis;* MIL-STD-882, *System Safety Programs;* and U. K. MOD-0056, *Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment.*

MIL-STD-1629 "Failure Modes, Effects, and Criticality Analysis" establishes consequence classes, simply referred to as "severity" (meaning severity of the consequences), based on post-event observables (deaths, injuries, major economic loss, etc.). It also defines "criticality"(This is equivalent to the term "risk" use in this report) as a function of both severity and expected frequency of occurrence. A graphic representation of the concept of criticality is shown in Figure 2.3-3 (reproduced from MIL-STD-1629). The difficulty of assessing post-event observables (frequency and severity) during the development of a system are overcome by the assignment of alpha and beta factors. Because this methodology is potentially useful in translating service oriented classifications to severity oriented ones it is explained here by means of an example. Alpha factors translate device failures to effects at the next higher (component) level. The alpha factor is similar to the decomposition of failure modes in IEEE Std. 500, e. g., failures in circuit breakers will result in the following fractions of effects at the controlled device:

not energized when commanded	0.13
not deenergized when commanded	0.09
deenergized without command	0.04
degraded operation	0.74

(this example is taken from the "Recommended" column of Table 3.1, page 106).

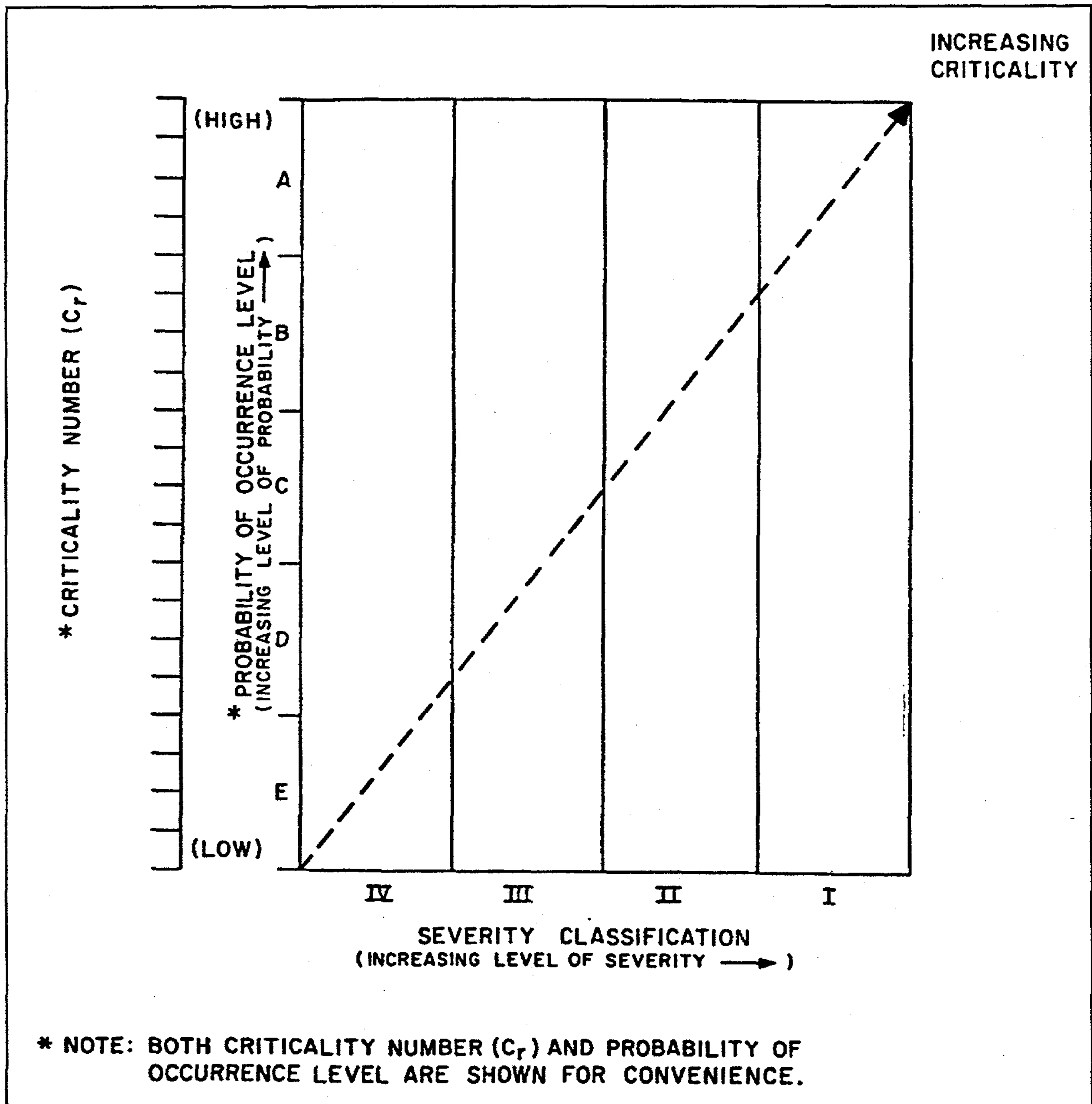


Figure 2.3-3 Risk Definition in MIL-STD-1629

Beta factors translate component failures to system failures. If 100 of the circuit breakers in the previous example are used in a nuclear power plant, and 90 control non-essential services, 6 control services that must be energized for normal operation and 4 control services that must be deenergized for normal operation, then the following beta factors will be computed:

not energized when commanded	0.06
not deenergized when commanded	0.04
deenergized without command	0.06
degraded operation	0.10

The effect of part (e. g., circuit breaker) failures on the system is computed by multiplying corresponding alpha and beta factors and adding the products where appropriate. Thus, the conditional probability (given that a circuit breaker failure occurs) of normal operation being catastrophically interfered with due to a circuit breaker failure in our example is $0.13 \times 0.06 + 0.09 \times 0.04 + 0.04 \times 0.06 = 0.0138$, and the conditional probability of an essential function being affected by degraded operation of a circuit breaker is $0.74 \times 0.1 = 0.074$. These conditional probabilities can be used to modify qualitative risk evaluations as discussed in Section 3, or be converted to unconditional probabilities by multiplying with the part (here, circuit breaker) predicted failure rates and used in quantitative assessments as discussed in Section 4.

MIL-STD-882B "System Safety Program Requirements" also establishes consequence classes there called "hazards severity". The classification is essentially identical with that of MIL-STD-1629. The frequency of occurrence is called "probability of hazard", and "risk" is used in the same sense as in the present report.

UK MOD 00-56 Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment (Interim Standard, dated April 91). The standard is based on consequences (directly expressed as multiple deaths, single death or multiple severe injuries, etc.) and probability ranges for the consequences. It establishes a risk classification consisting of four levels:

A - intolerable
B - undesirable, acceptable only when further risk reduction is impractical
C - tolerable with the endorsement of the review committee
D - tolerable with the endorsement of normal project reviews.

The assignment algorithm to these risk levels is shown in Table 2.3-2 below.

Table 2.3-2. Risk Classification in MOD-0056

Probability of occurrence	Consequence			
	Catastrophic	Critical	Marginal	Negligible
Frequent	A	A	A	B
Probable	A	A	B	C
Occasional	A	B	C	C
Remote	B	C	C	D
Improbable	C	C	D	D
Incredible	D	D	D	D

Events below the double line (to which an Incredible probability has been assigned) may in most cases be disregarded in the development of safety requirements. From this risk classification a requirement for the attainment of certain safety integrity levels is derived, designated S4 (highest) to S1. This classification is in principle similar to that of the integrity levels (IL) shown in Table 2.3-1 but the definitions shown there do not apply. The assignment depends on the number of protection provisions: for a single or principal protection system, all applications that have catastrophic or critical consequences propagate to integrity level S4 (except those below the double line), those with marginal consequences propagate to level S3, and those with negligible consequences propagate to S2. For second and subsequent protection means, level S4 corresponds to risk category A, S3 corresponds to B, S2 corresponds to C, and S1 to D. An exception to this direct correspondence are applications with marginal consequences and frequent occurrences which require only S3 although they carry a risk designation of A.

An important provision in MOD 0056 is the explicit provision for two or more components of a lower safety integrity level to be used instead of a single higher level component, provided that the failure modes are independent, and that the combining function meets the requirements of the higher level. The following example is cited in the standard:

> Thus a Safety Integrity Level S4 function to provide measurements of fuel level in a tank could be implemented by two Safety Integrity Level S3* components, provided they determined the level in diverse ways (perhaps by direct measurement of level and by monitoring flow out of the tank), in conjunction with a Safety Integrity Level S4 voting system that combined the two outputs. The * marking indicates the strict independence that shall be maintained between the components through specification, design, development, and maintenance.

2.3.3 Discussion of Classifications from Other Fields

Although the terminology differs, all of the standards reviewed here employ a multi-level, risk based, classification for establishing design requirements and evaluation criteria for safety critical equipment in general or for software controlled equipment in particular. Risk is defined as a function of probability of occurrence of an event and the severity of the consequences associated with that event. In most cases four categories of safety-relevant risk are identified, and a fifth category of events not relevant to safety is either defined or implied.

The four consequence categories shown in Table 2.3-2 (and used with slightly different terminology in all these standards) can be brought into correspondence with the four notification requirements in 10CFR50 Appendix E (see Section 2.1 of this report), but the probabilities of occurrence have in the past not been accepted as classification factors in the nuclear field. It has earlier been noted, though, that ANS 50.1 (draft 6, January 93) uses probability of occurrence (there called Plant Condition) extensively as a *de facto* classification factor. The scarcity of failure data may initially impede the use of this classification but its longer term benefits are significant. The risk based classification is rational and, particularly in the ISA SP84.01 formulation, permits trade-offs between multiple layers of protection in establishing the requirements for any one level. This may lead to considerable cost savings.

2.4 QUANTITATIVE RELIABILITY ASSESSMENT

In contrast to other types of systems, a standard quantitative metric such as reliability or availability is not directly appropriate for a protection system. It is not of much importance that the system had an availability of 0.9999 or an MTBF of 10 years if, at the time it was needed, the safety system experienced a failure. Hence, the *probability of failure on demand* (f/d) is a better indicator for specifying the requirements for a protection system, and this is indeed the metric of choice in those nuclear plant applications where quantitative requirements have been established.

The actual values or ranges of the required probability of failure on demand vary from country to country. In the United Kingdom a probability of 10^{-7} f/d for reactor trip systems is discussed in the literature [HUGH92]. Informal contacts during the course of this research with German protection system developers and regulatory bodies in Finland indicated the acceptance of higher probabilities (10^{-5} to 10^{-6} f/d) in those countries.

All of these values are beyond being capable of being verified by direct statistical methods. In practical terms they represent the operation of two independent systems, each of which has a demonstrated or verifiable probability of 10^{-2} to 10^{-4} f/d. In discussions with Siemens KWU (Germany) it was stated that the former figure is assumed for the first operational application of a new digital system (without apparent distinction as to development methodology or test results) while the latter is assumed for an established (mostly analog) back-up system with hundreds of plant-years of successful operation.

The practical implications of the quantitative approach lead to the conclusion that for the highest risk applications two functionally diverse implementations are required for the reactor shut-down system. For systems of lesser criticality, quantitative methods of classification and assessment hold promise of a better matching of safety requirements and capabilities than is possible by the prescriptive approach. Among other benefits, it will reward developers of systems with demonstrated dependability by permitting application of their products with fewer restrictions and for more demanding applications. Even for reactor trip systems the cumulative observation of the failure frequency of each system (where two or more are installed) may permit adjustment of trip points to reduce the probability of false trips or the requirement for maintenance activities. It is therefore expected that the use of quantitative criteria will permit the procurement of high dependability protection systems at lower cost than currently achievable. At present, the greatest obstacle to evaluation and application of quantitative methods for classification is the lack of creditable data. A strong recommendation for the acquisition of such data is presented in Chapter 3 of this report.

The lack of creditable software failure data is not restricted to the nuclear industry. This contrasts markedly with the dissemination of plant failure frequency and downtime data in the nuclear field, and of hardware failure data from multiple repositories, such as the Reliability Analysis Center (RAC) associated with the USAF Rome Laboratory, the Government Industry Data Evaluation Program (GIDEP), and Bellcore. The implications of this lack of data, and possible remedies, are discussed in the next chapter in connection with error classification.

2.5 CONCLUSIONS AND RECOMMENDATIONS

Although the governing provisions of 10CFR50 neither preclude nor mandate multi-level safety classifications, the predominant current practice for electrical and electronic systems (either analog or digital) is based on a single safety classification, 1E. This practice is carried over into two significant emerging standards, IEEE P-7-4.3.2 and ANS 58.14. The single safety classification is not a suitable methodology for classifying software based systems that employ functional diversity and which contain well isolated routines (as defined in Section 4.2.5) that are not directly involved in the operation of the safety critical function, such as diagnostics. Current regulatory approval of new systems therefore has to be negotiated for each individual case, introducing uncertainty that inhibits the introduction of digital equipment.

Multi-level safety classifications are widely employed in the process industries and in the military services, and a multi-level classification has now been adopted for nuclear instrumentation and control systems in IEC 1226. Multi-level classifications provide a systematic approach to reducing the verification requirements where diverse software or functional diversity are employed (e. g., as indicated by the difference between IL3 and IL1 in Figures 2.3-1 and 2), and they can also reduce the requirements for well isolated non-mainline components in critical programs. Support for acceptance of these standards is therefore recommended.

Most safety standards outside the nuclear field now employ a risk-based classification, where risk is defined as a function of probability of occurrence of an event and the severity of consequences associated with the event. These are inherently multi-level and therefore provide the benefits mentioned above. In addition, they permit experience relative to frequency of failure and severity of failure that is accumulated in use to be factored into the safety requirements. This motivates the design and deployment of highly reliable systems, and also promotes objective trade-offs of diversity vs. quality, and of high quality at one layer vs. at another layer of plant protection. Appendix A provides guidelines for a risk based classification of digital equipment in protection systems for nuclear reactors.

The major impediment to the use of this classification is the lack of pertinent failure data. It is urgently recommended that failure data on digital systems employed for safety or control functions in nuclear plants be collected. These can be analyzed both statistically and qualitatively (determining failure mechanisms and failure consequences) to support establishment of requirements and to promote the use of design, test and maintenance practices that address the root causes of failure in nuclear power plants.

SECTION 3 - ERROR CLASSIFICATIONS

3.1 OVERVIEW

The technical requirements for this chapter are identified in paragraph 4.1.3 of the Statement of Work, the introductory clause of which states:

> Develop error classification guidelines and a methodology to analyze errors and improve the development process.

Other significant requirements are to address errors originating in hardware, firmware, software, and from hazards in the operating environment, and to cover the entire lifecycle. The classification should permit the evaluation of error avoidance capabilities of various software development methodologies and it should investigate the relationship between errors, safety and cost.

3.1.1 Motivation for Error Classification

In the specific context of high integrity digital systems the motivation for error classification arises from a combination of the following needs:

- to identify

  lifecycle phases and activities that are likely to be a source of errors

  major system partitions, such as hardware, software, human interface, and components that are either the cause or the target of the errors

  operational states and activities, such as operating points or load shedding schedules, that minimize the impact of errors

- to evaluate overall development strategies and specific design, maintenance and test techniques that minimize errors, and to characterize the errors that are likely not to be eliminated

- to support the development of fault tolerance techniques that are particularly effective in dealing with the remaining error types

- to establish objectives for verification and validation, and to investigate methodologies for making these processes more effective

In the more general context of software development, error classifications have been used to evaluate hypotheses about causes of software failures, means for avoiding or tolerating them, and

policies that may reduce the cost of software development. Examples of existing classifications motivated by these general objectives are discussed in Section 3.2. The requirements for error classifications for high integrity systems arising from the present effort and the resulting recommendations are presented in Section 3.3, and applications of the classification are discussed in Section 3.4. Conclusions of the error classification task are summarized in Section 3.5. The subsection immediately following is intended to familiarize the reader with the general concepts of the failure process and its nomenclature.

3.1.2 Nomenclature and General Concepts

Comprehensive definitions of the terms *error, failure,* and *fault* are essential to the understanding of the error classification, and therefore the definitions and the associated concepts are discussed here in more detail than in the Glossary. The source for the following definitions is the IEEE Dictionary of Electrical and Electronic Terms, 1984 edition. Where separate definitions for specialized fields were listed the entry applicable to software was selected.

> **error**: (1) A discrepancy between a computed, observed, or measured value or condition and the true, specified or theoretically correct value or condition. (2) Human action which results in software containing a fault. Examples include omission or misinterpretation of user requirements in a software specification, incorrect translation or omission of a requirement in the design specification. This is not a preferred usage.

In this report only definition (1) is used. Where there is a need to refer to the human action that resulted in the software containing a fault the term *mistake* is utilized, which is defined in the IEEE Dictionary as "A human action that produces an unintended result".

> **failure:** (A) The termination of the ability of a functional unit to perform its required function. (B) The inability of a system or system component to perform a required function within specified limits. (C) A departure of program operation from program requirements.

In this report the primary emphasis is on definition (A) because it is the most inclusive one. Note, however, that the three definitions are not inconsistent with each other.

> **fault:** (1) An accidental condition that causes a functional unit to fail to perform its required function. (2) A manifestation of an error in software. A fault, if encountered, may cause a failure. Synonym: bug.

In this report definition (1) is used. If "a mistake" is substituted for "an error" in (2) both definitions will be applicable. The synonym is valid for both definitions.

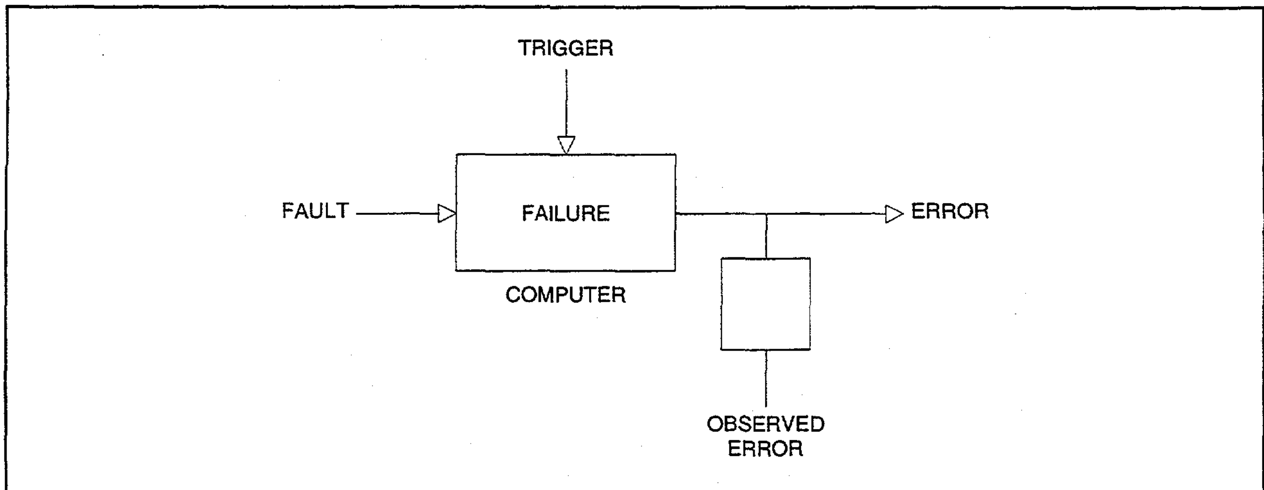


Figure 3.1-1 Generic Failure Model

The general failure concept on which the terminology is based is shown in Figure 3.1-1. The term *failure* represents an event, usually of very short duration, during which the computer changes from a valid state in which it furnishes specified outputs to an invalid state due to which current and/or future outputs will differ from required ones.

The cause of the failure is usually the coincidence of two conditions, the *fault* and the *trigger*. The latter term is used in its common meaning of an initiating condition or action.[3] For software failures the fault was present prior to the failure (one or more faulty statements or declarations) but it caused a failure only due to the presence of a specific data set or computer state. In hardware failures the fault usually (but not always) develops during operation and it may cause the failure immediately (no trigger required) or it may become manifest only under an external stimulus (shock, temperature change, or voltage spike).

The failure takes place in a computer component (e. g., a register or an output port) and usually it cannot be directly observed. The manifestation of the failure is an *error*, an output state that deviates from the specified (or desired) one. Not all errors are observed, and in some situations it may be necessary to distinguish between observed and latent errors. An example is a delay between input and output that is slightly greater than the specified value. This may go unnoticed until another event that should always follow the specified output occurs coincident with it or even ahead of it.

The concepts discussed above can be applied at several layers of a hierarchy the top of which is "the system" (the largest entity for which the narrator is responsible), and the bottom of which is comprised of parts (for hardware) and statements (for software). There are no general rules

[3] Thus, the trigger activates a previously latent fault, resulting in a failure that produces an error.

for the number of intermediate layers but there is usually at least one for software and two for hardware. The software case is illustrated in Figure 3.1-2 where the lowest level is designated as the *logic level*, the intermediate one as the *information level,* and the top one as the *system level* [AVIZ82]. In this example the fault is represented by the lack of a "buffers full" exception handler in the program. The trigger is a "buffers full" condition, and the error at the logic level is an incorrect alteration of memory. At the information level the altered memory represents the fault, and the propagation of this condition to "faulty data" constitutes the error. The faulty data are the fault at the system level and cause the "lost message" error. In this example the altered memory is not detected (a latent error) but the faulty data state and the lost message are detected (observed errors). For hardware the lowest level is the *physical level* (representing parts failures), and this is followed by the logic level where the parts failure results in an incorrect binary pattern.

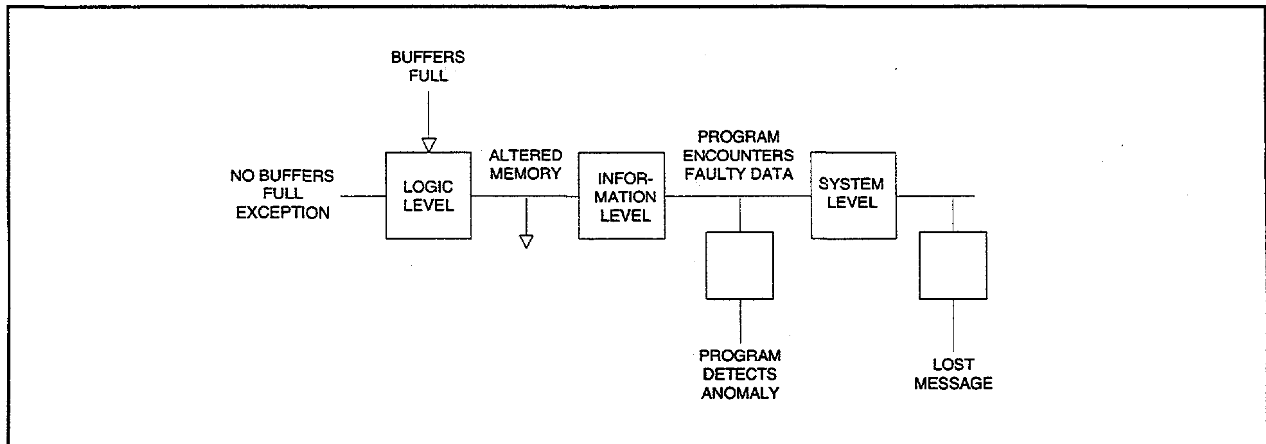


Figure 3.1-2 Propagation of Failure Effects

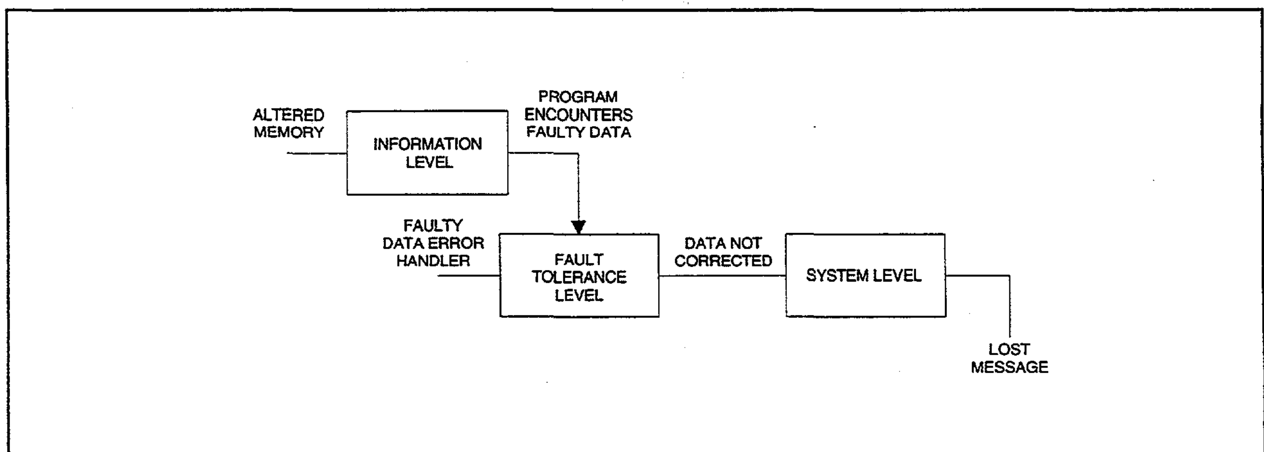


Figure 3.1-3 Failure Model for Fault Tolerance Provisions

In this example only a single trigger at the lowest level is indicated, but it is possible for multiple triggers to be present. An example is an information level failure (with an original trigger at the logic level) that is displayed in the control room and should normally be responded to without causing a system level anomaly. Because a shift change occurs at that instant the indication goes unnoticed and results in a system level failure. The shift change may be regarded as the second trigger.

Fault tolerance provisions can be represented by an additional intermediate level as shown in Figure 3.1-3. In this case the fault tolerance consists of an exception handler that receives inputs from the information level and requests retries or activates alternate routines when it detects errors. As shown in the figure the data error handler is faulty, and this permits the error to be propagated to the system level. Note that the error at the information level constitutes the trigger at the fault tolerance level whereas in Figure 3.1-2 it was the fault at the system level.

The nomenclature and concepts discussed here are essential to the consistent description and use of error classifications for high integrity systems. The levels at which the classification applies cannot usually be prescribed in advance, but in discussing classifications it must clearly be stated at which level(s) they are applicable.

*3.2 REVIEW OF PRIOR WORK

3.2.1 Administrative Error Classifications

The earliest error classifications were initiated to serve administrative needs, such as to assign responsibility for corrective action and to identify trends in the operational capabilities of the system. A typical administrative classification is: hardware, software, skinware (direct human actions), and environment (circumstances not controlled by the administrator, such as power, weather, earthquakes). In large systems there may be a need for a finer structure within some of the primary classifications, e. g., software may be partitioned into sensor processing, displays, control algorithms, and interface routines.

The administrative classifications are useful for high integrity systems and are frequently encountered in practice. The information contained in them is sometimes used inappropriately. It is reported that management of an airline reservation system allocated maintenance resources so as to keep failures due to hardware, software, and other causes at approximately the same level [GIFF84]. SoHaR has encountered similar tendencies in several large government systems [HECH86].

3.2.2 U. S. Air Force Software Fault Classifications

Early in the 1970 decade the USAF Electronic Systems Division and the Information Processing Branch of the Rome Air Development Center (now Rome Laboratory) sponsored several broad based investigations in software reliability, and two of these produced comprehensive error classification methodologies [AMOR73] and [THAY76], which are in the following text referred to as MITRE and TRW schemes, respectively.

These pioneering investigations have had a significant impact on subsequent research and are therefore discussed in some detail here. Both studies resulted in deliberately "open ended" classifications which left room for the addition of other classification criteria as well as additional categories within each of the defined criteria. These decisions were taken because of the rapid changes that were foreseen (and indeed occurred) within the computer and software fields. Multi-processors and distributed computing were advanced research projects at the time and are today the practical environment in which most serious computing takes place. The resulting hardware architectures are very sensitive to software timing and this has caused the classification of response delays to be a major consideration in the establishment or evaluation of any error classification scheme. Similar changes must be anticipated in the future and a classification for high integrity systems should therefore be open ended.

The top level of the MITRE classification consists of five categories:

Where - the context in which the error appeared

What - the manifestations of the error

How - identification of the specific code or data involved

When - the development stage at which the error occurred

Why - presenting the reasons for the error

Classification details for the first two categories are shown in Tables 3.2-1 and 2. It is seen that there is extensive detail at the lower levels, and that it may take much training of the classifiers to use this scheme effectively (some of the headings permit several interpretations). In comparing the two tables it will also be noted that there is overlap between the two tables in the software category. Moreover, the headings are not at consistent hierarchy levels. There are equivalent overlaps and inconsistencies in break-outs of the other top level categories. The top structure of the classification is appealing but the problems at the lower levels indicate lack of independence of the categories at the top. Notably absent from this classification scheme is a severity or criticality category. The report does not indicate that data were ever collected to populate the classification scheme. It is concluded that the MITRE classification scheme is not suited for high integrity systems because of difficulties of data collection, a partially inconsistent structure and the absence of consequence considerations.

Table 3.2-1 Digital System Error Classification for Where?

- People
  - Structure
    - Technical
      - Name
      - Qualifications
      - Responsibility
    - Administrative
      - Name
      - Qualifications
      - Responsibility
  - Procedures
    - Operating Procedures
    - Coding and Checkout Proc.
    - Documentation Standards
- Hardware
  - Computer
  - Communications
  - Support
- Software
  - Operating System
  - Support
    - Language Processor
    - Loader
    - Linkage Editor
    - Utility
  - Application
    - Sizes
      - Number of Lines
      - Number of Statements
    - Adjacent Modules
      - Names
      - Relationships
        - Superior
        - Coequal

Table 3.2-2 Digital System Error Classification for What?

- Software
  - Operating System
  - Language Processor
  - Linkage Editor
  - Loader
  - Utility
  - Application
- Functions
  - Name
  - Procedure Input
  - Procedure Output
  - Resource Use
- Resources
  - Name
  - Used Too Long
  - Used Too Much
  - Not There
  - Misused
- Scope
  - Statement
  - Internal Block
  - External Procedure
  - Application
  - System

The TRW approach is based on extensive experience in collecting and analyzing software problem reports and is a successor to four prior classification schemes. It has 12 major categories (reduced to nine in some presentations) and 79 detailed categories. The top classification is shown in Table 3.2-3.

Table 3.2-3. Top Software Error Classification of TRW Methodology

A - Computational Errors
B - Logic Errors
C - Data Input Errors
D - Data Handling Errors
E - Data Output Errors
F - Interface Errors
G - Data Definition Errors
H - Database Errors
I - Operation Errors (Operating system, hardware, operator)
J - Other (Timing, memory limitations, compilation)
K - Documentation
X - Rejected Reports (not an error)

A typical lower level classification, this one for logic errors, is shown in Table 3.2-4.

Table 3.2-4. TRW Detail Classification of Logic Errors

B-000 Errors not falling into the following classifications
B-100 Incorrect operand in logical expressions
B-200 Logic activities out of sequence
B-300 Wrong variable being checked
B-400 Missing logic or condition tests
B-500 Too many or too few statements in loop
B-600 Incorrect loop iteration (includes endless loops)
B-700 Duplicate logic

It can be seen that the TRW concentrated on specific fault symptoms, and that it called attention to the significant causes of software errors as they existed in the 1970 - 75 time period. The reference includes samples of data collection into these (or related) categories from four major projects, with several represented by multiple versions or increments. Figure 3.2-1 is a sample of data reporting by use of this scheme. The strong point of this classification is therefore that it is practical. In its original or modified form this classification is widely used in the aerospace field. One of the authors of this report has used the classification of Table 3.2-3 on software problem data from a NASA spacecraft program and from a commercial flight control system and experienced only moderate difficulties in fitting prose descriptions by personnel not familiar with the TRW scheme into the top categories.

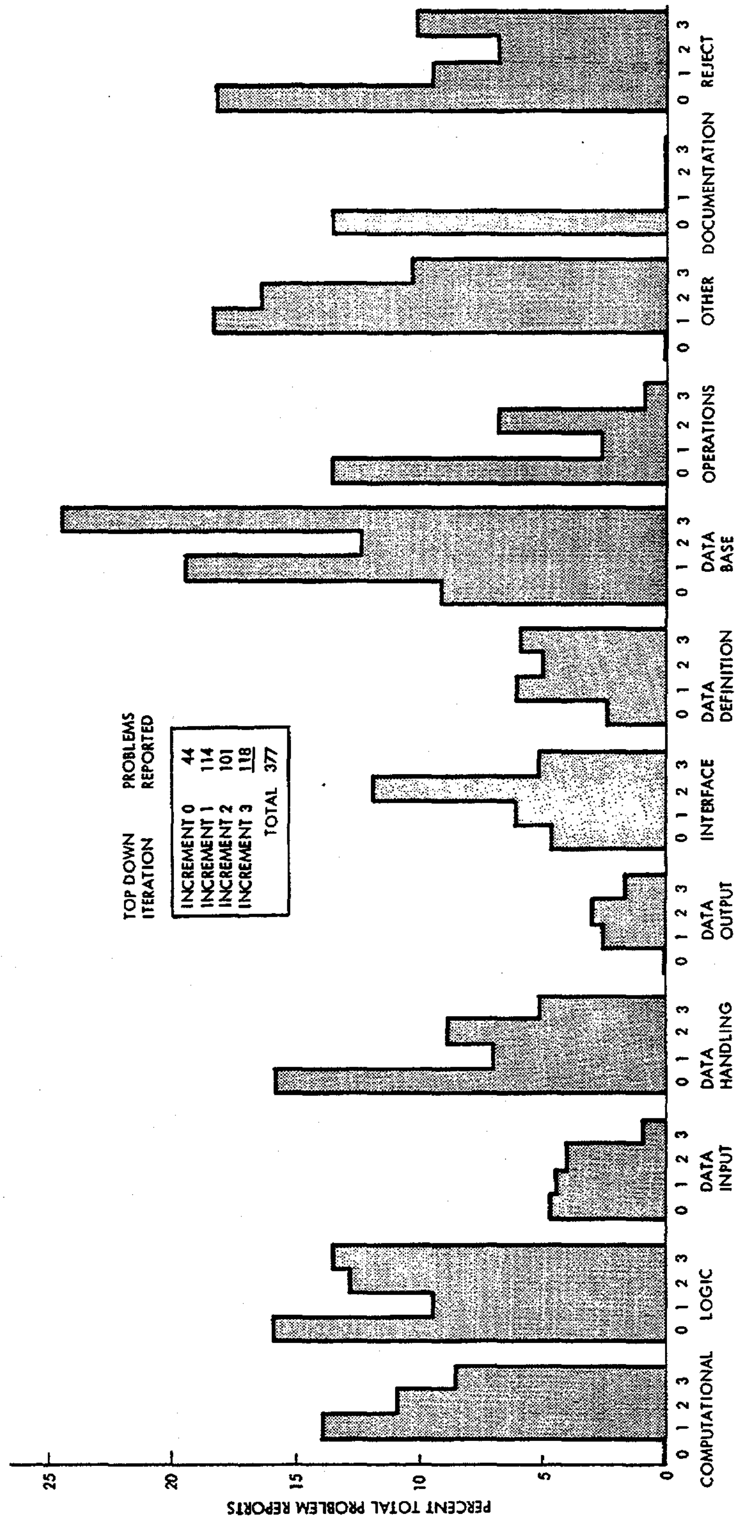


Figure 3.2-1 Use of TRW Classification

The TRW report acknowledges the need for severity (consequences of failure) classification but does not offer any recommendations or examples for this. The classification does permit recording of failures resulting from hardware faults or external causes but it is primarily software oriented and needs considerable expansion to serve as a general error classification scheme. Within its scope it is a useful scheme, but because of the lack of failure consequence considerations it is not recommended for application to high integrity systems.

3.2.3 Classifications in Connection with Specific Cause Hypotheses

There have been many attempts to find causes of failure by associating software attributes with fault density, failure probability, and effort required to correct faults once they were known to exist. Some of the publications resulting from these efforts include fairly detailed classifications of the attributes by which causes were to be established. The following are samples that were selected to show the range of possible classifications that may conceivably be of interest to high integrity systems.

Schneidewind and Hoffman conducted an experiment on four small software programs that totaled about 2,000 lines of Algol W code [SCHN75]. All were designed and coded by the same person in an effort that comprised about 300 hours, including testing and debugging. The top level classification involved the phase of activity in which the fault was introduced, and five categories were established for this:

1. Design Errors (missing cases or steps, data handling)
2. Coding Errors (missing data declarations, missing delimiter)
3. Clerical Errors (manual error, mental error, procedural error)
4. Debugging Errors (inappropriate tool, insufficient test data)
5. Testing Errors (inadequate test cases, misinterpretation of test results)

There were 17 detailed categories for design errors and 31 for coding errors. The other top level topics had five or six detailed categories. There is considerable overlap between categories. This is very obvious in comparison of 4 and 5 above, where it will in most cases be difficult to distinguish between insufficient test data and inadequate test cases. Note also that the first two categories deal with faults whereas the final three deal with mistakes.

Knowing in what phase faults are introduced is very important for project management, for deciding on the review level for each phase, and for selecting effective software engineering tools. Unfortunately, the assignment of responsibility to a phase is very subjective. Structural and conceptual deficiencies in the program can be attributed to requirements, design and test (proper test case selection should have found them). Coding problems are frequently difficult to separate from design weaknesses and can always be attributed to insufficient test. The authors of this report have attempted to identify the phase from error reports in a number of projects and have found it very difficult to avoid ambiguities.

The phase in which the fault is detected can be determined objectively, and this can sometimes provide insights into the phase where it may have been introduced. It cannot have been introduced later than the phase during which it was detected. This sounds trivial, but for problems discovered in a design review it resolves the design vs. coding ambiguity. Also, if in a given project 50% of logic faults involving missing conditions are detected during a design review, 25% in code inspection, and 25% in test, it is a fair presumption that most of those discovered during the latter two phases were introduced in (or prior to) design and had escaped earlier detection.

The phase in which the fault is detected is used by Glass to assess the cost for fixing it [GLAS81]. He coins the term *persistent error* for errors that are not detected until the maintenance phase and that are therefore the costliest ones to correct. To characterize the underlying faults (from the text in problem reports) he uses the classification shown in Table 3.2-5. The listing is in order of decreasing frequency of occurrence.

Table 3.2-5. Categories of Persistent Software Errors

1. Omitted logic (existing code is too simple)
2. Failure to reset data
3. Regression error (fault introduced during maintenance)
4. Documentation error (software is correct)
5. Requirements inadequate
6. Patch in error
7. Commentary in error
8. IF statement too simple
9. Referenced wrong data variable
10. Data alignment error (leftmost vs. rightmost bit)
11. Timing error causes data loss
12. Failure to initialize data
13. Others

These categories are not mutually exclusive, e. g., an error can be both a regression error and fall into one of the other categories, and one of the sample problem reports cited is characterized as omitted logic (1) and IF statement too simple (8). The author solves this by permitting a given report to be assigned to multiple categories. This is not considered desirable for high integrity systems because (a) a casual reader summing the numbers in the categories may get a misleading impression of the quality of the software, and (b) broad categories will automatically register a higher population than specific ones. This latter problem is present in all classification schemes but it is particularly prominent if multiple entries are permitted and the categories vary widely in specificity. It is concluded that this classification is not desirable for high integrity applications.

Another classification within the operations and maintenance phase is reported for a French telephone switching system [KANO87]. At the top it considers three mutually orthogonal factors

(also referred to as axes): the function of the software, the operations phase during which the error was detected, and the severity of the error. Primarily because it includes severity (and is one of the very few classification schemes to do so) it is pertinent to high integrity systems and is therefore described in more detail. The classification within each axis is shown in Table 3.2-6.

Table 3.2-6. Classification for Software Errors in Telephone System

Software Functions	A. Telephony (all switching routines) B. Support operations (including initialization) C. Hardware fault tolerance (detection and recovery) D. Operating system
Operations Phase:	1. User initiated operations 2. Operator commands 3. Combinations and hardware malfunctions
Severity:	a. Global unavailability b. Partial unavailability c. Loss of one unit d. Failure to execute command

The comparatively few and non-overlapping categories along each axis permit the collection of creditable statistics (not likely to be influenced by exceptional events or selection of extremely broad or narrow categories) even with a modest error population, here comprising a total of 58 failures and about 140 fault (correction) reports. Examples of analyses presented in the paper are shown in Tables 3.2-7 and 8. The latter table has no row for the operating system because it did not cause failures leading to global unavailability.

Table 3.2-7. Distribution of Failures for each Axis

Function	Phase	Severity
A. 29%	1. 31%	a. 13%
B. 26%	2. 25%	b. 24%
C. 30%	3. 44%	c. 30%
D. 15%		d. 33%

Table 3.2-8. Contributions to Total Unavailability

Software Function	Failure Percentage Causing Total Unavailability	Percent Contrib. to Total Unavailability
Telephony	11	29
Support	9	22
Fault Tolerance	20	49

The classification shows the importance of software in support of fault tolerance for failures that caused total unavailability. The fault tolerance function has the largest percentage of failures that lead to total unavailability, and it accounts for almost one-half of all failures in the most severe category.

The desirable features of this scheme are the mutual exclusivity of categories both at the top and at the lower level, and the small number of categories at each level. The paper draws several significant conclusions from the data of which the one shown in Table 3.2-8 is but one example.

3.2.4 Classifications in Connection with Fault Tolerance

The effective design of fault tolerance provisions requires knowledge about the errors that are to be prevented. Some classification is implicit in every fault tolerance scheme, e. g., most don't protect against persistent design defects or against sabotage, in effect classifying these causes out of the realm of covered failures. To make these assumptions more visible, Avizienis proposed a comprehensive classification scheme, shown in Table 3.2-9 [AVIZ87].

Table 3.2-9. Error Classification for Fault Tolerance

1.	By origin:	Physical vs. man-made
2.	By activity:	Dormant vs. active
3.	By duration:	Transient vs. permanent
4.	By extent:	Local vs. distributed
5.	By value:	Fixed vs. variable
6.	By consistency	Time vs. value
7.	By count:	Single vs. multiple
8.	By time:	Coincident vs. separated
9.	By cause:	Independent vs. related
10.	By intent:	Accidental vs. deliberate

The classification can serve as a checklist to determine what factors should be considered in a given application. It has some obvious deficiencies, particularly that severity and function served are not addressed (the extent classification conveys a crude notion of severity). It is not known whether this classification has ever been populated, and what the success has been in capturing the required data.

3.2.5 Classifications from Recent SoHaR Projects

Fault and Failure Classification for Air Traffic Control

In connection with its reliability support activities for the Federal Aviation Administration, SoHaR has developed a number of error classification schemes that are described below [HECH92]. The data to which these apply come from a very large distributed computing development where well over a thousand reports are generated each year. The large data volume warrants finer (more detailed) categories than may be appropriate for the high integrity systems that are the subject of this report. For this reason, and also to protect proprietary information of other contractors, some editing has therefore been necessary. The categories shown in Tables 3.2-10 and 11 are appropriate for a central database. For data from individual installations a further joining of detailed categories will be desirable.

A significant feature of this classification is the separation into two primary formats: faults and failures. Faults can always be associated with a location; failures can always be associated with a time. The separate treatment of faults and failures was also adopted in the classification of telephone switching problems represented in Tables 3.2-7 and 8. This approach has the following practical advantages:

(a) faults found during reviews or other activities that are not associated with operation of the equipment can be entered into the fault database without requiring generation of a pseudo-failure report

(b) the failure report is typically generated by a test or operational organization which is concerned with both hardware and software failures; the report format can be specifically tailored to their responsibilities

(c) faults are identified and corrected by specialist organizations who are able to determine whether a given fault has been reported previously and thus avoid double counting of faults (and can assign multiple failures to one fault where necessary).

Further analysis of summarized fault data is of interest to the component specialists while further analysis of failure data is primarily of interest to project management, reliability, and test functions.

Table 3.2-10. Fault Classification

Attribute	Explanation	Subcategories and Examples
Cause	Phenomenological cause of fault	*Hardware* "Random" fault (hardware materials properties meet specifications) Part design or application fault Component manufacturing fault Assembly fault Installation fault (including cabling, etc.) Switch settings, initialization Physical/electrical environment Test/inspection/review procedures
		Software Function Problem Interface Problem Assignment Problem Timing/Serialization Problem Documentation Problem Algorithm Problem Test/inspection/review procedures
Component	Component in which fault was *resident.*	Component taxonomy defined by specification tree
Identification stage	Development stage at which fault was identified	Requirements Top level design Detailed design Coding or manufacture Integration & Test System Test Configuration Management Site installation Operation
Isolation	Means by which fault was isolated. Includes identification of specific traces, tools, and techniques	Software failure analysis (manual or specific tools, techniques, and traces) Hardware failure analysis (specific tools, techniques) Network analyses
Recurrences	How often fault seen before isolated and fixed	Numeric value

The top level cause classification is restricted to hardware and software because this is a development project in which operator mistakes and environmental causes are not considered significant. For an operational classification the classification should be extended to include these two potential causes of failures. It is intended that the subcategories listed in the last column be exclusive. Classification guides enforce this property. A particular concern is the differentiation between test procedures and other categories. The cause will be classified as test if one of the following is present: (a) existing procedure was not carried out correctly, (b) existing procedure is unclear, or (c) existing procedure does not comply with a higher level document. In all other cases the cause will be classified as one of the other categories. This does not prevent changes of the test procedures to support more effective for occurrences of the observed type.

In the fault classification the component classification relates exclusively to the component in which the fault was found. The failure classification has an entry for "Affected component" which may or may not be identical to that in the fault classification. As previously discussed the identification stage acts as a surrogate (admittedly weak) for the stage at which the fault was introduced. A primary use is to determine whether the fault could have been detected at an earlier stage, and to improve the software development and V&V processes. The related studies should initially concentrate on the late stages (installation and operation) and gradually work towards the earlier stages. A more detailed discussion of this problem is presented below in connection with the NASA space shuttle avionics software. In a predecessor of the classification shown in Table 3.2-10 there was a field for "Fault Introduced Stage". It was found that fewer than one-tenth of all reports permitted a clear determination of that item, and that statistics based on those reports that could be classified would probably be biased because the uncertain cases included many more that might fall into the requirements area than the classifiable ones. Hence this field is currently not used.

The "Isolation" and "Recurrence" fields are also intended to promote earlier detection of faults. This can clearly be accomplished by focusing on faults that had manifested themselves several times before they were identified, and by determining which isolation means are effective in the early development stages.

The failure classification shown in Table 3.2-11 is limited to occurrences in which an error was detected during execution, which may be in test (including testing during maintenance) or operation. Because the classification shown in the table concerns a project under development it is understood that the failures are observed during test. In the general case a field that identifies the operation in progress at the time of failure is desirable.

Failures are events that are associated with a time of occurrence. The significance of that time to analysis is (a) to establish a sequence of events, e. g., to determine that failure of type X occurred one minute prior to a failure of type Y, (b) to associate groups of events with time in the project schedule, e. g., there was a significant increase in failures after a requirements change, (c) to support modeling of the reliability growth, e. g., to determine how the failure rate decreases after a certain number of faults have been removed. The latter type of analysis usually requires that data on execution time be available.

Duration of the service interruption is one indication of the severity of the failure. In this classification the transient vs. permanent aspect is captured in the same field, but a separate field can also be assigned for this. Transient failures are those for which the system restores the service without operator intervention. For high dependability applications the occurrence of any transient failures that are not repeatable is a very dangerous condition (unless the failures have little effect on the service) and a separate category is therefore established for these. Permanent failures require operator intervention to restore the service, and they are recorded with a time parameter that indicates how long it took to resume operation.

The extent of the failure is another indication of severity. This classification was generated for a distributed computing environment in which a wide spectrum of affected resources is possible. A typical high dependability application does not at present provide that many options. The following field, intensity, can be used to record whether a single channel, multiple channels, or all channels were affected. Criticality is used here to describe the actual or potential operational impact of the failure. In a typical high dependability application this may involve unnecessary shut-downs or power reductions, exceeding temperature or pressure limits for portions of the plant, or excessive radiation release.

The affected component may be different from the component in which the fault resided (see Table 3.2-10). The classification of detection mechanisms is significant where it is desired to localize failures or to minimize the time between first symptom and response. Detection mechanisms of broad scope, such as interval timers, are easy to implement but more specific detectors provide faster response and tighter containment. The classifications of recovery and restoration mechanisms are valuable for studies of the efficiency of the fault tolerance provisions, e. g., where redesign is under consideration or where experience on a given installation is being used to support a new design. From the regulatory point of view the effect of these mechanisms is captured in the duration field.

In addition to their value for the high level management of software safety and reliability, Tables 3.2-10 and -11 contain the essential information for correction of the cause of each failure.

Table 3.2-11. Failure Classification

Attribute	Explanation	Examples
Failure Mode	Manifestation of failure or error	Crash; incorrect result; late response; no response
Time	Date and time of occurrence	Calendar date and wall clock time; can also be used to record project phase and elapsed execution time
Duration	How long the service is interrupted	Transient non-repeatable (No Trouble Found); Transient repeatable; Intermittent; Permanent (time)
Extent	Impact of the failure or error	Confined to site of origination; Affects single program; Affects all tasks on processor; Affects multiple processors
Intensity	No.of simultaneous occur	Numeric value
Criticality	Operational effect of failure or error	Application dependent classification; ranging from complete loss of site to no impact
Affected component	The site of the failure	Identification of specific hardware or software components affected by the failure or error
Detection Mechanism	Means by which failure or error is detected	Interval timer, parity or cyclic redundancy check, range/type check, explicit signal, indication, exception, error message, reasonableness check, operator monitoring, none
Recovery mechanism	Means by which failure or error is recovered given a detection has occurred	Fault masking (e.g., data encoding which corrects results transparently); Forward recovery (substitution of default or previous value and continuing); Rollback and retry in same processor; Rollback and retry on alternate processor (transition address space); Controller or operator intervention None
Restoration mechanism	Means by which affected site is restored after conclusion of incident	None necessary Reinitialization of affected component Processor warm restart Processor cold restart Repair and recertification

An example of the use of these classifications for analysis is shown in Table 3.2-12, where rows correspond to the categories of the first field in Table 3.2-10 and columns to the categories of the first fields in Table 3.2-11.

Table 3.2-12. Failure Mode Classes for Hardware and Software

Top Level		Failure Mode				Invalid Report	Total
		Crash	No resp.	Incorr.	Late		
Software	No.	370	237	495	25	70	1197
	%	31	20	41	2	6	100
Hardware	No.	50	20	19	1	8	98
	%	51	21	19	1	8	100
Total	No.	420	257	514	26	78	1295
	%	32	20	40	2	6	100

That the preponderance of failures in this environment were due to software is not surprising because hardware was mature while the software was under development. As previously mentioned, errors due to personnel actions were omitted from this classification by direction of the sponsor. A significant finding is the large fraction of incorrect responses[4]. It is sometimes argued that the vast majority of failures will cause gross deviations from normal program behavior which can be detected by interval timers or operating system checks. In this sample the fraction of incorrect response failures is so large that detection based on gross deviations is not sufficient. A similar observation regarding the importance of incorrect response failures is contained in an assessment by DeMillo in the report of the Independent Fault Tolerance Analysis Team for the Voice Switching and Command System [VIFT93]. The fraction of no response failures may also be a problem in some applications where special monitoring for this condition may be required.

[4] The root cause of this may have been a very compressed design schedule for complex real-time software.

3.2.6 NASA Space Shuttle Avionics Failure Classification

Another significant classification task recently performed by SoHaR involves failures during the formal test of NASA space shuttle avionics software. The classification concentrated on the severity of the consequences of the failures and on the conditions that initiated the failure. It was found that the majority of failures, and particularly in the highest severity categories, occurred under input conditions that contained at least one rare event (RE), a condition not likely to be encountered in routine operation. Examples of rare events in that environment include loss of main engine thrust, very unusual or unauthorized crew procedures, and computer hardware failures.

Table 3.2-13. Classification of NASA Space Shuttle Avionics Software Failures

Severity	No. Reports Analyzed (RA)	No.of Rare Reports (RR)	No.of Rare Events (RE)	Ratios		
				RR/RA	RE/RA	RE/RR
1	29	28	49	0.97	1.69	1.75
1N	41	33	71	0.80	1.83	2.15
2	19	12	23	0.63	1.32	1.92
2N	14	11	21	0.79	1.57	1.91
3	100	59	100	0.59	1.37	1.69
4	136	63	92	0.46	0.88	1.46
5	62	25	42	0.40	0.63	1.68
All	385	231	398	0.60	1.23	1.72

When at least one RE was responsible for the failure the corresponding failure report was classified as a rare event report (RR). The data were collected during the acceptance test for release 8B of the program, the first flight program immediately after the Challenger accident. The program had undergone intensive test prior to the period reported on here. NASA classifies the consequences of failure (severity) on a scale of 1 to 5, where 1 represents safety critical and 2 mission critical failures with higher numbers indicating successively less mission impact. During most of this period test failures in the first two categories were analyzed and corrected even when the events leading to the failure were outside the contractual requirements (particularly more severe environments or equipment failures than the software was intended to handle); these categories were designated as 1N and 2N respectively. Results of our analysis are shown in Table 3.2-13.

Rare events were clearly the leading cause of failures among the most severe failure categories (1 - 2N) and were an important cause among all reports in this population. The number of rare events per report involving rare events (RE/RR, the entries in the last column) remains relatively constant for all severity classes around the average of 1.72. This indicates that inability to handle more than one RE at a time is really at the root of the problem. At this point it is appropriate for each of us to ask ourselves "How often have we traced a thread involving more than one rare event?" and "How often have we concentrated on test cases that involved more than one rare event at a time?" The thoroughness of final testing in the shuttle program surfaced these weaknesses which probably would have been detected only after they caused operational failures in many other situations.

Classification by the number of rare events in the conditions (usually the trigger) that caused the failure has several interesting consequences.

(a) It can focus the V&V activities; these should specifically look for consideration of multiple REs in the requirements and include multiple RE test cases as discussed in Sections 7.2 and 7.3.

(b) Development test activities must consider multiple REs; path testing offers a systematic way of covering multiple REs but is not practical for a large program. Segregation of tasks that are essential for the same operation of the plant into small segments or into an object-oriented form will make path testing feasible and permit an objective evaluation of test coverage.

(c) The instrumentation required to determine path coverage will be compatible with statistical (random) test case generation; review of the coverage obtained by various randomization schemes can be used to achieve high coverage with a limited test budget.

3.3 REQUIREMENTS AND RECOMMENDATIONS FOR ERROR CLASSIFICATION

3.3.1 General Requirements

This section describes requirements for and implementation of an error classification methodology for high integrity systems. The emphasis is on the purposes of the classification, the required data, and their logical ordering. There is no intention to specify a specific data structure or database management system. Nevertheless, it is very convenient to adopt the terminology of a generic database manager as will be explained now. A classification scheme may be thought of as a table or as a *file*. The columns of the table are the classification topics and are referred to as *fields* and the rows as *records*. In most instances we have suggested the allowable entries into the fields, and these are referred to as *categories*. Where a given field is subject to hierarchical decomposition, its categories become fields at the next lower level. As an example, the origin field of a fault classification file allows categories of hardware, software, skinware, and environment. At the next level the hardware will be broken down into the specific component

that was faulty. At this point "hardware" is a field, and the component names represent the categories. Restricting the entries to defined categories makes for a more focused analysis than is possible with the entry of unconstrained text or numeric values.

Several general recommendations have become apparent from the review of prior efforts in classification and these are summarized below.

(a) The classification framework must be open, including at the top level, to accommodate changes in the computing environment, in fault identification techniques, and in recognition of new failure effects. In general there should be an "other" category for each field. In the following it is assumed that there will be such a category as well as the possibility of adding remarks without specifically listing these.

(b) It should be a goal that the categories within each field are mutually exclusive. Where this cannot be achieved directly by designation of the categories, one of the following shall be provided: (i) a classification guide that directs the assignment to a specific category wherever there is a possibility that more than one category may apply, or (ii) the establishment of one or more categories for cases that could be assigned to several of the primary categories. As an example consider a failure that caused a primary service output to be slightly late and completely disabled the generation of a maintenance report. A classification guide may direct that where a primary service and a maintenance service are affected the category applicable to the primary service be assigned. Alternatively, additional categories may be established either for multiple services affected in general, or, with more specificity, such as multiple services -- primary no output; multiple services -- primary delayed.

(c) The purpose of the classification shall be stated. A classification for administrative monitoring (progress, satisfaction of requirements) will in general be simpler than one for technical monitoring (identification of causes, isolation of the potentially most dangerous failures), and much simpler than one intended to support research.

(d) Separate classification files for faults on the one hand and failures on the other are desirable because this facilitates capture of faults not associated with failures, particularly those found in inspections and reviews. Also, the fault file will primarily support the assessment of preventive measures, whereas the failure file will primarily support the assessment of protection and circumvention measures. Where it is desired to compare or combine the reliability experience from several sites or service functions, an environment classification file will be helpful that captures the characteristics of the individual sites or service functions, such as computer types and configurations, program language, and size of programs.

(e) The number of categories within a given field should be kept to a minimum. Where more than five categories are necessary, consider a two level scheme where categories at the top level are more finely divided at the lower level. This rule can be modified where the

number of categories are dictated by external conditions. Typical examples of this are designations of software or hardware components and of the program stage (phase) at which a fault is identified. It is also desirable (but not always feasible) to keep the scope of the categories as uniform as possible so that the a priori expectation is that an equal number of reports will be found in each category.

The following subsections provide general recommendations for file formats suitable for the nuclear power environment. Because of differences between sites and motivation for data collection the format will need to be tailored for most applications, keeping in mind the basic guidelines in (a) - (e) above.

3.3.2 Fault Classification File

The fault classification is intended to characterize defects that are encountered either by inspection, analytical activities, or execution (operation). The primary purpose of fault classification as described here is to aid in the minimization of future faults. This requires (a) allocation of resources to the areas in which faults reside and the activities associated with detecting them, and (b) technical assessment of causes and corrective measures. The minimum classification topics, covering only the (a) requirements, are:

1. Origin of the fault -- the top level categories are hardware, software, skinware, and environment. Lower level categories for hardware and software are the component in which the fault was found; for skinware it is the job classification of the responsible person and/or the shift; for the environment it is services (electricity, heating or cooling), fire, natural catastrophes, and hostile acts.

2. Activity responsible for detection -- categories are inspection, review, test, operation, and alarm (for skinware and environmental faults).

3. Identification stage -- program phase at time of detection

Additional classification topics for technical assessment of problems, and particularly for the selection of preventive measures, are listed below. These correspond to requirement (b) in the opening paragraph of this topic. The reason for the later placement is that the data for these fields are usually more difficult to get. Within the following listing the order is from the most commonly available to the least available data. In a given application the lower topics can be deleted and there will still remain a usable framework for the assessment of preventive measures.

4. Cause -- a multi-level classification with the top level identical to the origin of the fault. Categories for hardware are: specification or design, manufacture, inspection or test, and installation. Categories for software are: Requirements or specification, design, coding and test. There may be lower levels, particularly for the design category, such as algorithms, structure, data, and interfaces.

5. Isolation -- to identify the activity following detection that is required to proceed with corrective action. Categories are analysis, test, simulation, and special instrumentation.

6. Corrective action -- the top level categories are identical to those of item 1. For software and hardware further subdivisions are desirable for immediate and definitive actions. Immediate actions are: restart (without any other modification), restart with restrictions on operation, switch to an alternate, and replacement of failed component. Definitive actions are: redesign, permanent change in operating procedures, and special testing to detect the responsible condition.

7. Root cause analysis (where available) -- this classification topic can shed light on the project phase during which the fault was introduced. Categories are: requirements, development process, interfaces, and reviews and test. Lower level classification may be warranted in some cases.

3.3.3 Failure Classification

The failure classification is intended to characterize an event and its consequences. The purpose of the classification is to support the assessment of protection and circumvention measures that will minimize the undesirable consequences if the same fault is encountered again. As in the case of the fault classification, it is divided into two parts: (a) the minimum set that is based on directly observable data, and (b) a desirable set that incorporates data that may require further investigations.

The failure data must be obtained for all failures, including those that have no operational effects, such as switchover to a redundant unit, or masked failures. The use of data on failures that have no operational effect is discussed in the application chapter.

The recommendation for the minimum set includes:

1. Failure mode -- the event observed at the computer system level (not at the controlled plant level). Typical categories are: crash, incorrect response, late response, no response, no system effect. This is considered the key entry because the computer system response is available from test, and most of the target data are presumed to come from test. During test the effect on the plant cannot be directly observed, and it is therefore placed in part (b) of this file.

2. Date and time of failure -- required for several types of analyses: (i) sequential ordering of events, (ii) placement of event relative to development milestones or operational changes, (iii) investigation of time of day or workload effects, (iv) reliability growth modeling.

2a. Operation in progress -- the activity during which the failure occurred. Typical categories are: test (may be divided into categories), routine operation, heavy workload, other non-routine operation, maintenance.

2b. Trigger -- record specific events that contributed to the failure. Suggested categories are power transients, other exceptional environment, failure of associated components, unusual operator inputs.

3. Duration (time to restore service) -- important as one indicator of severity of consequences of the failure; this may be a numerical entry or categories such as less than one minute, one to fifteen minutes, and over 15 minutes.

4. Extent of the failure -- expressed in terms of facilities and services affected. Categories are: single channel, multiple channels (less than all), all channels. There can be subdivision for partial failures of channels.

The recommendations for the additional desirable set of data includes:

5. Criticality -- expressed in terms of effect on the controlled plant. In operational environments this can be directly observed and thus become part of the minimum data set. In a test environment analysis and judgment are required to propagate the effects observed at the computer level to the plant level.

6. Affected component -- identification of the component most directly affected by the failure. This can make use of the multi-level component identification introduced in the fault classification.

7. Detection mechanism -- means by which the first indication of failure was obtained. Top level categories include: alarm or trouble indication, operation of fault tolerance or protective provisions, gross anomaly or cessation of service (without prior alarm), periodic test.

8. Recovery mechanism -- means by which the error was eliminated. Categories include: fault masking, automatic switchover to another component, automatic roll-back or restart, operator intervention, shut-down of service.

9. Restoration mechanism -- required only where recovery does not lead to restoration of service. Categories include: replacement of defective component, re-load or re-initialization of software, and none (where recovery leads to restoration).

10. Prevention and circumvention recommendations -- this is equivalent to the root cause analysis in the fault classification. It is intended to capture the suggestions of personnel who observed a failure on the best means of preventing the effects

at the computer level or circumventing the effects at the service or plant level, assuming that the underlying fault or faults similar to it are still present. Categories include: improved monitoring of the computer operation, adding redundancy within the affected system, prevention (or improved monitoring) of the initiating events, circumvention measures at the plant level such as restrictions on power output under certain conditions.

3.3.4 Environment Classification

The classification of the environment in which the data for the fault and failure files were obtained is essential when comparisons are drawn among data obtained at different sites, or when data from different sites are to be combined. The environment file permits accounting for differences in maturity among installations, differences in computer equipment, programming language, plant characteristics, and peculiarities of data collection at a given site. Each file for a given site will result in a record in the environment file.

The major fields of the environment file are:

1. Site identification -- a coded, numeric designation of sites is desirable to assure protection of proprietary data and to avoid ambiguities in free text descriptions.

2. Service function -- typical classifications will include: reactor trip, emergency core cooling, containment protection, qualified display, venting and fluid disposal, support.

3. Digital system identification -- coded designation for the reasons described under 1.

4. Number of channels -- number of hardware replications, each one of which is capable of accomplishing the service function

5. Internal redundancy provisions -- within a given channel list number of replications for sensors, sensor communications, data converters external to the computer, computers, output adapters.

6. Computer language (may be subdivided into primary and additional languages) -- typical categories are: assembly, processor oriented languages (such as PL/M), structured languages (Pascal, Ada), object-oriented languages (C++).

6a. Size of developed source code -- may be numeric or in categories: less than 10k lines, over 10k and up to 30k, over 30k and up to 100k, over 100k.

6b. Fraction of non-developed code -- may be commercial or reused code. Estimate as a fraction of the size of the developed code in the following categories: none, less than 0.05, 0.05 to less than 0.15, 0.15 to less than 0.5, and over 0.5.

7. Development methodology -- typical categories are based on tool usage: compilers and related tools only, static analyzers and related tools, multiple tool usage without dynamic analyzer, multiple tool usage including dynamic analyzer, clean room.

8. Test methodology -- typical categories are: functional test, functional test with complete requirements coverage, functional and structural test with branch coverage of at least 0.95, functional and structural test with path coverage.

9. Independence of V&V -- is represented by the organization from which the V&V team is recruited. Typical categories include: the development organization, another development organization at the same plant, an independent quality assurance organization at the same plant, an outside organization.

10. Maturity of design (applicable to pre-operational systems only) -- expressed in years since start of coding. For system modifications use a weighted average, based on lines of original code and lines of added code.

11. Maturity of system (applicable to operational systems only) -- captured in two fields: number of years since completion of first acceptance test, and total number of installation-years. Both are counted to the start of the failure data collection to which this environment record applies.

12. Maturity of installation (applicable to operational systems only) -- expressed in years since start of operation.

3.3.5 Cost Considerations

It is realized that the capture of cost data is a sensitive issue. The following recommendations are based on technical needs that arise in performing trade-offs between the cost of advanced verification and test methodologies and the savings that these make possible in avoided software maintenance, plant shut-downs, and potential damage from an accident. Even approximate data will be beneficial in arriving at economically viable decisions about the application of advanced methodologies.

The following data are all intended to be supplied in current dollars. Labor hours (for defined skill levels) are an acceptable alternative in most cases. Adjustments for the time value of money are discussed in the next chapter. Such adjustments are necessary because the cost of the

advanced methodologies is incurred ahead of the time that the savings will be realized (in most cases the interval will exceed ten years).

The organization of the cost file is similar to that of the environment file in that data from each site will be entered as a record. The same site coding that is used in the environment file can be carried over here and will serve to protect the privacy of data. Fields 6 - 9 of the environment file can be used together with the cost data to assess the economic impact of various development and test methodologies. The information enables developers and regulators to identify practices which reduce the number of software problems at a low cost, and conversely, those that are costly without benefit to operational reliability.

The following cost data are required from the developing organization:

1. Overall development cost -- numeric or in categories: less than 0.2 million, 0.2 to 1 million, 1 to 5 million, 5 to 25 million, and over 25 million.

2. Time interval over which these costs were incurred

3. Cost of requirements formulation and analysis -- as a fraction of total development cost, in categories: less than 0.05, 0.05 to 0.1, 0.1 to 0.15, over 0.15

4. Cost of design -- as a fraction of total development cost, to be furnished separately for hardware and software in categories: up to 0.1, 0.1 to 0.2, 0.2 to 0.3, over 0.3

5. Cost of implementation -- as a fraction of total development cost, to be furnished separately for hardware and software in categories: up to 0.1, 0.1 to 0.2, 0.2 to 0.3, over 0.3

6. Cost of integration and test (without V&V activities) -- as a fraction of total development cost, in categories: up to 0.1, 0.1 to 0.2, 0.2 to 0.3, over 0.3

7. Cost of V&V -- as a fraction of total development cost, in categories: up to 0.1, 0.1 to 0.2, 0.2 to 0.3, over 0.3

8. Cost of software debugging and correction during development, in categories: less than 0.05, 0.05 to 0.1, 0.1 to 0.15, over 0.15

The following cost data are required from the using organization:

9. Number of system failures requiring maintenance (including those not causing an outage) during the (a) past year and (b) past five years (numerical entries)

10. Number of system failures causing a system outage during the (a) past year and (b) past five years (numerical entries)

11. Number of system failures affecting plant operations during the (a) past year and (b) past five years (numerical entries)

12. Estimated range of cost for failures not causing a system outage (min and max)

13. Estimated range of cost for failures causing a system outage but not affecting the plant (min and max)

14. Estimated range of cost for failures affecting the plant (min and max)

3.4 APPLICATION METHODOLOGY

This chapter suggests a methodology for applying the classification described above for gaining insight into the failure process at a given site, and for investigating global improvement to reduce the incidence of faults and to improve the protection against failures when they occur. A final section deals with economic analysis of the data. These examples represent only a small sample of possible investigations that will be supported by the classifications described in this report.

3.4.1 Site Specific Applications

The association of faults and failures with specific elements of the system is essential for allocating resources for both prevention and mitigation. With respect to faults this is accomplished directly from field 1 (origin of the fault) of the fault classification, and with respect to failures it is accomplished directly from field 6 (affected component) of the failure classification.

Once resources are available, they can be utilized for advancing the time of detection (earlier detection reduces the probability that the fault will lead to severe consequences). For this purpose fields 2 and 3 (activity responsible for detection and identification stage) of the fault file are examined, first with respect to detection of the specific fault, and then with respect to all faults in the component (possibly also in similar components). Detection activities that lead to early detection should be selected in the future.

If the fault resulted in a failure, fields 5 (criticality) and 6 (detection mechanism) of the failure classification are examined. Detection mechanisms that reduce the criticality of failures should be selected in the future.

The probability of a system failure can be computed from the data on non-system failures as shown in the following example.

Assume a two-out-of-four channel safety system; this configuration can sustain two failures and still remain operative but, because a subsequent failure will bring it to an uncertain state, the operating procedures require that the safety system must be declared non-operative after the

second failure. For the time being we are not concerned with the effect of this non-operative safety system on the plant.

Over the past twelve months there have been two channel (non-system) failures, one of which took one-half hour to repair and one took four hours to repair. The specific question to be answered: what is the probability of a second failure occurring while the first one is being repaired (plant operating procedures require that in case of a second failure the safety system will be declared non-operative). This probability of failure during the maintenance interval, P_m, can be computed from

$$P_m = \lambda \times \tau \qquad (1)$$

where λ is the failure rate and τ is the repair time (expressed in the same units as the failure rate). In our example the probability of a channel failure is 1/4320 hours = 231×10^{-6} per hour. The average time to repair is 2.25 hours and thus the expectation for a second failure while the first is being repaired is $2.25 \times 231 \times 10^{-6} = 521 \times 10^{-6}$. The first failure is indicated to occur twice a year, and therefore the probability of the system declared to be non-operative is 1.04×10^{-3} per year.

Because of the large variation in repair times it is desirable to augment this expected value calculation by one based on the longest time to repair (which represents a 50% probability in this small sample). In that case the probability of a second failure while the first one is being repaired is $4 \times 231 \times 10^{-6} = 924 \times 10^{-6}$, and the probability of the system being non-operative is 1.85×10^{-3} per year.

3.4.2 Global Applications

This topic is concerned with two broad application areas: drawing inferences from differences in fault or failure data based on environment characteristics, and merging data from different sites to form an aggregate database.

An example of the first type, drawing inferences based on environment characteristics, is at first glance very simple as shown in the following example. Two projects were started at about the same time, and after three years both have completed acceptance test. Project X has a fault density of 5 per 1000 non-comment source lines and project Y has a fault density of 2.5 per 1000 non-comment source lines. Project X was written in language A and project Y in language B. The hypothesis is formulated that language B leads to lower fault density than language A. Should this hypothesis be accepted?

A conventional approach to answering this question is to utilize statistical methods of testing hypotheses. But an underlying assumption in all of these is that the "treatments" being compared (in our case the two computer languages) are the only differences between the alternatives. This assumption is practically never valid for software development environments. Therefore the statistical procedures discussed below should be considered a clue rather than evidence of a

possible cause for the observed differences. Detailed comparisons of various elements of the software product and of the processes utilized must then be used to confirm or refute the statistical results.

Table 3.4-1. Hypothetical Comparison

Component or Computed Quantity	Project X	Project Y
	Fault Density, k lines	
1	6	0
2	7	8
3	4	0
4	3	9
5	5	2
6	4	1
7	6	0
8		0
n	7	8
$\Sigma\xi$	35	20
m	5	2.5
R	4	9
$\Sigma\xi^2$	187	150
σ	1.3	3.8

Because of the restricted role assigned to the statistical investigation it is rarely worthwhile to compute statistical confidence limits. The rules of thumb presented here are simpler (some might say cruder) and are drawn from the experience of the authors but they are based on the same fundamental principle as conventional statistical analysis: consider the hypothesis proven only if the difference is large compared to the "noise" in the data. For the small number of observations that are typical of this environment the noise is measured in the simplest approach

by the range[5], or in a more sophisticated approach by the internal standard deviation, of the measured variable within each population, in this case the fault density in project X and project Y. In our example the project X software consisted of seven major components and the project Y software of eight. The individual fault densities are shown in Table 3.4-1, together with key computed results from each data set.

Among the computed quantities (bottom of the table), n is the number of major components in each project, $\sum\xi$ is the sum of the component fault densities, m is the mean of the component fault densities[6] computed from $m = \sum\xi/n$, and R is the range (difference between the highest and lowest fault density for each project. For a significant statistical difference to exist the two treatments should differ by at least one-half of the average range. In this case the average range is 6.5 and the difference in fault density is only 3. Thus this test does not indicate a basis for accepting the hypothesis.

The variance is computed from $\sigma^2 = (\sum\xi^2/n) - m^2$ and the standard deviation, σ, represents the square root of the variance. As a rule of thumb a statistically significant difference exists only if the two observations (here the mean of the fault densities) differ by at least the larger of the two standard deviations. This criterion is also not met here, and thus there is no basis for accepting the hypothesis.

Where the failure rates of two projects are compared the noise content can be measured in terms of the intervals between failures. Assume that failures are observed at intervals of 100, 200, 20, 80, and 200 hours (five failures during 600 hours of operation). The mean time between failures is 120 hours, and the corresponding failure rate is 1/120 = 0.0083 per hour. The half-range is 90 hours, and at least that difference should by our criteria be observed to accept that another project has significantly superior or inferior product characteristics. The standard deviation can also be computed by the methodology outlined above and the same criteria can then be applied.

Merging of data can be motivated by administrative requirements or by technical ones. An example of an administrative requirement is to determine the total number of failures that have been observed in safety system software during the past year. For this purpose all failure files can be merged, and a total for the appropriate dates (see item 2 of the failure file) can be obtained. Differences in functions served by the software, in the associated hardware, and the year of development do not affect the suitability of the data.

Technically motivated data merging may aim at increasing the number of observations by combining fault or failure files from several projects to facilitate statistical comparisons. Suppose

---

[5] For larger populations the difference between quartiles should be substituted for the range.

[6] This will usually be different from the mean software fault density which is obtained by dividing the total number of faults by the total number of lines of code.

that instead of the single projects with languages A and B that were discussed above there were four that used each of these languages. The resulting larger data sets normally reduce the noise and thus increase the likelihood of positive results when hypotheses are tested. Merging for this objective requires considerable care in the evaluation of environment factors. This can be accomplished by evaluating at least those items in the environment file which have been identified as contributing to differences in fault density, failure rate, or related characteristics: function and size of the software, characteristics of the hardware, maturity of hardware and software, and the year the project was initiated.

In addition, statistical criteria can be used to determine whether two data sets come from a homogenous population. In principle this is just the reverse of the process described to show that populations are distinct: it must be shown that the difference in the means is within the noise boundaries. The limits that will be set for acceptance depend on the purpose for which the combined data will be used. If an unenforced recommendation for future use is to be formulated data sets can be combined with more generous limits than when mandatory acceptance criteria are to be generated.

3.4.2 Application of Cost Data

Cost data are difficult to obtain because of (a) proprietary and privacy concerns, (b) unavailability in a format that supports technical investigations, and (c) bias caused by the desire to adhere to contractual or internal budgets. Therefore a fairly low resolution must be accepted for all economic investigations based on publicly available data. But these limitations cannot cause abandonment of cost considerations because that would lead to promotion of only the most sophisticated technical methodologies. Thus a general approach is presented here for using whatever cost data are available in a responsible manner.

Expenditures for high integrity digital systems (beyond the cost of equivalent conventional systems) are motivated by the objective of avoiding:

a. catastrophic failures affecting the plant and its environment

b. unnecessary plant shut-down due to failure of the digital system

c. maintenance cost (personnel, materials, test equipment)

The cost associated with plant shut-down and avoidance of maintenance are usually well known, and the probability of these events can be computed from generally available data by the methods described in Section 4.1. Examples of measures to reduce the frequency of failures in the digital system are advanced software design and test techniques or additional hardware redundancy. Expenditures for these have to be made during the system development phase, whereas the benefits accrue during the operations phase. Because the expenditures precede the expected benefits by several years, an adjustment for the time value of money is necessary. The effective

rate of interest that is used in this adjustment is a matter of judgment but it is well above the published "prime rate" or similar indices because:

a. the money is tied up for many years

b. the benefits are not certain

c. intervening events may preclude system operation into the benefit period (plant shut-down, mandatory replacement of the safety system).

The accompanying table therefore uses interest rates that are high by prevailing standards. The interval between the expense and the benefit can be divided into three periods with the following characteristics: (i) development -- uniform pay-in over two to five years, (ii) qualification and marketing -- a waiting period during which neither pay-in nor pay-out occurs, (iii) operation - a uniform pay-out period of ten to twenty years. Texts on Quantitative Methods in Economics develop exact formulas for handling these conditions [BIER69], and abbreviated forms of these are found in many spread sheet programs. Because of the inherent limitations of cost data that were pointed out above only an approximate method is shown here which is based on a pure waiting period between the mid-point of development and the mid-point of operation, an interval that ranges from a minimum of 10 years to a maximum of 30 years. The factors shown below can be used either to multiply the expected cost or to divide the expected benefits.

Table 3.4-1. Cost Adjustment Factors

Interval (years)	Interest			
	8%	10%	12%	15%
10	2.16	2.59	3.11	4.05
15	3.17	4.18	5.47	8.14
20	4.66	6.73	9.65	16.37
25	6.85	10.83	17.00	32.92
30	10.06	17.45	29.96	66.21

For all but the top row it is seen that these adjustment factors can make a large difference in selecting economically viable improvement alternatives.

3.5 CONCLUSIONS AND RECOMMENDATIONS

The selection of software development methodologies for high integrity systems in general, and specifically of verification and validation methodologies, cannot be based purely on examination of features and suitability for a given environment. Substantial weight must be given to the effectiveness of a methodology in preventing or detecting errors that interfere with the performance of the function assigned to the software product. This effectiveness cannot be assessed without access to an organized database on failure modes and failure rates of existing systems.

The error classifications recommended here are intended to be used in such a database, and will thus contribute to the objective evaluation of software development methodologies, and particularly of verification and validation methodologies.

Because a database is so essential to the development of a verification and validation policy, and the evaluation of supporting methodologies, it is strongly recommended that the delivery of failure data (pre-installation and post-installation) be made a condition of future licenses for both digital and analog safety systems, and that these data be provided in the format shown in Sections 3.3.2 - 3.3.4. These formats have been designed so that they can be easily completed by the developer or maintainer, while at the same time yielding useful data for the analyst. The first (administrative) part of the fault and failure reports should be required in all cases. The second part (technical analysis) can be omitted if supplying it presents a great hardship on the licensee. Within the scope of this report the data source is always the plant operator (utility). It is assumed that the utility will generate requirements for vendor data where this is appropriate.

Because of the very limited data from nuclear power plants this chapter has utilized data from other sources to draw some inferences, but the limitations of that process had to be recognized. Examples of findings that are particularly relevant are:

- software support for fault tolerance was a leading cause of serious failures in at least one environment (Table 3.2-8)

- incorrect response rather than complete loss of computational capability was the leading error manifestation in another environment (Table 3.2-12)

- inability to handle multiple rare conditions that were encountered in close time proximity was the leading cause of failures in a third environment (Table 3.2-13). Note that this finding is consistent with the first one mentioned above since fault tolerance management is involved in many rare conditions

As part of this effort partial data from the operation of a digital safety system at a nuclear generating station became available, and example analyses of these data are presented in Appendix B. The data had been requested by the NRC as part of the licensing conditions, but apparently no specific data format had been imposed. In spite of the essentially unformatted

nature of the data, some interesting inferences could be drawn. Had the data been provided in the format recommended in Section 3.3 much more useful information could have been obtained.

The difficulty of obtaining software failure data is not restricted to the nuclear power field. However, during the past year some encouraging signs have emerged, such as the establishment of a software reliability database repository under the auspices of the American Institute of Aeronautics and Astronautics [AIAA92], and the emphasis on software measurement (with implied data collection) within DoD, as evidenced by the Cooperstown I workshop on this topic, convened in September 1993 with participation by the office of the Secretary of the Air Force [COOP93].

SECTION 4 - VERIFICATION AND VALIDATION OBJECTIVES

4.1 OVERVIEW

This chapter does not directly address a paragraph in the Statement of Work, but contains material that forms the basis of, and is common to, the following three chapters that deal, respectively, with metrics, verification, and validation. The organization of this chapter therefore differs from that of the others that are devoted to specified tasks.

The first part of the body of this chapter investigates broad requirements for assuring that high integrity software does not fail in operation. This part highlights the major areas of reliability and safety concerns and forms the technical basis for evaluating the methodologies described in the following three chapters.

The final part of this chapter discusses the specific roles that metrics, verification and validation play in achieving the objectives previously defined.

4.2 FREEDOM FROM FAILURE IN OPERATION

The user as well as the regulatory agency require that software in safety systems operate exactly as required all the time. The common goal of metrics, verification and validation is to demonstrate that this requirement is met. Specifically, this involves showing that each feasible combination of requirements will be met for all data inputs and states of the computer and associated components (sensors, effectors, and communication channels). This may be accomplished by exhaustive test or by exhaustive analysis, e. g., through formal methods. Neither approach is feasible for realistic programs. Exhaustive test is impossible because of the extremely large number of test cases that have to be run and evaluated [HOWD78, HAML90], and exhaustive analysis because current formal methods are deficient in either expressiveness (for direct mapping of plain text requirements) or capability of automated proof checking (or both) [RUSH92].

Because assurance of complete freedom from failure thus seems to be beyond our grasp, the approach here is to identify the functions and operations that have been persistent sources of software errors so that the verification and validation methodologies can focus on these. The selection of difficult functions is based on the experience of the authors of this report and includes primarily operations that are invoked infrequently or under unpredictable conditions. Examples are software exception handlers, hardware diagnostics and redundancy management software, and operator initiated mode change or reconfiguration sequences [KANO87, VELA84]. The reason for concentrating on these is partially based on the large fraction of rare events failures (mostly involving the above examples) in the NASA Space Shuttle as reported in the preceding chapter, and partially on the almost trivial observation that operations that are routinely executed or are executed under predictable conditions are analyzed and tested more thoroughly

than those that are not. A related discussion, including analysis of test results reported by others, has been published recently [HECH93].

Software failures arise primarily from two causes: incorrect implementation of system and interface requirements, and incorrect use of software constructs. The latter cause is dealt with by use of modern high order programming languages and the associated tools. Metrics and verification practices that address that area are discussed in the appropriate chapters that follow. The present discussion is therefore restricted to requirements that flow down from the system level.

Software development depends on well defined requirements in at least the areas listed below. For each area both the system requirements and the specific subset to be implemented in software must be identified. To permit comprehension of the requirements, the external system interfaces, operator procedures, operator and maintainer skill levels, and security requirements should be made available to the software developers. To facilitate reference in later text, each area of concern has been given a short title, listed in italics, that is only partially descriptive but will hopefully serve as a memory jogger.

1. *Normal Service:* The service to be performed by the system[7] in each operating mode

2. *Failure Modes:* Failure modes of the hardware required for these functions, fault detection requirements (including calibration and self-test) and fault tolerance provisions and algorithms

3. *Unsafe Actions:* Specification of actions to be avoided by the system

4. *Human Interfaces:* Identification of the human interfaces for (a) normal operation, (b) exceptional operating states (e. g., recovery from hardware failures), and (c) maintenance and other non-operational states

5. *Isolation:* software segments that perform functions not directly related to plant safety must be well isolated from the safety critical segments

6. *Test:* System level test activities and the software support required for these (test drivers, simulators, enabling/disabling provisions for certain functions)

7. *Attributes:* Attribute requirements: quality assurance, configuration management, reliability, and availability.

[7] In this context *system* designates a portion of the nuclear power plant that can be separately tested and that is controlled by the software under development. Examples are: plant shut-down system, containment isolation system, auxiliary feedwater control system.

A more detailed description of the significant requirements that arise in each area is presented below.

4.2.1 Definition of Normal Service

The definition of system service enables the software designer to provide required software functions. This is the most conventional part of software development for which many standards and methodologies are available. The following are specific items that apply to most nuclear power software:

- Operating modes (plant start-up, system start-up, routine operation, maintenance or test mode, plant shut-down)
- Allowable transitions between modes
- Method and frequency of invocation in each mode (cyclic, by event, operator command)
- Possible states of the controlled plant at time of invocation
- Function to be performed in each mode (e. g., safety algorithm)

4.2.2 Failure Modes, Error Detection and Fault Tolerance Requirements

The hardware installation for safety systems in nuclear power plants usually includes redundant channels for each safety function. Software may be required to validate operational channels, identify faulty channels to the operators, perform automatic switching between channels to maintain the safety system operational after a fault has been diagnosed, and to initiate alerts when the safety system is no longer fully functional. These activities are collectively referred to as surveillance. Sensors are substantial contributors to the system failure probability, and frequently sensors have a higher degree of redundancy than other hardware components. Sensor surveillance is therefore discussed in a separate subsection below.

4.2.2.1 Sensor Surveillance

Sensors operate under more severe environmental conditions than other parts of the system. Their output normally contains a noise component, it can drift, and it is frequently affected by variations in the power supply. In addition, the sensor can experience transient or permanent failure. In analog systems sensor surveillance is a labor intensive activity, and one of the advantages of digital systems is that they permit it to be automated (i. e., implemented in software). The sensor surveillance software typically analyzes a time series of sensor outputs, extracts a current estimate of the true value of the sensed quantity from the noisy raw measurements, and must make decisions about the validity of the current estimate (i. e., whether the sensor has failed). If a failure has been identified there may be further decisions required about the value of the affected variable that is utilized in the system, e. g., to minimize sensor switching for transient failures it can be temporarily held at the last valid level and the affected sensor sampled again during the next interval. The design of sensor surveillance software

requires identification of sensor and power supply redundancies, the preferred sensor configurations, and the following data:

- sensor failure modes
- sensor range under normal plant conditions
- sensor range under abnormal plant conditions
- mechanical and electrical limits on sensor output
- maximum expected change in output between samplings
- noise characteristics of the sensor and power supply
- worst expected drift characteristics of the sensor
- allowable time interval between abnormal sensor output and safety action
- typical time history of plant conditions requiring safety action

Sensor surveillance normally includes the wiring to the control components; i. e., a failure in the connection will be treated as a sensor failure. Where separate surveillance of the wiring is desired, the software designer will need data that permit a differentiation between sensor and wiring failures.

4.2.2.2 Surveillance of other System Components

Other system components typically include the computer and output devices, such as a relay network. In some cases the output interface includes actuation of control rods or pumps. Surveillance of computer operation includes at least a self-health check, but it can also include monitoring of computers in other channels, of analog-to-digital interfaces, and of intra- and inter-channel communications.

The surveillance of the output devices involves comparison of the commanded state (as generated within the computer) with the actual state and reported by an independent measurement. For relay networks this measurement is usually provided by an auxiliary contact that operates in synchronism with the main contacts; rod position can be determined from dedicated sensors, and pump operation from centrifugal switches or tachometers.

The software designer needs the following data to support required functionality:

- computer and output device failure modes
- error detection and correction requirements arising from these
- the topology of the intra- and inter-channel communications
- alternate allowable topologies to deal with component failures
- data formats used by each communications path
- maximum expected delay between output command and output activation
- allowable delay between detection of a faulty state and annunciation

4.2.3 Unsafe Actions

These actions must be identified in the requirements; they fall into two broad categories:

1. Actions to be avoided in normal computer operation

2. Actions to be avoided after computer or software failure

The first category includes actions that may result from failures outside the computer, such as an erroneous sensor measurement or inappropriate operator actions (mode changes). Examples are:

- prohibition of repeated output commands (sending a command twice)
- definition of prohibited output sequences
- actions to be avoided during or immediately following a mode change
- prohibition of actions after detection of a sensor failure
- prohibition of actions after detection of an output device failure

Examples of the second category are:

- actions to be avoided after self-diagnosis of a failure
- actions to be avoided after detecting failure of another computer
- prohibited actions after entering a software exception handler.

In addition to these requirements that are derived from the system specification certain actions to be avoided may be established on the basis of software considerations, e. g., prohibition of certain calling sequences.

4.2.4 Human Interfaces

Although safety systems are frequently intended to serve functions in which the human response may be too slow or uncertain, they are not insulated from interfaces with operators and maintainers. Under failure-free conditions of the safety system the operator initiates mode changes and monitors plant and system status indications furnished by the automated system. Under exceptional states of the safety system, e. g., recovery from a hardware failure, the operator is responsible for taking corrective action, such as initiating maintenance. And once the system is in a maintenance mode, human skill and judgment is required to bring it back to operation. These essential human interfaces demand that the software developer be aware of:

- Availability and capabilities of the operational staff
- Suitable provisions for staff initiated test of the system
- Human interfaces of present or predecessor systems (to avoid introduction of inconsistent input or display formats)

- Alternate actions that may be initiated by the operational staff for a given plant condition (including remotely initiated actions)
- Alternate indications of a given plant condition available to the operators
- Training facilities for the operational staff (to permit integration of training for the system under development)
- Availability and capabilities of the maintenance staff
- Diagnostic provisions desired by the maintenance staff
- Plant operating procedures while system maintenance is in progress
- Procedures for restoring the system to operation following maintenance

4.2.5 Isolation

Software segments that perform functions not directly related to plant safety (diagnostics, self-test, initialization or shut-down, calibration of limited scope) may receive a lower classification than the directly safety critical segments (see Chapter 2), provided that they are prevented from interfering in any way with the execution of the critical segments. The required isolation involves:

- restricting the execution time for non-critical segments; this is usually accomplished by use of watchdog timers (interval timers that operate independent of the application program and generate an interrupt if the application does not terminate in the specified time)

- preventing non-critical segments from writing into the memory utilized by critical segments

- preventing non-critical segments from accessing input/output ports used by critical segments

- avoiding the use by non-critical modules of communication buses used by critical ones, or restricting the time allocation for non-critical modules (e. g., by prioritized polling)

- assurance that the non-critical modules will terminate in the computer state and operating mode that was present when they started

4.2.6 Test

To facilitate system test it is frequently desirable to (a) disable or modify certain software controlled functions, (b) to add temporarily functions normally supplied by the system environment, and (c) to provide indications and records of test progress. If these requirements are realized at the outset, patching or other irregular software structures can be avoided. Requirements for the following functionality should be provided, associated with the test phases for which they will be activated:

- functions to be disabled or modified, e. g., feedback of output actuation
- differences in input timing or sequencing
- single channel operation (vs. multiple channels in the plant)
- fault insertion capability (including superposition of noise)
- simulation of operator commands
- programmed or random generation of inputs or internal states
- indications or recording of internal states, test sequence numbers, and generated outputs
- provisions that assure restoration of normal service after test

4.2.7 Attributes

System level attribute requirements must be propagated and interpreted for the software development. The primary attribute requirements arise from quality assurance, configuration management, reliability and availability. Security and portability (ability to operate on multiple computer types) requirements may also be invoked. In most cases these requirements must be interpreted for software development, and this interpretation is a joint system engineering and software engineering responsibility.

The most stringent requirements are usually intended only for the code associated with the activation of a safety function (reactor shut-down, containment isolation). But the extent of that software segment and the attribute requirements for other segments must be identified by joint system engineering and software engineering analysis techniques. Typical topics are:

- status (safety-critical or not) of
  - sensor surveillance software
  - software for monitoring and diagnostic indications
  - mode change software
- attribute requirements for
  - the above functions judged to be not safety-critical
  - test support software (subsection 4.2.6 above)
  - software exclusively used in non-operational modes

4.3 DISTINCTIVE ROLES OF METRICS, VERIFICATION AND VALIDATION

While much of the literature refers to "V&V" or verification-and-validation as an indivisible entity, there are conceptual differences between verification and validation that are described in the following. Metrics, which are discussed in the immediately following chapter, are an important yardstick that is used in both verification and validation. Metrics can serve (a) to identify software segments that need special emphasis in verification or validation, e. g., because of their complexity, and (b) as criteria for satisfaction of verification or validation activities, e. g., the branch coverage metric can indicate that structural test requirements have been met.

The definitions of the key terms in the IEEE Standard for Verification and Validation Plans (IEEE Std. 1012-1986) are:

> *verification* The process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements established during the previous phase.
>
> *validation* The process of evaluating software at the end of the software development process to ensure compliance with the software requirements.

The definitions in IEEE/ANS Std. 7-4.3.2 differ in identifying the scope of the activities as the computer system rather than the software:

> *verification* The process of determining whether or not the product of each phase of the digital computer system development process fulfills all the requirements imposed by the previous phase.
>
> *validation* The test and evaluation of the integrated computer system to ensure compliance with the functional, performance, and interface requirements.

The latter definition of validation permits the inclusion of end-to-end tests which are more significant than tests of the software alone. On the other hand, verification activities, with the possible exception of requirements verification, are normally conducted at the software level. Therefore the first definition of verification and the second definition of validation have been adopted in this report (see Glossary).

In a recent publication validation is identified with "building the right product" and verification with "building the product right" [BISH90].

Verification is a continuing activity throughout the development stage, whereas validation is defined above as a single activity (this restriction is modified in Chapter 7). Also, the definition of verification as a stepwise process, covering at a given time only a limited interval of the development, suggests that it may be amenable to automation, or at least to being conducted by means of a check list. In contrast, because the scope of validation includes the entire development, it is less suitable for being precisely defined or automated. These distinctions cause the activities associated with verification to be substantially different from those for validation, and this has led to treatment of these topics in separate chapters in this report.

Test is usually employed in both verification and validation, but because it is more essential to validation the discussion of test methodologies and test termination criteria has been allocated to that chapter. In accordance with the definitions shown above, the planning and analysis of results of the acceptance test is a validation responsibility, whereas equivalent analyses for unit testing and software integration testing are typically allocated to verification.

4.4 CONCLUSIONS AND RECOMMENDATIONS

The objective of this chapter was to collect information that is applicable to metrics, verification and validation so as to avoid repetition or frequent cross-referencing. The most important overall conclusion is that specific objectives for each of these activities must be stated. As error data become available (see Chapter 3), the information derived from these will become the source of directions for verification and validation, as well as for the metrics that support these.

In the interim we have attempted to (a) draw a strong distinction between verification and validation activities (Section 4.3), and (b) call attention to attributes associated with failures encountered in other high integrity environment, and which should therefore be addressed by the verification and validation processes (Section 4.2).

SECTION 5 - QUALITY METRICS

5.1 OVERVIEW

This chapter responds to paragraph 4.1.6 of the Statement of Work, which reads in part:

> Develop quality metrics for evaluating digital safety systems. Identify [and] evaluate existing metrics, or develop new metrics, for software quality attributes such as completeness, consistency, structure, verifiability, traceability, modularity, etc. The quality metrics shall be indicative of the absence of critical errors in safety system software.

Metrics are desirable because they may furnish a quantitative and, it is hoped, objective assessment of software attributes that can otherwise only be characterized in a qualitative and subjective manner. Metrics have been applied to software development in three major areas: project management (cost estimation, scheduling), performance (memory utilization, timing margins) and quality (reliability, maintainability). This investigation is restricted to quality metrics, i. e., quantitative statements about the extent to which the software possesses attributes associated with quality.

Software quality metrics can be aimed at the development process (process uniformity, process improvement) or at the product. A high quality software development process is essential for producing a high quality product, but it cannot by itself assure that the software produced in that environment possesses the reliability and safety attributes required for high integrity systems. Therefore the emphasis in this report is on finding product metrics that have high correlation with safety and reliability as established by test.

The most comprehensive and directly applicable product quality metrics can only be obtained late in the development process, e. g., the measurement of software failure rate requires an executable program in a formal test environment (instrumented to capture execution time of code segments). Metrics that can be obtained earlier in the development are more desirable because more effective remedial action is possible and less rework is required. Two caution flags must be raised in this connection: (a) metrics obtainable early in development are usually not as correlated with the validators (e. g., reliability measurement) as are metrics obtained in later phases, and (b) no single metric will capture the total quality requirements of the user.

Among currently used metrics none were found capable of furnishing objective and absolute measures of freedom from software faults. The investigation has focused on metrics that are indicators of reliability because this is considered the most important single quality factor for high integrity software. A modified form of the Halstead metric [HALS77] (which is based on the "mental discriminations" required to produce a software segment) has shown an acceptably high correlation with fault density in one application to real-time air traffic control software written in Ada. This has a potential of furnishing a relative measure of freedom from faults

(identify failure prone and less failure prone software components), and it is recommended that the suitability of this metric for safety grade software be investigated by applying it to software segments for which test data from the final phases of development are available. Potential uses of this metric are

- alerting developers, users, and the licensing organization to potentially troublesome software segments or major components; this will permit corrective action at an earlier stage than waiting for test results

- permitting V&V efforts to concentrate on the most failure prone components

- after experience is gained with the metric it may be used in a proactive way to discourage the development of components that exceed a certain threshold value.

The most comprehensive framework of software quality metrics encountered in our investigation is that developed under the sponsorship of the USAF Rome Laboratory. Most of the metrics of interest require manual evaluation, and the overall process associated with the framework is too complex to be practical for high integrity software for nuclear power plants. However, portions of the detailed metrics that are part of the framework may be converted into checklists to guide the V&V effort in the areas of reliability, maintainability and verifiability.

The greatest difficulty encountered in this effort has been the lack of metrics that can be obtained early in the development and that have demonstrated high correlation with relevant later metrics such as fault density or failure rate. The preference for metrics that can be used early in the development is motivated by the much lower cost and greater effectiveness of corrective action when deficiencies are discovered prior to full design and coding [BOEH81]. From the regulatory point of view, the benefit of early detection is that subsequent effort is devoted to "clean" software, thus reducing the probability that the delivered product will have been subjected to only an abbreviated regression test. The potentially low correlation between early metrics and later ones has already been mentioned. Very few quantitative studies have been published on this, and some of those that exist may be flawed, e. g., by not accounting for differences in size of code segments that are used for the regression. One of the most useful early metrics is the number of deficiencies identified in design reviews (normalized to the expected size of the code), but there is no evidence that the types of deficiencies uncovered in reviews are strongly correlated with deficiencies that cause failures in test or operation. A possible explanation is that the deficiencies reported in a review depend on the quality of the review as much as on the quality of the product being reviewed. If meaningful standards for reviews can be generated, a better correlation could be obtained. None of the early life cycle metrics investigated was judged to be useable as an absolute verification criterion.

While the quest for metrics valid during early life cycle phases should be continued, the aim of early correction can also be achieved by use of a spiral development approach, particularly when executed in accordance with the paradigm "build a little, test a lot".

The major headings for the body of this chapter are:

Section 5.2: Major Frameworks for Quality Metrics

Section 5.3: Specific Software Metrics

Section 5.4: New Quality Metrics

Section 5.5: Conclusions and Recommendations

Reliability has been selected as the key validator of the metrics although it has been remarked that there is a distinction between safety and reliability, and that the two may have differing objectives [LEVE86]. The reference mentions that the reliability of munitions is based on probability of detonation when desired, whereas safety is the probability of not detonating when not desired. The latter goal can frequently be enhanced at the expense of the former (by reducing the probability of detonation at any time).

In the context of high integrity software for nuclear power plants the commonality of objectives between safety and reliability is vastly more important than their differences. The consequences of a software failure are frequently very dependent on the instantaneous environment. Therefore it can be difficult to distinguish safety related faults from non-safety related ones. Further, serious failures are often caused by a combination of individually less serious anomalies, and therefore safety is substantially enhanced as the overall failure rate is decreased. Finally, it is acknowledged that safety requires steps beyond those undertaken for the sake of reliability, such as assertions in the code that prevent the issuance of unsafe commands or place the computer into an unintended state. These steps are an important concern of V&V procedures in general but they are difficult to capture in metrics. Therefore the quality metrics discussed here are primarily aimed at reliability.

Reliability is preferably measured in terms of failure rate based on execution time. Since the failure rate of delivered software is hopefully zero or very close to it, the correlation should be based on the observed failure rate at the beginning of formal test. This is also desirable because the metric is intended to capture the quality achieved during development. The reliability of the accepted software (at the completion of formal test) is largely a function of the effectiveness of test and is not within the scope of conventional quality metrics. Metrics for the effectiveness of test have been proposed [VOAS92] but have not yet been sufficiently validated to be considered for application to high integrity software.

Throughout this chapter the term *complexity* is used to denote structural complexity of the control flow, of the data flow, or of input-output relations between software elements because these attributes are easily quantified and can in most cases be captured by automated tools. Functional complexity is much more difficult to quantify. However, well-designed programs do not introduce structural complexity unless it is called for by the functional complexity of the task. Verification techniques, such as static analysis, indicate the presence of unneeded structural

complexity and motivate its elimination. Under these conditions structural complexity can be taken as a proxy for functional complexity. A gross direct measure of functional complexity is provided by a metric that counts the number of *shall* statements in the requirements (included in the Rome Framework discussed in the following section). There is no evidence that this is superior to reliance on structural complexity alone.

5.2 MAJOR FRAMEWORKS FOR QUALITY METRICS

Just as there is no single metric for the quality of a motor vehicle, there is none for the quality of software. Even within a given application area, evaluators may find it very difficult to agree on a set of metrics, leave alone a single one, that adequately expresses all the required quality attributes. The two frameworks, for product and process metrics, described in this section represent attempts to provide an integrated ensemble of metrics for the entire software development. The user is expected to define quality goals (factors) for the application, and from this the framework guides the selection of detailed metrics that match the objectives. The use of individual metrics that can be assembled into a do-it-yourself framework is discussed in the following section.

5.2.1 Rome Laboratory Software Quality Measurement Methodology

Description of the Framework

The Rome Laboratory Software Quality Measurement Methodology [MCCA77, BOWE85] is the most complete product oriented quality metrics framework encountered in this investigation. It is based on the hierarchical ordering of software quality attributes originated at TRW [BOEH73] to enable a software acquisition manager to determine and specify software quality factor requirements. Most of the concepts are applicable to software developed for high integrity systems.

The model is hierarchical (see Figure 5.2-1) in which *factors* needed by the user (e.g. reliability, correctness, maintainability) are at the top level, software oriented *criteria* are at the next level, and *metrics* -- quantitative manifestations of the criteria -- are at the lowest level.

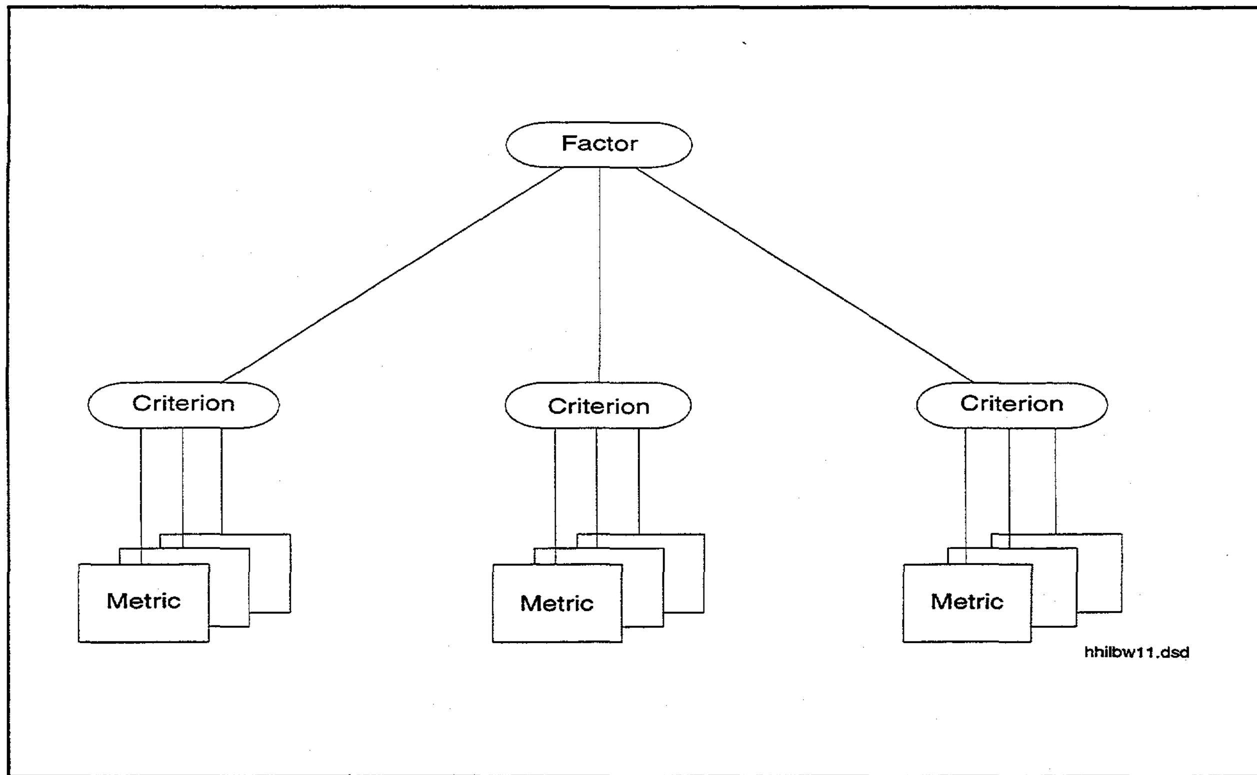


Figure 5.2-1 Software Quality Model

The model permits updating of individual elements to reflect technology advances without affecting the basic structure of model. For example, as new user concerns evolve, new factors can be added at the top level; and as software technology evolves, criteria and metrics can be added, deleted, or modified as necessary. There are currently 13 quality factors, 29 criteria, 73 metrics, and more than 300 metric elements (distinct parts of a metric). Table 5.2-1 shows the 13 quality factors together with an evaluation of their applicability to high integrity software for nuclear power plants. Neither factors nor criteria are necessarily disjoint. This problem is discussed in the evaluation of the framework.

It will be recognized that the first five factors deal with product performance; the next three factors deal with product design; and the final five deal with product adaptation. This last group of subjects is of importance to the developer because adaptability enhances the market. This area is not a major concern for either the regulatory agency or the user. Reusability has been identified as having some applicability because licensing will be easier if modules that have a good operating record can be reused in new installations.

Figure 5.2-2 shows the quality factor, criteria, and metrics in the hierarchical relationship of the software quality framework for reliability as an example.

Table 5.2-1 Quality Factors in the Rome Labs Framework

No.	Factor	Applicability to High Integrity Software
1	Efficiency	Not a high integrity concern; of interest to user
2	Integrity	Limited applic.; defined as access protection
3	Reliability	Highest rated factor; freedom from failure
4	Survivability	Limited applic.; ability to work in disturbed envrnmt.
5	Usability	Limited applic.; absence of special training req'mt
6	Correctness	Applicable; conformance to specification and standards
7	Maintainability	Applicable; ease of locating cause of failure
8	Verifiability	Applicable; ability to identify specified operation
9	Expandability	Not a high integrity concern; of interest to user
10	Flexibility	Not a high integrity concern; suitability for other uses
11	Interoperability	Not a high integrity concern; refers to interfaces
12	Portability	Not a high integrity concern; change of computers
13	Reusability	Limited applic.; other uses of components

The four factors that were identified in Table 5.2-1 as applicable to high integrity software encompass the following criteria (definitions for these are provided in the Glossary of this report).

Reliability	Accuracy Anomaly Management Simplicity
Correctness	Completeness Consistency Traceability
Maintainability	Consistency Visibility Modularity Self-Descriptiveness Simplicity

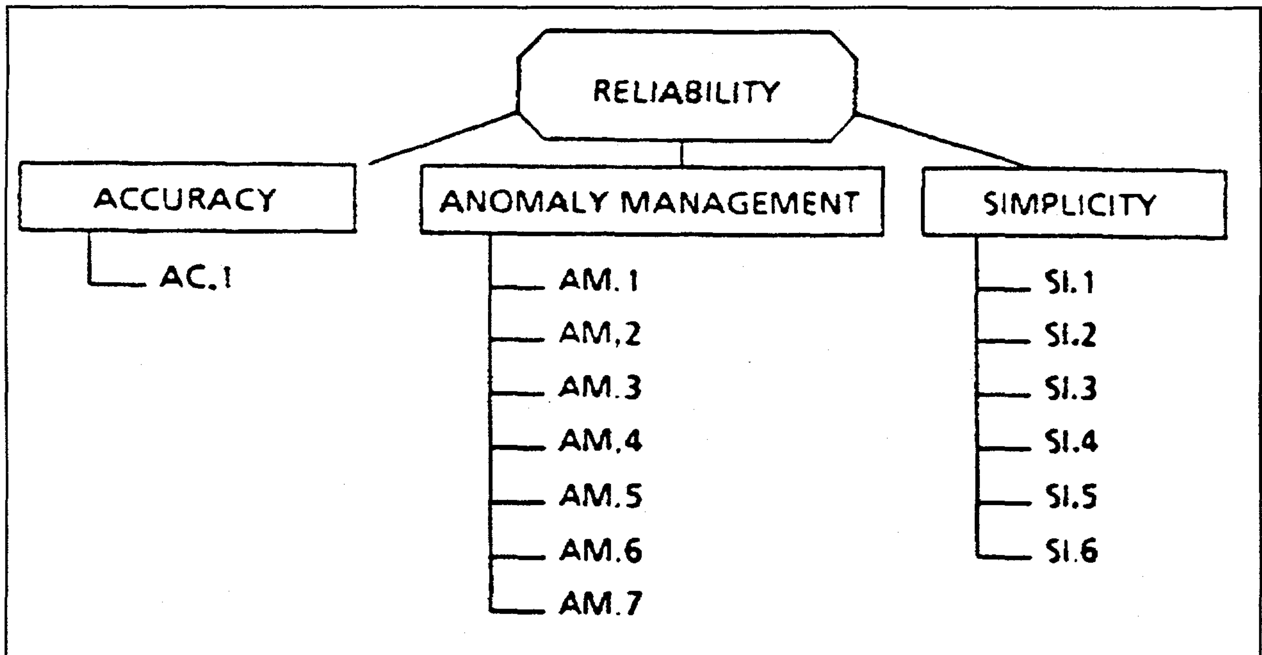


Figure 5.2-2 Reliability Factor

Verifiability	Visibility Modularity Self-Descriptiveness Simplicity

Of the areas of concern discussed in the previous chapter, only two are addressed by the criteria shown here:

Normal Service: by accuracy, completeness, and consistency

Failure Modes: by anomaly management

The factors that were identified as having limited applicability contribute the following additional criteria (those not applicable or already listed under the primary factors are not shown):

Integrity	System Accessibility
Survivability	Autonomy
Usability	Training
Reusability	Document Accessibility Independence System Clarity

The training criterion partially addresses the human interface area of concern.

After elimination of duplications, a total of 15 criteria are involved in metrics that are pertinent to the development of high integrity software. The factors and criteria identified here cover all software attributes that were specifically mentioned in the statement of work for this effort as follows:

- completeness: a criterion under the correctness factor
- correctness: a factor
- predictability: related to consistency, a criterion under both correctness and maintainability
- robustness: related to the survivability factor
- consistency: a criterion under both correctness and maintainability
- structure: related to system clarity, a criterion under reusability
- verifiability: a factor
- traceability: a criterion under correctness
- modularity: a criterion under maintainability and verifiability.

An example of the metrics that support the accuracy (AC) and anomaly management (AM) criteria is shown on the following pages (Figure 5.2-3, parts 1 and 2). Note that all metrics listed there require evaluation by trained personnel, and that subjective judgment enters into at least some of them. Metrics supporting other criteria can in some cases be automated (interpreted by machine readable data).

Evaluation of the Rome Laboratory Framework

The major strengths of the framework are its comprehensiveness and its coverage of the entire development cycle. The weaknesses are closely related to the strengths: the comprehensiveness demands considerable training before it can be properly applied and increases the effort for using it, and the coverage of the entire development cycle brings with it substantial reliance on subjective metrics (before machine readable software becomes available).

METRIC WORKSHEET 1 CSCI LEVEL

SECTION B - METRIC QUESTIONS

AC.1(3) Are there quantitative accuracy requirements for all applicable inputs associated with each applicable function (e.g., mission-critical function)? Y | N | N/A

AC.1(4) Are there quantitative accuracy requirements for all applicable outputs associated with each applicable function (e.g., mission-critical function)? Y | N | N/A

AC.1(5) Are there quantitative accuracy requirements for all applicable constants associated with each applicable function (e.g., mission-critical function)? Y | N | N/A

AC.1(6) Do the existing math library routines which are planned for use provide enough precision to support accuracy objectives? Y | N | N/A

AM.1(1) a. How many instances are there of different processes (or functions, sub-functions) which are required to be executed at the same time (i.e., concurrent processing)? ____ N/A

b. How many instances of concurrent processing are required to be centrally controlled? ____ N/A

c. Calculate b/a and enter score. ____ N/A

AM.1(2) a. How many error conditions are required to be recognized (identified)? ____ N/A

b. How many recognized error conditions require recovery or repair? ____ N/A

c. Calculate b/a and enter score. ____ N/A

AM.1(3) Is there a standard for handling recognized errors such that all error conditions are passed to the calling function or software element? Y | N | N/A

Figure 5.2-3 Metric Worksheet Part 1

METRIC WORKSHEET 1 CSCI LEVEL

AM.1(4)	a.	How many instances of the same process (or function, subfunction) being required to execute more than once for comparison purposes (e.g., polling of parallel or redundant processing results)?	___ N/A
	b.	How many instances of parallel/redundant processing are required to be centrally controlled?	___ N/A
	c.	Calculate b/a and enter score.	___ N/A
AM.2(1)		Are error tolerances specified for all applicable external input data (e.g., range of numerical values, legal combinations of alphanumerical values)?	Y N N/A
AM.3(1)		Are there requirements for recovery from all computational failures?	Y N N/A
AM.3(2)		Are there requirements to range test all critical (e.g., supporting a mission-critical function) loop and multiple transfer index parameters before use?	Y N N/A
AM.3(3)		Are there requirements to range test all critical (e.g., supporting a mission-critical function) subscript values before use?	Y N N/A
AM.3(4)		Are there requirements to check all critical output data (e.g., data supporting a mission critical system function) before final outputting?	Y N N/A
AM.4(1)		Are there requirements for recovery from all detected hardware faults (e.g., arithmetic faults, power failure, clock interrupt)?	Y N N/A
AM.5(1)		Are there requirements for recovery from all I/O device errors?	Y N N/A
AM.6(1)		Are there requirements for recovery from all communication transmission errors?	Y N N/A
AM.7(1)		Are there requirements for recovery from all failures to communicate with other nodes or other systems?	Y N N/A

Figure 5.2-3 Metric Worksheet Part 2

Neither factors nor criteria are disjoint (i. e., they share subordinate elements), and there is also correlation between metrics (e. g., in Figure 5.2-4, AM.6, recovery from transmission errors, and AM.7, recovery from failures to communicate). This makes it possible for a single deficiency in an element that is shared to be greatly amplified at the summary level. There is no evidence that this amplification coincides with the importance of the captured information to user concerns.

The framework has seen little or no use in its entirety. A consortium has been organized by Rome Laboratory to evaluate in which applications and life cycle stages the metrics are most effective. A tool has been developed that automates the score keeping and associated activities. At this time no results of these studies are available.

It does not appear practical to mandate use of the framework for V&V on high integrity software because (a) a substantial part of the development cycle may have been completed outside the V&V structure, (b) the possibly high cost associated with the use, and (c) current the lack of validation of the metrics against key requirements of high integrity software, primarily safety and reliability.

On the other hand, the metric worksheets (see examples in Figures 5.2-3 and 4) can in many instances be used as V&V checklists for the presence of desirable or required characteristics. As an example, AM.1(2) asks the following questions:

a. How many error conditions are required to be recognized (identified)?

b. How many recognized error conditions require recovery or repair?

c. Calculate b/a and enter score.

Since it makes sense that every recognized error condition be associated with a corrective action, the checklist can specifically raise this as a requirement, in effect forcing the score for this metric to be 100%. Selection will be required to isolate questions pertinent to high integrity software. The metric worksheets comprise approximately 180 pages, and examination of a practical software product to even a small fraction of them will exhaust a typical V&V budget. The selection should be based on anticipated sources of difficulties or errors; hence the need for data collection and analysis before meaningful checklists can be developed.

5.2.2 SEI Capability Maturity Model

Description of the SEI Model

Another framework for assessing and improving software quality is the Capability Maturity Model (CMM) for Software developed by the Software Engineering Institute (SEI). [PAUL91, WEBE91]. It provides an evolutionary strategy for software organizations to use in developing a mature, disciplined, software process. It covers practices for planning, engineering, and managing software development and maintenance. The CMM is based on established quality principles. It implements Total Quality Management for software, applying methods of statistical quality control.

The CMM casts these principles into a maturity framework. The framework consists of five maturity levels that describe the progression from a chaotic process to a well controlled, optimizing process. The levels lay successive foundations for continuous process improvement. By determining their position in this framework, organizations can readily identify the most fruitful areas for improvement actions. Each level establishes an intermediate set of goals toward higher levels of process maturity.

The following characterizations of the five maturity levels highlight the primary process changes made at each level.

1. Initial The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

2. Repeatable Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

3. Defined The software process for both management and engineering activities is documented, standardized, and integrated into an organization-wide software process. All projects use a documented and approved version of the organization's process for developing and maintaining software.

4. Managed Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled using detailed measures.

5. Optimizing Continuous process improvement is enabled by quantitative feedback from the process and from testing innovative ideas and technologies.

Metrics play an important role in CMM:

- Measurable goals and priorities for product quality are established and maintained for each software project through interaction with the customer, end users, and project groups.

- Measurable goals for process quality are established for all groups involved in the software process.

- The software plans, design, and process are adjusted to bring forecasted process and product quality in line with the goals.

- Process measurements are used to manage the software project quantitatively.

Process and product metrics are collected and analyzed based on their usefulness to the organization and the projects in accordance with the following guidelines:

- The metrics support the overall goals and objectives of the measurement program.

- The metrics and the data collection are consistent across the projects.

- The metrics are chosen from the entire software life cycle (i.e., both the development and post-development stages).

- The metrics cover the properties of the key process activities and major products.

- The specific data items to be collected, their precise definitions, the intended use of each data item, and the life-cycle control points at which they will be collected are defined.

- The metrics are selected to support predefined analysis activities.

Examples of collected data items include:

- estimated or planned versus actual data on software size, cost, and schedule;

- quality measurements as defined in the software quality plan;

- peer review coverage and efficiency;

- test coverage and efficiency;

- number and severity of defects found in the software requirements;

- number and severity of defects found in the software code; and

- number and rate of closure on action items.

It will be recognized that most of these are subjective evaluations and not easily automated. as was also the case in the Rome Laboratory Framework. For achieving quantitative process management the following procedures are recommended as part of the CMM concept:

- Quantitative product quality goals are defined and revised throughout the software life cycle.

- Quantitative process quality goals are established and tracked for the software project, software requirements, software design, software code, and software tests.

- When quality goals are discovered to conflict (one goal cannot be achieved without compromising another goal), the software requirements, software design, software development plan, and software quality plan are revised to reflect the necessary tradeoffs.

- The groups involved in the software process review, agree to, and work to meet the project's quality goals for its process and products.

- Process data are monitored to identify actions needed to satisfy the process quality goals.

- The quality of the project's products are compared against the product's quality goals on a regular basis.

- Corrective actions are taken by the groups involved in the software process when the quality measurements indicate process or product problems.

The measurements are collected and tracked with the following:

- A centralized database to store organization's metrics data.

- A uniform data definition is used across projects.

- Key cost and quality measures and analyses should be required at each major project milestone.

- Resources should be provided for gathering, validating, entering, accessing, and supporting the projects in analyzing metrics data.

The following tools for measuring, tracking, and analyzing quality metrics are recommended:

- data collection tools

- database systems

- spreadsheet programs

- life-cycle simulators

- statistical analysis tools

- code audit tools

Evaluation of the CMM Methodology

The CMM methodology is still evolving, and the following remarks apply to the procedures in use in mid-1992. The approach is primarily process oriented, and while a high quality process is a desirable environment in which to build a high quality product, there has as yet been no proof that control of the process will necessarily result in safe and reliable software. An important participant in this effort stated under the heading "Where's the return on product improvement?" [HERS93]:

> ... Product quality is the only thing we sell. The problem is, how do we know when [process] change is improvement?
>
> Finding a creditable answer to this question poses problems for software engineers that other industries seem to have solved. In manufacturing and hardware engineering, the value of changing processes can be proven with hard, projectable date. But -- despite our own firm belief in process improvement and our intuitive expectation that substantial returns will result from moving up the SEI scale -- we still can't prove it.

The CMM methodology does not measure the accuracy of translating system requirements into the software specification, and yet that is an area in which many of the most serious reported failures originated [NEUM93]. The CMM methodology does not directly address any of the areas of concern discussed in the preceding chapter, but it does not preclude tailoring some of the measurements to these.

To be useful in the high integrity software environment the CMM methodology must be supplemented by other metrics for assessment of product quality. The current CMM methodology concentrates on the development aspects between requirements and test. It is not very specific in the requirements and test areas, and these are of primary concern for the V&V of high integrity systems. Little direct benefit will therefore be derived from this methodology although there may be considerable indirect benefit in that the standards for software development for a broad segment of the software vendors will be raised by the application of CMM.

5.3 SURVEY OF SPECIFIC SOFTWARE METRICS

The preceding section discussed complete systems of metrics, while the present section is devoted to individual metrics that may be applicable to high integrity software. Table 5.3-1 shows a list of representative software metrics from which the ones in bold face have been selected for further discussion in the following subsections.

The metrics can be classified according to the program attributes which they measure: (1) program size, (2) control flow, (3) data flow, and (4) a composite of two or three of the above measures. At least one metric of each type is discussed below. None of these metrics directly addresses the areas of concern discussed in the preceding chapter, and none have been shown to have a strong correlation with operational reliability measures, such as failure rate.

5.3.1 Line of Code Measure

The most used measure of source code program length is the Number of Lines of Code. However without a careful definition of a line of code, there will be many ways in which this measure may be calculated, i.e. it is ambiguous. Examples of uncertainties in this measure are the counting of

- comment lines
- variable declarations
- a `line' with multiple statements.

The most widely used definition [CONT86] is:

> A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations and executable and non-executable statements.

Lines of code is an example of a size measure. In addition, it is a verify important normalizing factor, as in *fault density* which is the number of faults found in a software segment divided by the number of lines of code.

5.3.2 Halstead's Software Science Measures

The Halstead's Software Science [HALS77] is another measure of program size, primarily intended to capture the intellectual effort ("mental discriminations") required to generate the software. It considers a program P as a collection of tokens, classified as either operators or operands, from which primitive, intermediary, and comprehensive metrics are generated as follows:

Table 5.3-1 Software Metrics

Metric	Description	Category	Reference
1. **LOC**	lines of code	Volume	-
2. LOC-CMT	lines excluding comments	Volume	-
3. STMT	number of statements	Volume	-
4. UNIT	number of modules	Volume	-
5. STMT/U	avg. statements per module	Volume	-
6. nl*	unique operators	Volume	Hals77
7. n2*	unique operands	Volume	Hals77
8. N1*	total operators	Volume	Hals77
9. N2*	total operands	Volume	Hals77
10. n*	vocabulary	Volume	Hals77
11. N*	program length	Volume	Hals77
12. $\hat{N}$*	predicted program length	Volume	Hals77
13. V*	program volume	Volume	Hals77
14. E*	predicted effort	Volume	Hals77
15. **v(G)**	cyclomatic complexity	Control	McCa76
16. Gilb-N	number of binary decisions	Control	Gilb77
17. Gilb-R	decisions and stmts ratio	Control	Gilb77
18. Chen	max intersect of flow graph	Control	Chen78
19. Knot	knots count	Control	Wood79
20. FP	function points	Control	Albr79
21. Band	avg. level of nesting	Control	Bela80
22. Scope-N	scope number	Control	Harr81
23. Scope-R	scope-N and nodes ratio	Control	Harr81
24. Mebow	weights on linear flow graph	Control	Jaya87
25. NPATH	products of construct complexity	Control	Nejm88
26. Span	span of data references	Data	Elsh77
27. Slice	statements affecting a variable	Data	Long86
28. Q	index of difficulty	Data	Chap79
29. Live Var	avg. number of live variables	Data	Duns79
30. **Inflow**	(fan-in * fan-out)**2	Data	Henr81
31. DataStruc	structural complexity of data	Data	Tsai86
32. Rend-active	concurrently active redezvous	Comm	Shat88
33. Hansen	volume and control	Composite	Hans78
34. **Oviedo**	control flow and data flow	Composite	Ovie80
35. Henry	volume and information flow	Composite	Henr81
36. Basili	volume and control	Composite	Basi83
37. Li	Halstead E and SCOPE-R	Composite	Li87
38. Ramamurthy	Halstead and McCabe	Composite	Rama88
39. Shatz	local and communication	Composite	Shat88
40. van Verth	Oviedo and procedures	Composite	Vert87

* Components of the **Halstead** metric

Unique Operators: n_1
Unique Operands: n_2
Total Operators: N_1
Total Operands: N_2
Program Vocabulary: $n = n_1 + n_2$
Program Length: $N = N_1 + N_2$
Program Volume: $V = N \times \log_2 n$
Predicted Length: $\hat{N} = n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2$
Predicted Effort: $E = V \times \frac{n_1}{n_2} \times \frac{N_2}{2}$
Predicted Time: $\hat{T} = E \;/\; 64{,}800$
Predicted Bugs: $\hat{B} = V \;/\; 3{,}000$

The program volume V represents the size of an implementation, which can be thought of as the number of bits necessary to express it. The effort E is perceived as a measure of the number of mental discriminations required to implement an algorithm. The predicted time $\hat{T}$ is in units of "hours" and is calculated by dividing the effort by S, where S = 64,800 and is number of mental discriminations/hour. The predicted bugs, $\hat{B}$, is defined as the total number of "delivered" defects in a given implementation. It is calculated by the program volume divided by e, where e = 3,000 and is the mean number of mental discriminations between programming mistakes.

In the years immediately following its publication the Software Science metric was reported to be very effective in predicting fault density and other indicators of reliability. More recently the evaluations have been predominantly negative [BAIL81, JENS85, LIND89]. A possible explanation for this shift in validity of the metric as an indicator of reliability is the increased use of software development environments which reduce the "mental discriminations" required to keep track of variables. At the same time, the introduction of programming languages that provide formal exception handling, real-time constructs, and support multi-tasking, has introduced new requirements for intellectual effort that were not foreseen by Halstead. The effect of these combined changes is taken into account in a significant modification of the Halstead metric that is discussed in Section 5.4.

5.3.3 McCabe Cyclomatic Complexity Metric

McCabe's cyclomatic complexity metric [MCCA76] is a widely used metric to express the complexity of the control flow of a program. According to McCabe, a program unit is represented by a flow graph and its cyclomatic complexity is calculated by:

$$v(G) = e - n + 2p$$

where e is the number of edges,
n is the number of nodes,
and p is the number of connected components of a flow graph.

For program units with a single entry and a single exit, v(G) equals the number of decisions plus one, i.e.,

$$v(G) = \text{decisions} + 1$$

The metric can identify poorly structured code that will present difficulties during V&V. It has also been reported that it is a predictor of the total number of faults in a program, but in this regard it is not significantly superior to the lines of code measure which is more easily obtained. Because poorly structured code can also be identified by inspection, the use of this metric for its detection is not essential. It is occasionally used as a vehicle for controlling the structural complexity of code, e. g., by specifying a maximum value of v(G) for each module. Because of absence of a strong correlation between v(G) and software reliability such provisions are not advocated for high integrity software.

5.3.4 Henry and Kafura's Information Flow Metric

The Information Flow metric developed by Henry and Kafura [HENR81] indicates the complexity of a program by measuring the interconnectivity of the source program's components. It is based on the information-flow connections, called fan-in and fan-out, between a subprogram (i.e. procedures and functions) and its environment. Fan-in is the number of local flows into a subprogram plus the number of global data structures from which it retrieves information. Fan-out is the number of local flows from a subprogram plus the number of global data structures that the subprogram updates. The complexity for a subprogram is defined as:

$$C = (\text{fan-in} \times \text{fan-out})^2$$

Henry reported a statistical correlation of 0.95 between errors and structural complexity as measured by the Information Flow metric [HENR81]. Independent verifications are not available. Limited experimental use of the metric within SoHaR yielded inconclusive results.

5.3.5 Measure of Oviedo

Oviedo has developed a "Model of Program Quality" [OVIE80] which represents an example of a combination metric. Oviedo defines the complexity of a program by calculating the control complexity, CF, and data flow complexity, DF, as part of one measure. The control flow complexity of the flowgraph is the number of edges in the graph. The data flow complexity of a node is the total number the locally exposed variables reaching that node. Locally exposed variables are variables which are defined outside the block but can be referenced by that block (and therefore the reference requires going up the calling hierarchy to reach the definition).

Use of this measure as an indicator of reliability is not discussed in the literature. Because of the limited validity as indicators of the individual components of the model a fair amount of skepticism appears warranted.

5.4 NEW QUALITY METRIC

The metrics of most interest for high integrity software are product oriented and permit early identification of segments that may pose safety or reliability problems. In the preceding section no metrics were encountered that meet these goals. Therefore a new or, more appropriately, modified metric is proposed that shows promise to coming closer. The starting point is the Halstead metric that has been discussed in the preceding section. Its basic premise is that the fault content of a program will be a direct function of the intellectual difficulty posed by the function being implemented. As was pointed out earlier, the difficulty of just keeping track of operators and operands has been substantially overcome by contemporary programming environments. Therefore the number of tokens is no longer a good indicator of the "mental discriminations" required of the developer.

In current high integrity programs in general, and specifically for safety applications in nuclear power plants, the presence of real-time constraints and the prevalence of exception handling requirements (to deal with hardware, software, and communication failures) pose challenges for which as yet little automated support exists. Therefore it is suspected that an extension of the Halstead approach that focuses on the existence of real-time and exception handling constructs will be indicative of reliability problems.

Another deficiency of the original Halstead approach is that it does not utilize the association of functions and data that is the core of the object oriented approach and new programming languages. In the following, the Ada (TM, Department of Defense) language is used as an example, but the concepts discussed are not restricted to that specific language. Additional language features utilized are the tasking concept, intertask communications and formal support for exception handling.

A metrics study conducted on production-grade real-time software indicated that the extended Halstead metric (described in the following) has a much stronger correlation to software reliability problems found during testing than the original one.

The metric discussed here is primarily an indicator of reliability. High integrity software must also possess qualities of correctness, maintainability and verifiability. No new metrics could be proposed for these within the scope of the present effort. However, the Rome Laboratory framework, if transformed into checklists, can provide a starting point for systematic review for these factors.

5.4.1 Metrics Requirement for High Integrity Software

The primary function of a high integrity system in nuclear facilities is to integrate the data acquired by sensors into information useful for safety operations of the plant. These functions affect the characteristics of high integrity systems software in the following significant ways:

1. Hard deadlines: high integrity systems contain sampled-data feedback control loops (e. g., in analog-to-digital converters) and other functions (e. g., response to unsafe conditions) with rigorous response time requirements. The software is not only required to deliver correct results, but also must deliver them by the required deadline.

2. Error handling: in high integrity systems the software has to detect and recover from errors and other exception conditions.

3. Inter-task communications: the systems are required to handle asynchronous events raised by the sensors, devices, and displays simultaneously. The software handling these asynchronous events naturally consists of a number of tasks each handling one or more events. These tasks interact with each other to pass information and results.

The requirements of real-time, error handling and inter-task communications for high integrity software make its development and testing much more difficult and more prone to errors.

The conventional Halstead measures are based only on the syntax of the program text (operators and operands), without considering the semantics of the applications. Real-time software is generally more difficult to design, implement, test, and comprehend due to the interaction of concurrent processes and real-time constraints. However, according to Halstead measures, a sequential program and a concurrent program consisting of a similar number of operators and operands are of a similar complexity and hence require similar amounts of programming effort. The inadequacy of this approach for real-time programs is obvious.

Because Ada was designed to support the development of real-time embedded systems, it has introduced concepts such as tasking, exception handling, and intertask communication to support their software development [ICHB79]. The new language commands explicitly express these real-time aspects of software. This provides the opportunity to capture these characteristics and measure their complexity. The purpose of the new constructs is to enhance the productivity and reliability of software development. From the perspective of metrics, one benefit is that the new

constructs express the information needed to assess many of the real-time aspects of the software that contribute to complexity.

5.4.2 Implementation of Extensions to the Halstead Metric

Our extensions to the Halstead measures make use of the explicit declarations within Ada that support real-time operations. The following Ada constructs make the features of the application apparent through analysis of the program text:

1. Intertask communication - by the entry declarations and rendezvous calls.
2. Nondeterministic program behavior - by the select commands.
3. Real-time constraint - by the timed rendezvous, delay commands, and priority pragma.
4. Error handling - by the declarations of exception handler, abort, and terminate commands.

The operators and operands associated with these constructs are Ada Realtime (AR) operators and operands. Hence, the AR Halstead tokens consist of:

Unique AR operators: $a1$
Unique AR operands: $a2$
Total AR operators: $A1$
Total AR operands: $A2$

Instead of treating all the operators and operands in a program with the same importance as in the original Software Science metrics, the extended AR Halstead metrics will be formulated with weights added to the AR operators and operands.

Combined unique operators: $t1 = n1 + w \times a1$
Combined unique operands: $t2 = n2 + w \times a2$
Combined total operators: $T1 = N1 + w \times A1$
Combined total operands: $T2 = N2 + w \times A2$

where w is the weight added to the Ada-Realtime constructs, with a value determined by the metrics evaluation, and the n and N symbols are defined in Table 5.3-1. The extended AR Halstead Software Science metrics are:

AR Program Vocabulary: $n_a = t_1 + t_2$

AR Program Length: $N_a = T_1 + T_2$

AR Program Volume: $V_a = N_a \times \log_2 n_a$

AR Predicted Length: $\hat{N}_a = t_1 \times \log_2 t_1 + t_2 \times \log_2 t_2$

AR Predicted Effort: $E_a = V_a \times \frac{t_1}{t_2} \times \frac{T_2}{2}$

AR Predicted Time: $\hat{T}_a = E_a / 64{,}800$

AR Predicted Bugs: $\hat{B}_a = V_a / 3{,}000$

5.4.3 Metrics Evaluation

The evaluation focused on the correlation of fault density (taken as an indicator of reliability) and the metrics because it is believed that this represents the greatest need in high integrity software. If a strong correlation between the complexity metric and the fault density can be established, then the metric could be used to determine the amount of attention that the various software modules should receive during testing. The original as well as the extended Halstead metrics can be completely evaluated at the end of coding but reasonable estimates can be generated during design. Thus, the metrics represent a considerable advance in the development at which objective statements about reliability can be made. Another use of the metric is fault estimation. An accurate prediction of the number of residual faults is a useful indicator of the quality of the development process and the quality of the final product. Since most high integrity systems require very high reliability, the mean time between failures (MTBF) is far longer than any practicable interval of testing. Software complexity metrics have potential for estimating the lower bound for the reliability of the software under operational conditions, but this capability is untested at present.

The evaluation is performed using samples of software developed for a major U.S. air traffic control system. The software is written in Ada and involves real-time operations including exception handling. The software was selected for evaluation because it is a real-time command and control system and has many characteristics similar to high integrity systems software. It consists of 137 files with a total of 21,129 lines of code. Each file contains one compilation unit.

The functions of the software specimen are:

1. Detect and recover from processor failures within a group of redundant processors connected to a local area network.

2. Control the formation of a redundant group of processors upon initialization.

The first of these functions is representative of what might be found in safety system software, particularly if the processor failures are interpreted to include processor, sensor, and link failures. The second function is akin to a general health check of the plant and safety system which may be run as part of initialization. If representative safety system software using an advanced programming language (Ada, C, or C++ are examples) can be made available a similar evaluation should be undertaken to confirm the findings reported below.

The fault density, as measured by the number of programming faults per statement in a compilation unit, is being used rather than the number of faults because the latter may just indicate the size of the unit.

The fault statistics were collected manually from problem reports that were produced during the testing of the software. Although the reports do not pinpoint the location of the faults, they do contain descriptions of the errors observed during testing. This information, together with the functional description and revision history of the software units, has been used to relate errors to the units.

An automated tool measuring the original and extended Halstead metrics has been developed based on an Ada parser (source code analyzer) which was created by use of automatic parser generator tools [TSO91]. The tool distinguishes ordinary and AR tokens (terms in a source statement) and the AR weight w can be input in order to compute the extended Halstead metrics. The same tool was used in the evaluation of the original and extended metrics. For the latter case the assignment w = 10 was made. This weight was selected so that the AR metrics dominate in all modules where there are a significant number of AR operators, and that the conventional Halstead metrics remain operative where there are very few or no AR operators.

The statistics in Table 5.4-1 clearly show that Ada-Realtime code has a significant impact on the programming faults of the software. There are 31 compilation units containing AR constructs out of a total of 137, but 34 of a total of 47 programming faults are located in these AR units. This means that 72% of the faults occurred in 23% of the units.

Table 5.4-1 Characteristics of the Units

Type of Unit	No. Units	No. Faults
Non-AR Units	106 (77%)	13 (28%)
AR- Units	31 (23%)	34 (72%)
Total	137 (100%)	47 (100%)

Table 5.4-2 shows a breakdown of the compilation units according to the number of faults they contain. Most of the units (105 out of 137) have no faults. There are 11 units without AR

constructs that contain 1 fault and only one that contains 2 faults. For the units with AR constructs, the probability of containing faults is much higher (0.65 instead of 0.11 for those without AR constructs). Most of the multiple faults occurred in these units. The results confirm our expectation that real-time code is more difficult to develop and test and consequently more liable to faults.

Table 5.4-2 Number of Faults per Unit

No. of Faults	Non-AR Units	AR-Units	Total
0	94	11	105
1	11	12	23
2	1	5	6
3	0	1	1
4	0	1	1
5	0	1	1

Figure 5.4-1 shows a scatter graph of fault density against both the original and the extended Halstead potential bugs measures. The fault density is computed as the number of faults in a compilation unit divided by its number of logical source statements. The Halstead potential bugs (faults) metric is a measure of the total number of "delivered" defects in a given implementation. It is calculated by the Halstead program volume divided by the mean number of mental discriminations between errors.

For software units that do not contain AR constructs the data points for the conventional and AR metrics coincide, i.e. the symbols "diamond" denoting original Halstead and "cross" denoting extended Halstead overlap. For software units that contain AR constructs, the figure shows that their potential bugs measures have larger values - predicting more bugs in the software than the original - because these AR constructs cause more programming errors. In this figure, the weight for AR tokens is 10. A smaller weight would move the AR data points closer to the original.

The data does not show a relationship between the fault density and the original or extended Halstead potential bugs measures. The result is inconclusive for two reasons: 1) a large number of units without errors, causing many data points to lie on the x-axis, and 2) a number of small units each having one error, causing data points with high fault density values.

To provide a more meaningful analysis the small compilation units were eliminated. Most of these units contained the specification part of packages or subprograms. The rationale for the

exclusion is that complexity metrics are inherently not suitable for predicting the quality of very low complexity code. Also eliminated were two units that were exceptionally large compared to the rest because a valid basis for evaluating such units did not exist. The following discussion relates to the evaluation of the remaining mid-size compilation units.

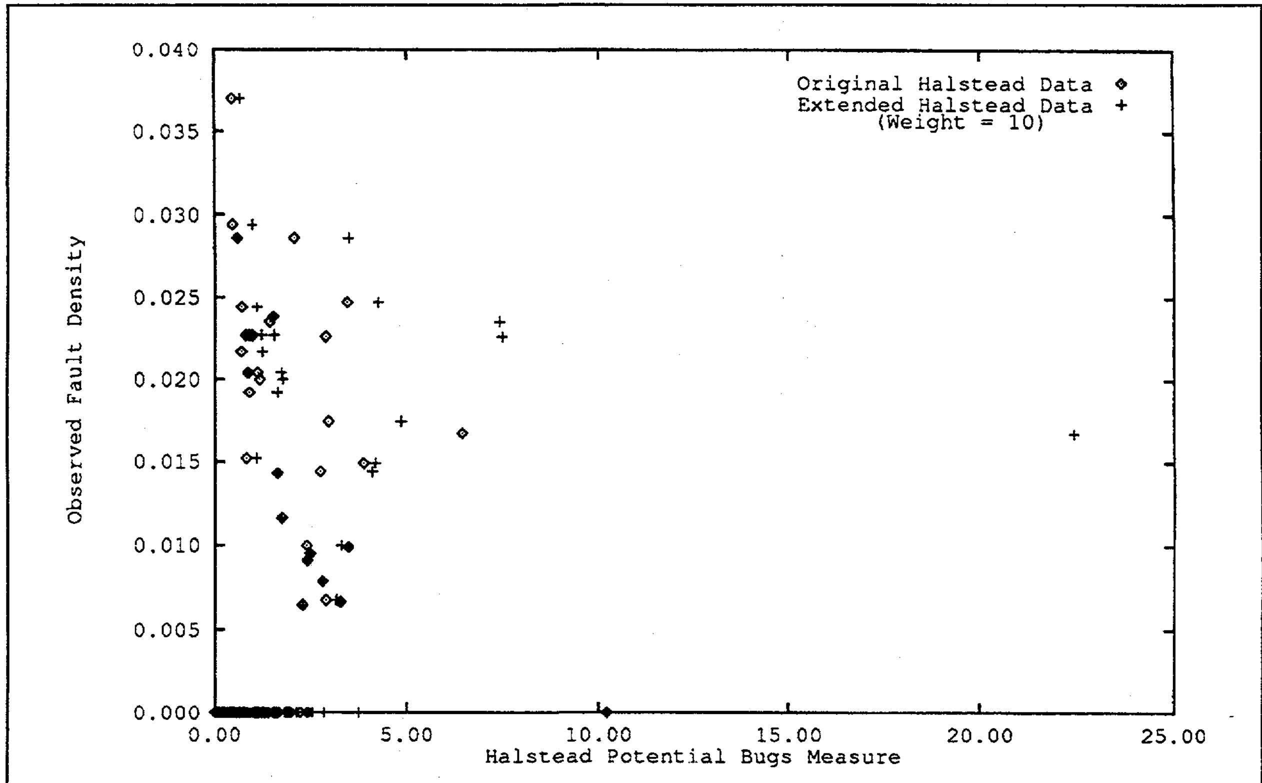


Figure 5.4-1 Failure Data of the Evaluated Software

Figure 5.4-2 plots the regression of the fault density against the original Halstead potential bugs measure. The correlation is very weak (with correlation coefficient 0.20). It can be concluded that the original Halstead bugs measure does not correlate well with the observed fault density.

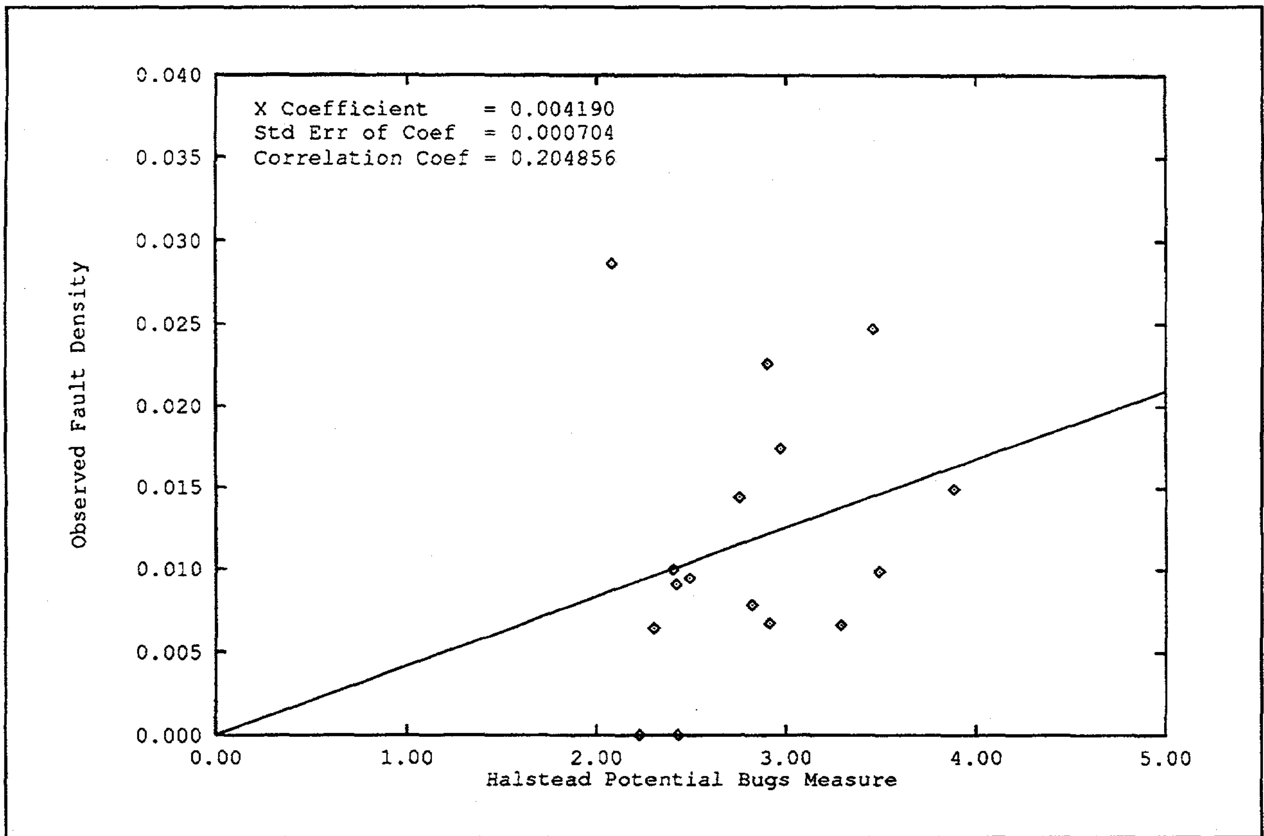


Figure 5.4-2 Regression on Halstead Metric

Figure 5.4-3 plots the regression of the fault density against the extended Halstead potential bugs measure. The correlation coefficient of the extended Halstead metric is 0.63, more than three times that of the original Halstead metric, and it has a much smaller standard error for the coefficient. For a dependable metric an even higher correlation coefficient is desirable. The main purpose of this example was to demonstrate that metrics should be selected to address error sources found in the specific environment.

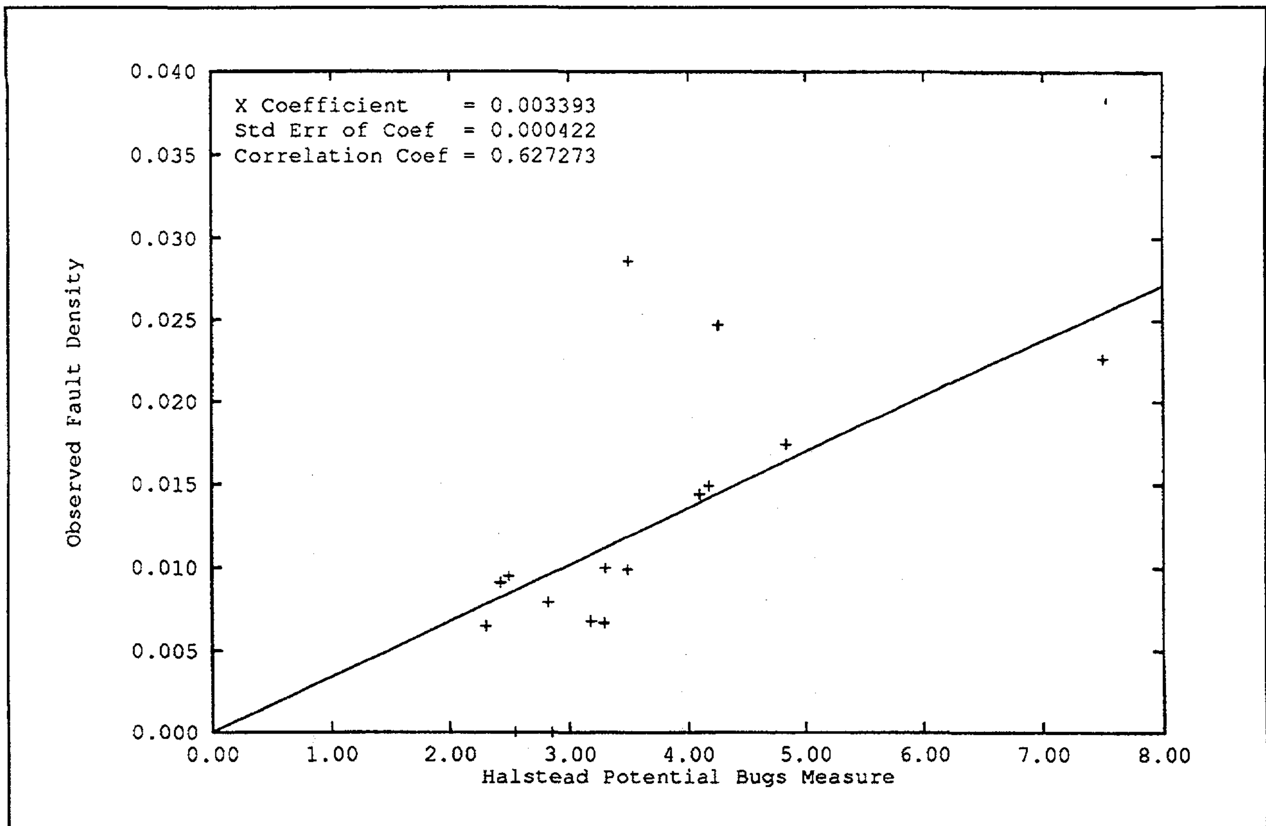


Figure 5.4-3 Regression on AR Halstead Metric

The following observations can be drawn from the metrics evaluation of the specimen software:

1. Ada--realtime (AR) constructs have significant impact on programming errors in software, as shown in the statistics of error reports among the compilation units.

2. The extended Halstead measure is very promising for predicting the potential failures of real time software.

3. The weight for the AR tokens that gave good correlation was determined to be ten times that of ordinary tokens. This may reflect that real-time software is an order of magnitude more complex than sequential software.

These results were obtained from one non-safety grade software project with small error counts. Additional investigations, preferably on safety system software, are needed to determine the generality of these observations.

5.5 CONCLUSIONS AND RECOMMENDATIONS

Metrics are desirable because they may furnish a quantitative and, it is hoped, objective assessment of software attributes that can at present only be characterized in a qualitative and subjective manner. Among currently available metrics none were found capable of meeting these objectives. Many current metrics are primarily intended for control or improvement of the software development process; these may be very beneficial for high integrity software in an indirect way, but there is no evidence that process control by itself can assure the quality of the delivered product.

The investigation has focused on metrics that are indicators of reliability because this is considered the most important single quality factor for high integrity software. The modified Halstead metric discussed in the preceding section has much higher correlation with fault density than the original metric in one application to real-time air traffic control software written in Ada. It is recommended that the suitability of metrics selected to address error sources for safety grade software be investigated by a demonstration effort on software segments for which test data from the final phases of development are available. Potential uses of such metrics are:

- alerting developers, users and the licensing agency to potentially troublesome software segments or major components; this will permit corrective action at an earlier stage than waiting for test results

- permit V&V efforts to concentrate on the most failure prone components

- after experience is gained with the metric it may be used in a proactive way to discourage the development of components that exceed a certain threshold value.

The most comprehensive framework of software quality metrics encountered in our investigation is that developed under the sponsorship of the USAF Rome Laboratory. Portions of the detailed metrics that are part of the framework may be converted into checklists to guide the V&V effort in the areas of correctness, maintainability and verifiability. The framework is not specifically directed at real-time programs, and any use made of it must consider the special requirements that arise from real-time constraints.

The greatest difficulty encountered in this effort has been the lack of metrics that can be obtained early in the development and that have demonstrated high correlation with relevant later metrics such as fault density or failure rate. While the quest for metrics valid during early life cycle phases should be continued, the aim of early correction can also be achieved by use of a spiral development approach, particularly when executed in accordance with the paradigm "build a little, test a lot". This will not only provide early indications of problem areas for a given software product but may also serve as a testbed for validating metrics in a specific environment.

SECTION 6 - VERIFICATION GUIDELINES

6.1 OVERVIEW

This chapter responds to paragraph 4.1.4 of the Statement of Work which states in part:

> Develop guidelines [for] ... the evaluation of verification programs. These guidelines shall address: (1) the verification plan; (2) the configuration management plan; (3) the adequacy of the design process used in developing the safety system; (4) the adequacy of the designer's error analyses, error tracking, and reliability evaluation based on qualitative measures; (5) verification of the specification against the requirements; (6) the adequacy of the diagnostic, fault tolerance and fail safe aspects of the design; (7) detection of unintended functions in the design; (8) use of computer aided software Engineering (CASE) tools in design verification; (9) verification efforts to detect unintended functions in hardware and software; and (10) the verification of commercial grade software in safety applications.

Because of partial overlap of these subject areas, and because of additional verification concerns that arose as the study progressed, the presentation of this chapter has been arranged in a different order. Table 6.1-1 points to the locations where the SoW subjects are discussed.

Table 6.1-1 Cross Reference to SoW Subjects

Sow Subject		Location(s) of Significant Discussion
No.	Short Title	
1	Verification Plan	6.2.3
2	Configuration Mgmt. Plan	6.2.4
3	Design Process Adequacy	6.3.1, 6.4.1, 6.4.3.2
4	Error Analysis and Tracking	6.3.2, 6.3.3, 6.3.3
5	Specification Verification	6.3.4, App. C
6	Diagnostics and Fault Toler.	6.4.1.1, 6.4.4
7	Unintended Functions (design)	6.1.1, 6.3.3, 6.3.4
8	Software Tools	6.5.2, 6.5.3
9	Unintended Functions (verif.)	6.1.1, 6.3.3, 6.3.4
10	Commercial Software	6.5.1

Verification is the process of determining whether or not the product of each phase of the software development process fulfills all the requirements imposed by the previous phase. Established standards and guidelines for verification assume that the verification activities will overlap the development, either being conducted concurrently or immediately upon completion of each phase. In the nuclear power field internal verification by the developer may follow this procedure, but the verification on behalf of the user (utility) or regulatory agency typically takes place after design is complete, and frequently after all code is implemented. Inherent in these scheduling differences are also significant differences in the organizational structure of the verification process. Where the sponsor (user) of a software development contracts directly for verification, he can select any degree of organizational independence desired. By contrast, in the U. S. nuclear industry the major part of the verification activities is usually funded by the developer and the verifiers are typically drawn from the development organization. An assessment of these differences in presented in Section 6.2. It is pointed out that performance of a significant part of verification after an overall system is essentially complete will preclude correction of any but the most serious deficiencies.

A major objective of this chapter is to bridge the gap between the environment in which conventional verification takes place and that prevailing in the U. S. nuclear power environment. For this purpose the available methodologies are examined in Section 6.3, and the time phasing of conventional verification activities is discussed in Section 6.4. Verification that is meaningful to the user and regulatory agency can be achieved in two ways:

1. By backward reconstruction (reverse engineering) which establishes that a completed software product satisfies requirements established in a preceding phase, or

2. By establishing standards for internal verification by the developer, and subsequently auditing the compliance with these standards.

Both of these avenues are explored, but each has distinct limitations that preclude a strong recommendation for sole reliance on it at this time. For reasons stated below, the combination of the two approaches is judged to offer a practicable basis for a comprehensive verification approach suitable for current procurement practices in the U. S. nuclear power industry.

Backward reconstruction is described in Section 6.3.3. Its advantages are that (a) it can be performed at any time, (b) it constitutes an inherently independent activity, (c) it has good potential for detecting unintended functions, and (d) it provides objective evidence of conformity with prior phase outputs. The disadvantages are that it is very labor intensive unless supported by tools (and currently available tools have only limited capabilities), and that decoupling from the development process may inhibit correction of all but the most significant deficiencies.

Specification of internal verification activities is discussed in Section 6.4 and subsequent auditing is discussed in Section 6.5.4. This approach has the advantage of concurrency with development

(for the internal auditing) and the resulting ability to promote positive improvement. The disadvantages are that it depends on subjective evaluations and lacks firm completion criteria.

The two methodologies can be combined by strong and independent auditing of internal verification activities, and also making tools for backward reconstruction available to the audit team. For this purpose further development of two promising and complementary tools is recommended.

Verification in the life cycle is covered in Section 6.4. Two scenarios are considered of which the first assumes interaction between development and verification teams throughout the development (as practiced by Ontario Hydro); this approach is potentially suitable for internal verification activities by U. S. developers. The second scenario is built on post-development verification that is suitable for use by utilities employing prevailing U. S. procurement practices and where verification is required during licensing procedures.

Section 6.5 deals with special verification concerns: commercial products and process audits. Conclusions and recommendations are presented in the final section of this chapter. Two tools that have potential for automating significant parts of the backward reconstruction activities are described in Appendices C and D.

This study has not found a valid technical basis for declaring any verification activities as "complete". This conclusion includes formal or algebraic methods that are discussed in Section 6.3.4. While these may lead to closure against a model of the real system generated in the process of formalization, there is no assurance that this model represents all software faiure mechanisms encountered in practice. As an extreme example, no known formal method protects against faulty configuration management. The study also found no objective evidence that the dependability of a software product is directly affected by the extent of verification to which it was subjected. A primary benefit of verification is that requirements, design, code, and the associated documentation are subjected to a critical review. At the very least this process establishes that the development products are reviewable (can be understood by non-participants in the development), are free from faults that are within the experience range of the reviewers, and are found to be suitable for the intended application by the reviewers.

The inherent subjectivity of verification activities also limits the value of checklists. While these can assure that relatively inexperienced reviewers do not overlook significant areas of difficulty, they carry a significant liability in implying that an adequate rating on a checklist equates to assurance of dependable operation.

The limitations of verification has led to the preference for functional diversity for applications demanding the highest integrity that has been stated in the introduction to this report.

6.2 ORGANIZATION AND PLANNING OF VERIFICATION

6.2.1 Requirements for Independence

A requirement that the verification be conducted by a person other than the implementer is almost inherent in the concept of verification. The degree of independence has been a matter of controversy in the nuclear field and also in related applications. Requirements for independence from several pertinent documents are reproduced below:

(1) IEEE/ANS 7-4.3.2 (1982):

> The verification group shall be independent* of those responsible for system design. The technical qualifications of the verification team shall be comparable to those of the design team. Communications between the groups shall be in written form.
>
> * See Section 4. of Supplement 3S-1 of NQA-1-1979 (equivalent to NQA2a, Part 2.7)

(2) IEEE/ANS P-7-4.3.2 Draft 7

> V&V shall be ... in accordance with NQA-2a-1990 Part 2.7. ... The V&V Plan shall specify activities and tests which shall be inspected, witnessed, performed or reviewed by competent individual(s) or group(s) other than those who developed the original design. The V&V Plan shall be reviewed by competent individual(s) or group(s) other than those who developed the plan.

(3) NQA2a, Part 2.7 (1990)

> Software V&V shall be performed by individuals other than those who designed the software.

(4) IEC 987 (1989) (nuclear safety systems) Par. 6.2

> Individuals or groups who perform the design verification shall be independent from those who are involved in the design activity. Persons involved with verification may be from the same organizations as the individuals responsible for the design.
>
> a) The skills of the verification personnel shall be similar to the skills of the people who carried out the design
>
> b) Communications between the design and verification personnel shall be formally documented to allow traceability of the verification activities

c) The persons responsible for the verification shall determine and document the level of verification to be performed.

(5) IEC 880 (1986) Par. 6.2

The management of the verification team shall be separate and independent from the management of the development team.

(6) RTCA DO-178B (flight critical software):

For software verification process activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified, and a tool(s) may be used to achieve equivalent coverage of a human verification activity.

(7) UK MOD 00-55 Clause 15

The Design Authority shall appoint a V&V team, independent of the design team, to verify and validate safety critical software.

(8) IEC 65A(Sec)123 Functional Safety of Electrical/Electronic/Programmable Systems: Generic Aspects (Draft 7, 1992) Clause 11, Functional Safety Assessment

LEVEL OF INDEPENDENCE	CONSEQUENCES		
	MARGINAL	CRITICAL	CATASTROPH.
Indep. Person	HR#	NR	NR
Indep. Department	HR*	HR#	NR
Indep. Organization	-	HR*	HR

Legend: HR = highly recommended, - = not recommended for or against, NR = not recommended. HR* applies to programs of high complexity, of new design, or using new technology. In all other cases HR# is applicable.

IEEE Std. 1012, Verification and Validation Plans, requires description of the reporting relationship of the V&V function to the design function but contains no recommendation on the degree of independence.

6.2.2 Discussion of Requirements

The requirements listed above are derived from two distinct procurement styles:

A. The user explicitly pays for the development of the system

B. The user procures a system without specifically paying for or controlling the development.

The UK MOD 00-55 standard, designated as (7) above, is typical of procurement style A. Since the procuring agency controls the development, it can appoint any agent it selects to conduct the V&V, and it will also separately pay for the V&V effort. Procurement style B. is typical of the nuclear power industry and is exemplified by documents (1) - (3) above which make the developer responsible for the independent verification but require only that the person(s) responsible for the verification not be involved in the development. IEC 880, document (5) above, goes a significant step further and requires that the verification group not report to the same management as the design group. An interesting alternative is presented by document (8), which comes out of the process industry, by requiring the degree of independence to be proportioned to the consequence of failure.

A survey of developers conducted by SoHaR in 1991 for the U. S. Nuclear Regulatory Commission found almost exclusive reliance for verification on in-house personnel, in many cases within the development organization. This met the minimum requirements of (1) - (3) because the individuals did not directly participate in the development of a given program although they might have worked on the development of related programs. The reasons given for this practice were:

- verification takes place intermittently during the development, and it is inefficient to bring in personnel from outside the department for these activities

- it takes too long for outsiders to achieve the familiarity with the development environment and terminology that will permit them to become effective verifiers

- the financial condition of the developers severely limited the verification budget and made it incumbent to use available (in-house) personnel.

In prior efforts SoHaR has audited software that had been verified by in-house personnel and has not found gross deficiencies that were due to this practice. Problems were uncovered in documentation procedures that would probably have been caught earlier if verification had involved personnel with a broader exposure to industry standards. In March 1993 the staff of the Atomic Energy Control Board (Canada) stated to the authors of this report that the outside review of the Darlington software (under the direction of Dr. D. L. Parnas), which may be likened to a highly independent verification, likewise found no "show stoppers". But it did identify over 60 instances in which code deviated from requirements (in ways that did not directly impact major functions), did not conform to good design practice, or violated language standards. These deficiencies should not be considered insignificant because serious failures can result from the coincidence of responses to multiple flaws, each one of which by itself would not have impacted the system.

Aside from the inferences drawn from the above, the authors of this report have not found any evidence that shows that a high degree of organizational independence of the verifier results in

more reliable software. One area of verification concern arises from increased use of tool and software engineering environments. While tool usage is highly desirable for both development and verification, the use of the *same* tool in both activities may permit problems to be overlooked. Therefore a requirement for independence of tools used for development and verification is recommended; such a requirement is implicit in the definition of independence in RTCA DO-178B, document (6) in the previous subheading.

A recommendation for independence of the verification team in the nuclear power context should consider that:

- the utilities and the NRC usually have little interaction with the developer in the early phases of software development; therefore their participation in conventional verification in accordance with IEEE Std. 1012 (see Section 2.3) will not be possible

- there is increasing use of previously developed software (including commercial) for which most verification steps are impracticable

- U. S. developers will resist requirements for verification by independent organizations unless they will be directly reimbursed for it

It is therefore recommended that the provisions for independence of IEEE/ANS 7-4.3.2 (either version) be accepted, subject to the following:

- the utility (or the NRC) arrange for auditing of the verification process by an independent organization; the audit shall follow the procedures for process audits described in Section 5.4.

- availability of complete failure and corrective action reports from factory test, installation at the site, and operational failures from similar installations.

The latter requirement arises because any failure reduces confidence in previously conducted verification.

6.2.3 Verification Plans

Planning of verification activities (i) identifies all steps required to achieve the objectives of the verification program, including regulatory approvals or acceptance, (ii) schedules the activities and coordinates with software development, (iii) establishes interfaces with associated activities, such as SQA and configuration management, and (iv) assures availability of required resources. A widely used guide for the preparation of verification and validation plans is ANSI/IEEE Std. 1012 (1986). An outline of the plan described in that document is shown in Table 6.2-1.

Table 6.2-1 Outline of V&V Plan from ANSI/IEEE Std. 1012

1. Purpose
2. Referenced Documents
3. Definitions
4. Verification and Validation Overview
   - 4.1 Organization
   - 4.2 Master Schedule
   - 4.3 Resources Summary
   - 4.4 Responsibilities
   - 4.5 Tools, Techniques and Methodologies
5. Life Cycle Verification and Validation
   - 5.1 Management of V&V
   - 5.2 Concept Phase V&V
   - 5.3 Requirements Phase V&V
   - 5.4 Design Phase V&V
   - 5.5 Implementation Phase V&V
   - 5.6 Test Phase V&V
   - 5.7 Installation and Checkout Phase V&V
   - 5.8 Operation and Maintenance Phase V&V
6. Software V&V Reporting
7. V&V Administrative Procedures
   - 7.1 Anomaly Reporting and Resolution
   - 7.2 Task Iteration Policy
   - 7.3 Deviation Policy
   - 7.4 Control Procedures
   - 7.5 Standards, Practices, and Conventions

The core of the standard is the listing of activities that shall be conducted in each of the Phases (5.2 through 5.8). The recurrent requirements are for traceability analysis (relating the product of the current phase to requirements established in the preceding one), and interface analysis (relating the inputs required by one component to outputs furnished by another one). It also emphasizes early consideration of test, starting with a system test plan to be generated during the requirements phase. The standard contains tables and figures that list (i) the inputs for and outputs of the V&V activities in each phase, and (ii) the relations between the V&V activities of successive phases. These are not reproduced here because of their limited applicability to the software development environment for nuclear safety systems, because they address a strict waterfall development schedule, and because they do not clearly distinguish between verification and validation activities. Material that is applicable has been incorporated into treatment of the life cycle activities (Section 6.4).

The standard is of greatest benefit to software development where the user (sponsor) exercises control from beginning to end, a condition not met in the typical procurement of nuclear plant protection systems or software. Adherence to the standard permits deficiencies to be addressed

in early life cycle phases when they are usually much easier to correct than later. Post-development verification may identify as many deficiencies as that conducted by concurrent verification, but it will seldom lead to correction of any but the most serious ones. This is an inherent limitation of post-development verification that must be recognized by the user as well as by regulatory agencies.

Regardless of the time at which it is to take place, planning for verification for nuclear plant protection software can be guided by the outline of the plan (Table 6.2-1), with major tailoring required only for the allocation of activities to life cycle phases. Verification activities must be conducted in accordance with a plan that defines responsibilities, tasks, and completion criteria for each task.

A standard for software verification and validation is discussed in NIST Special Publication 500-204, "High Integrity Software Standards and Guidelines". The standard implies a plan that includes the following activities:

- traceability analysis
- evaluation of development products by analysis, review and audits
- separation of test types (unit, integration, system, acceptance)
- test documentation
- management of V&V
- review of V&V products (e. g., test results)

The emphasis on review of development products (as distinct from review of the development process) is very pertinent to the nuclear safety systems environment. Products of the development process can be expected to be available even for commercial software, where the process itself is inaccessible. The Ontario Hydro (OH) verification guidelines described in Section 6.4.1 are essentially consistent with these requirements.

6.2.4 Interfaces with QA and Configuration Management

Because V&V and QA both employ independent personnel to conduct reviews or audits it may appear that there is much duplication in their work. In fact there is in most cases little overlap and considerable complementation even where the V&V proceeds alongside the development (there is practically no interfacing where V&V is conducted as a post-development activity). QA audits of software development are directed toward compliance with internal standards and procedures, whereas the purpose of the verification activities is to determine that the product requirements of each development phase are being met. Normally the internal standards will support the generation of satisfactory products, but by no means do they assure it by themselves. Therefore the V&V team will usually require access to QA records that indicate

- the type of inspections that were conducted in each phase

- the standards or other guidelines that were used

- deviations or anomalies found by QA personnel
- evidence that corrective action had been taken and was successful.

It will be seen that the verification of nuclear safety grade systems is heavily dependent on documentation produced as part of the development; therefore the quality of the documentation is vital to effective verification, and QA of the documentation is very important in this context. QA will benefit from having access to the results of the V&V activities, specifically if negative V&V findings were due to:

- non-compliance with internal or referenced standards (inadequate QA)
- deficiencies of the internal standards relative to the requirements of the target software
- deficiencies in tools, support software, or commercial software used as part of the delivered product.

The interfaces with the configuration management (CM) activities are usually well understood but not necessarily well enforced. There is anecdotal evidence that lapses in configuration management have occurred in both the developing and the user organizations. The V&V team requires that there exists a Configuration Management Plan and that the products (software and all documentation) submitted to it are:

- the latest releases and are under configuration control
- kept updated with all authorized changes
- the correct version (applicable to the specific site for which V&V is being conducted)

Requirements are a significant product subject to configuration control. It is important to be able to identify changes in requirements after completion of any verification steps.

Configuration management will want to be aware of deficiencies found in V&V that will require changes to the software or documentation, or that might restrict the use of the software (requiring a separate version designation). Because of the heavy dependence of the verification activities on documentation, the configuration management of documents is at least as important as that of the operational software.

6.3 VERIFICATION METHODOLOGIES

Verification has been defined as "The process of determining whether or not the product of each phase of the software development process fulfills all the requirements imposed by the previous phase." The following subsections describe methods that can be used in this determination, including

Reviews and Audits
Independent Equivalent Activity
Backward Reconstruction
Algebraic Methods

6.3.1 Reviews and Audits

Review is defined in [DORF90] as

1. A formal meeting at which a product or document is presented to interested parties for comment and approval. It can be a review of the management and technical progress of the development project.

2. The formal review of an existing or proposed design for the purpose of detection and remedy of design deficiencies that could affect fitness for use and environmental aspects of the product, process, or service, and for identification of potential improvements in performance, safety, or economy.

The reviews conducted as part of verification usually go much beyond a "meeting" and thus the second definition seems more appropriate. The specific purposes of the review in this case are (a) tracing all significant features of the current design phase to requirements established in the preceding phase, and (b) verifying that inputs required at interfaces correspond to outputs of the interfacing component. The term "design" must be interpreted broadly, to encompass the realization of the software at any time of the development cycle. As part of this activity the reviewer may partially duplicate some of the activities of the design team. If alternate means are available for generating a given analysis or report, the reviewer will be well advised to use them. As an example, if the design team used Tool A to generate a static analysis, the verification team may want to use Tool B.

The expected output of a review conducted as part of verification is a report that describes:

- the scope of the review (products or processes reviewed)

- the extent of independent analysis or design (products or partial products independently generated)

- the means used to verify analyses and design steps where independent generation was not employed

- acceptance of the design, subject to removal of stated deficiencies

- limitations of the review that may impact the safety or other suitability of the design.

Audits are an activity of narrower scope, usually aimed at verifying that the phase being audited uses appropriate requirements, that the design process, including tool usage, complies with good engineering practice and applicable safety system standards, and that the output (including intermediate steps) is properly documented. An audit can usually provide an assessment of the traceability that is equal to that obtained in a review, but may not be able to provide the same degree of assurance that design activities yielded correct results. Audits do not usually include independent generation of significant design products (they may include spot checks), and therefore a section dealing with independent analyses or design is not required in an audit report. In all other respects the format listed above for reviews will be applicable. Audits loose much less of their effectiveness than reviews by being applied retrospectively, and therefore they are a suitable format for assessment of design and coding in the typical nuclear safety software environment (where the software is essentially complete at the time of the assessment).

Audits and reviews can be applied to the software development process as well as to the product. The audit format is usually more appropriate for process assessment, because the main emphasis is on availability and suitability of tools, documentation of the design methodology, and qualifications of personnel. Procedures for process audits are described in Section 6.5.4.

6.3.2 Independent Equivalent Activity

The verification team may be tasked with independently conducting selected activities that are part of the design responsibilities. Examples are requirements analysis, generation of a hazards analysis, or preparation of test plans. These activities differ from the independent analysis or design that was mentioned above as a component of review in that

- it is of much wider scope, typically encompassing the entire software product

- the selection of activity and scope is dictated by the sponsor, whereas the independent activities conducted as part a review are usually selected by the verification team.

The independent equivalent activities are obviously costly, and the decision to require them is therefore usually based on a specific need, such as

- novel or unique features of the plant protection system

- software development or test methodologies that have not had extensive prior usage

- concern about the experience of the developer or a subcontractor in the plant protection field.

When independent equivalent activities are undertaken it is highly desirable that the tools and methodologies differ from those used by the design team.

6.3.3 Backward Reconstruction

This methodology involves reverse engineering applied to a product of the development process, and determining that the reversal of one or more completed steps yields the equivalent of the input to the step(s). This is a very powerful verification technique because (a) it addresses the traceability objective of verification in a very direct manner, (b) it is inherently highly independent of the design activities, and (c) it is effective in identifying unintended functions. The major disadvantage is that backward reconstruction is very labor intensive and hence expensive unless it is accomplished by automated tools.

As part of this research two tools have been identified which, when used in succession, span the most significant phases of software development: from software requirements to test. They are the ECT (Enhanced Condition Table) tool developed by SoHaR, and CATS (Code Analyzer Tool Set) developed by TÜV Norddeutschland.

ECT reconstructs condition tables from the source code, and these can be compared with condition tables used to formalize the software requirements. The condition table format is an engineering methodology for a formal representation of requirements, and has been selected by Ontario Hydro for their future work [JOAN93]. It was also used by Dr. D. L. Parnes as part of the review of the Darlington reactor software [PARN91]. A description of the Condition Table methodology and of the ECT tool is found in Appendix C.

CATS is applied to object code and furnishes structural and syntactic analysis which can then be compared with the structure and syntax of the source code. A description of the tool and examples of its output are presented in Appendix D. The use of CATS safeguards against outright errors in the compiler and also against undesirable restructuring of the code that may be introduced by the optimizing feature. It is not unusual for an optimizing compiler to generate object code having multiple entries or exits from source code that has a single entry, single exit structure. CATS also analyzes the data utilization of the programs, including number of local and non-local variables, number of uses, stack heights, and buffer margins. CATS can work directly from the output of a ROM, thus safeguarding even against errors introduced in the loading and manufacturing processes.

Use of the ECT depends on personnel skills primarily in the formulation of the condition tables that represent the requirements (this effort may be regarded as requirements analysis rather than related to the use of ECT), and in the comparison of the condition tables generated from the code with those representing the requirements. CATS depends on personnel skills to compare the structure generated from the object code to that of the source, and also to evaluate deficiencies

in data usage or interfaces. Both tools automate very significant portions of the verification process and therefore reduce its cost.

Neither one of these tools is as yet on the commercial market, but they have been in use by their developers for several years. They also are limited in the languages that they can accept: ECT is currently available only for C and Ada, and CATS is available for Intel 8048, 8051(family), 8086, 80186 (partial), 8096, and NEC 75x microprocessors. ECT can be extended to other languages by adding suitable parsers, a comparatively small task. The extension of CATS to higher performance processors, such as the Intel 80286, 80386, and 80486 processors is difficult because these incorporate dynamic memory mapping so that the location of a given instruction cannot be uniquely determined. A complete resolution of these problems may require considerable research, but most of the CATS capabilities for the higher level chips can probably be achieved in the near term by manually supplying starting addresses for each module. While the potential for using these tools at a late stage in the development is an advantage in the environment in which nuclear safety systems are currently being procured, it must be recognized that at that stage only the most serious deficiencies will be corrected, and that exclusive dependence on post-development verification activities implies a high likelihood accepting software with known weaknesses.

In spite of these current limitations, the ECT and CATS tools provide capabilities that make them uniquely suitable for the verification of nuclear safety software:

- they use only software products that are highly likely to be available: requirements documentation, source code, and executable code

- they require no interaction with the developer other than resolution of problems that are detected

- they provide a high degree of independence of verification from the design process, *regardless of the organization that performs the verification*

- they furnish objective indications of deficiencies.

It is therefore recommended that further research and development in this area be carried out in order to arrive at a verification methodology that is practicable in the environment in which safety software for nuclear plants is currently being procured and which is inherently objective and automated.

6.3.4 Algebraic Methods

Algebraic notation is unquestionably a very precise way of defining a requirement, and it permits use of the entire algebraic tool kit for transforming the posed requirement into one or more requirements for which solutions have already been found. A statement of software requirements in the form of algebraic notation provides these benefits and also has the potential for facilitating

automated generation of the design and code, e. g., by combining library code implementations of the solution fragments.

Formulation of requirements as condition tables (as discussed above) represents one form of algebraic notation because the individual conditions are usually expressed as equalities or inequalities, e. g., $x > A$, $x = A$, $x < A$. In practice it will be found that some parts of the requirements are readily translatable into an algebraic form, while others require considerable effort. An evaluation of the current state of this methodology by one of the leading researchers follows [PARN93]:

> Mathematical techniques can be used to produce precise, provably complete documentation for computer systems. However, such documents are highly detailed, and oversights and other errors are quite common.

Other investigators have concluded that

- there is inevitably a gap between the textual statement of a requirement and its algebraic representation [RUSH92]

- data presented to demonstrate the benefits of formal methods are frequently flawed [FENT93]

- the most consistently reported benefit is "tutorial" -- creating awareness of the capabilities of algebraic techniques -- rather than specific product quality enhancement [CRAI93].

On the basis of these assessments it is concluded that the use of algebraic methods for verification should be encouraged but not mandated. Considerable research and experimentation will be required before they can be depended on as a comprehensive approach.

As part of related efforts SoHaR has consulted several experts in the field who are strongly in favor of further research in formal or algebraic methods but who see currently no more than a supporting or selective role in their application to large real-time programs.

6.4 VERIFICATION IN THE LIFE CYCLE

An overview of verification activities over the life cycle was presented in connection with verification planning in Section 6.2. A specific implementation of the life cycle activities for certification of nuclear safety systems is included in the Ontario Hydro software engineering standard [1990] and is discussed in Section 6.4.1. The activities specified there are applicable in the Canadian environment where the utility and the system vendor collaborate much more closely during software development than is typically the case in the U. S. However, that format is also suitable for internal verification activities conducted by the developer, and these activities can then be subjected to a process audit (see Section 6.5.4).

Section 6.4.2 covers a modification of the OH approach that is applicable to verification by (or under the auspices) of a utility in the procurement environment prevailing in the U. S. That approach can also be used for verification as a part of licensing where the internal verification conducted by the developer is not considered adequate.

Verification frequently emphasizes compliance with applicable procedures and standards for design, coding, and test. Certain minimum requirements in that area have to be met for the code to be reviewable, maintainable, and capable of being analyzed by tools. However, extensive allocation of resources for the procedural aspects of verification can detract from hazards identification and reduction, particularly with regard to the areas of concern identified in Chapter 4. The authors of this report have therefore emphasized in the following those verification activities and documents that address hazards identification and reduction.

6.4.1 The Ontario Hydro Life Cycle Activities

The relation between development and verification activities that is assumed in the OH life cycle is shown in Figure 6.4-1. Verification and development are seen as concurrent activities, a scenario which is not usually applicable to the U. S. environment. However, an important feature of the OH methodology is that it is completely based on documentation and does not require interaction with the development process proper. Thus a considerable fraction of the activities described here can be transferred to verification of a developed product. The following description highlights portions of the verification that are most pertinent to the concerns described in Chapter 4.

The inputs and outputs of the activities are shown in Table 6.4-1. The most important activities and documents are described later. The acronyms used in the input and output columns are:

CRR	Code Review Report	RRR	Requirements Review Report
CVR	Code Verification Report	SDD	Software Design Description
DID	Design Input Description	SPH	Standards and Procedures Handbook
DRR	Design Review Report	SRS	Software Requirements Specification
DVR	Design Verification Report	UTP	Unit Test Procedure
HAR	Hazards Analysis Report	UTR	Unit Test Report

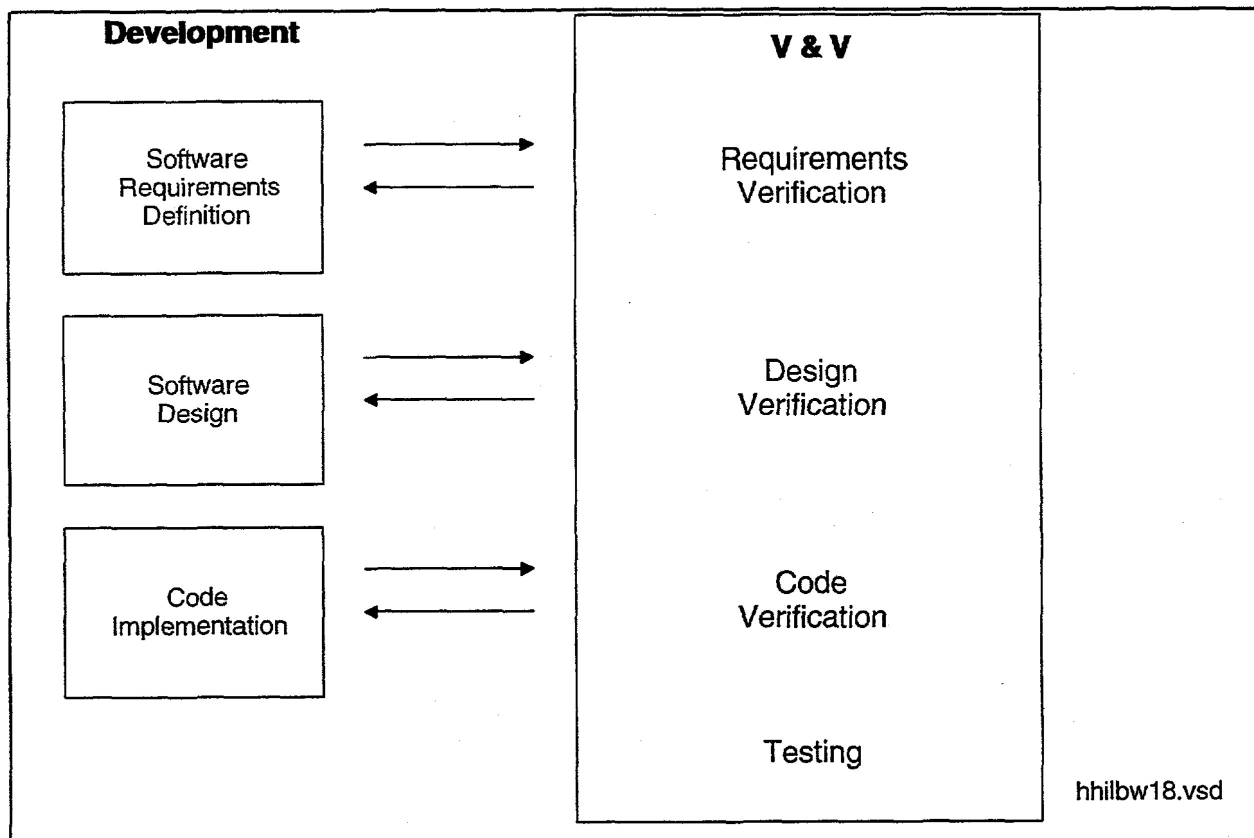


Figure 6.4-1 Development and Verification Interfaces in the OH Environment

Table 6.4-1. OH Life Cycle Verification Activities

Phase	Activity	Input	Output
Req'mts	Software Requirements Review	SRS, SPH, DID	RRR
Design	Software Design Review	SDD, SPH, DID, SRS	DRR
	Systematic Design Verification	SDD,SRS	DVR
Code	Code Review	Code, SPH	CRR
	Systematic Code Verification	Code, SDD	CVR
	Code Hazards Analysis	Code, DID, SRS, SDD	HAR
Test*	Unit Test	Code, SDD	UTP, UTR

* Additional test activities are listed that are outside the scope of verification

6.4.1.1 Software Requirements Review

The main purpose of this review is to determine that the Software Requirements Specification (SRS) accurately and completely implements the requirements identified in the Design Input Document (DID). The latter is the OH designation for the system level specification and its content is described later. Secondary purposes of this review are (a) to look for inconsistencies or omissions in the DID, and (b) to assess compliance of the SRS with the procedures of the Standards and Procedures Handbook (SPH)[8].

The DID contains 14 major headings of which the following are particularly pertinent to the verification issues discussed in the introduction to this section:

1. Partition the system into critical and non-critical subsystems and establish isolation between them

2. Define the functional, performance, safety, reliability, and maintainability requirements for each subsystem

3. Define interfaces with external inputs and outputs

4. Define accuracy requirements and tolerances

---

[8] This is the nomenclature used by Ontario Hydro. In other organizations it may be referred to as Software Procedures Manual; occasionally, there may be a separate Verification and Validation Procedures Manual.

5. Define all failure modes (of the external system) and the required response

6. Provide a clear definition of terms and identify requirements that conflict with one another.

The key objective for the SRS (the development team output at the end of the requirements phase) is to establish acceptance criteria for the design and coding phases in the areas of safety, functionality, performance, reliability, maintainability and reviewability. Approximately 50 detailed criteria for the SRS are listed, of which the following are judged to be the most important ones:

1. Contain or refer to all DID requirements relevant to safety, functionality, performance, reliability, and maintainability

2. Identify the physical variables that the software must monitor and control and represent them by mathematical symbols

3. Use mathematical functions to describe the behavior of the controlled variables in terms of the monitored variables

4. Define the required response to all types of errors and failure modes identified in the DID

5. Describe software reliability requirements consistent with the subsystem reliability requirements identified in the DID

6. Define requirements for fault tolerance and graceful degradation.

7. Demonstrate mapping of DID to SRS requirements by a coverage matrix or similar technique

8. Uniquely identify each software requirement so that it can be referenced in the System Design Description (SDD)

The key objectives of the Requirements Review Report (RRR), which is the output of the verification team for the requirements phase, are to provide *objective evidence* that the review has covered

- all requirements in the DID and the SRS

- (and identified) all requirements in the SRS that are not derived from the DID

- all standards and procedures applicable to the SRS

6.4.1.2 *Design Verification Report*

Two verification activities are identified in the table during the design phase: software design review and systematic design verification. The first of these is a conventional design review without specific verification import. Therefore only the systematic design verification and the resulting Design Verification Report (DVR) are described here. The objectives of the systematic design verification are to:

- verify, using mathematical techniques or rigorous arguments, that the behavior of every output defined in the SDD complies with the behavior of that output specified in the SRS

- identify functions outside the scope of the SRS that are provided in the design and check the justification

- identify ambiguities or incompleteness of the SRS

The inputs to the systematic design verification are the SRS and the SDD. The content of the former has already been described. The purpose of the SDD is to provide a complete description of the software design so that the resulting code will be fully compliant with the SRS. Among the extensive requirements levied on the SDD the following are particularly pertinent to verification:

1. The design must precisely meet the functional, performance, safety, reliability, and maintainability requirements and all design constraints as described in the SRS

2. Define types of errors (which are not specified in the SRS) and their handling

3. Show the hierarchical relation of all program components for each subsystem and define their interfaces

4. Define the function of each program for the full domain of all program inputs

5. Show that programs provide the required response to all error conditions

6. Schedule computer resources with only minimal dependence of interrupts

7. Define execution time plausibility (reasonableness) checking

8. Describe the function of each program in a notation that has a defined syntax so that the SDD can be systematically verified against the SRS and the code can be verified against the SDD

9. Provide periodic re-initialization of variables to facilitate trajectory based random testing

10. Demonstrate the mapping and complete coverage of all requirements in the SRS by means of a coverage matrix or similar technique

The SDD discussed above is the output of the *development team* during the design phase but it contains many provisions that support verification activities, particularly 8. and 10.

The requirements for the DVR, which is the output of the *verification team,* can therefore be reduced to demonstrating that the verification process has been complete. Specifically, it is required that the DVR provide *objective evidence* that the verification has covered all

- requirements in the SRS and all programs, data structures, and databases in the SDD

- justification for inclusion in the SDD of functionality outside the requirements in the SRS.

6.4.1.3 Code Verification Report

Table 6.4-1 lists three activities during the coding phase: code review, systematic code verification, and code hazards analysis. The code review is primarily concerned with the compliance of the code with software engineering practices identified in the appropriate manual. As implied by its name, systematic code verification is a specific verification activity, and the resulting report, the CVR, is discussed here. A further coding phase document that is relevant to verification, the Code Hazards Analysis, is covered under a separate heading.

The objectives of the systematic verification are:

- to verify, using mathematical techniques or rigorous arguments, that the behavior of outputs as a function of inputs is as specified in the SDD over the entire input domain (this objective probably reflects the experience of OH in the licensing of the Darlington reactor protection system, but it is expressed terms that allow fairly wide choice of the mathematical technique).

- to identify ambiguities or incompleteness in the SDD.

The inputs to the systematic verification are the SDD and the code. The content of the former has been described above. Among the many requirements levied on the code the following are particularly pertinent to verification. The code shall

1. precisely implement the design (in the SDD) and create all required databases and initial data values

2. insure that the accuracy requirements of all variables are met by the implemented data types and algorithms

3. not contain self-modifying code or recursion and rely primarily on static control parameters (limits on loops, constants in branch tables)

4. not rely on defaults of the programming language

5. provide protection against detectable run-time errors such as index out-of-range and stack overflow

6. employ only implementations that are easy to test

7. employ only single entry and single exit for programs (except for fatal error handling)

8. provide a cross-reference framework through which the code can be directly traced to the SDD

9. define the valid range for each variable

Here, again, the documentation provided by the *development team* contains much information that facilitates verification. Therefore the output of the *verification team* can be reduced to demonstrating that the verification process has been complete. Specifically, it is required that the CVR provide *objective evidence* that the verification has covered all programs, data structures, and databases in the SDD and in the code.

6.4.1.4 Hazards Analysis Report

The Hazards Analysis Report (HAR) is the remaining output of the verification team during the coding phase. It is the result of the code hazards analysis activity which has the following objectives:

- to verify that the software required to handle failure modes identified by subsystems hazard analyses does so effectively

- to identify any failure mode that can lead to an unsafe state

- to determine the sequence of inputs that can lead to the software causing an unsafe state.

The inputs to the code hazards analysis are the DID, SRS, SDD and the code. The contents of all of these relevant to verification has been described under previous headings. The output of the code hazards analysis is the HAR which must meet the following requirements:

1. identify the failure modes of RAM variables and ROM constants whose corruption could lead to an unsafe subsystem failure

2. identify failure modes related to instructions in the code which could lead to an unsafe subsystem failure

3. identify input conditions which could lead to the software causing an unsafe state

4. determine that the self-checking software contained in the code can eliminate the identified failure modes or reduce the likelihood of their occurrence

5. recommend revisions in the documents utilized for the analysis where desirable

6. summarize the analysis procedures used, the tools, and the participants.

6.4.1.5 Unit Test Procedure and Report

The verification activities during unit test produce two documents: the UTP and the UTR which are described together under this heading. Unit test is usually conducted on the smallest compilable section of code; in languages such as Ada are separately compilable, it will normally be conducted on the combination (the Program Unit). Unit test is assumed to utilize the target processor but a simulated environment. The key objectives of unit test are to establish that

- the code behaves as specified in the SDD
- the code does not perform unintended functions
- the program interfaces behave as specified in the SDD.

The inputs to unit test are the SDD and the code, both of which have been previously described. The key verification related requirements for the UTP are that the document include:

1. sufficient test cases, based on analysis of the SDD, to execute

   all possible decision outcomes
   all possible conditions for each decision
   values at both sides of the boundary of valid input ranges
   values that may uncover postulate coding errors

2. sufficient test cases, based on analysis of the code, to execute

   every statement
   every condition exit
   every condition for each condition exit
   each loop with maximum, minimum and intermediate values
   cause a read and write to each variable memory location
   cause a read for every constant memory location

3. sufficient test cases to exercise each interface

4. define the correct output for each test case

5. provide a cross-reference between the SDD and the test cases

The UTR shall include:

1. identification of the actual tests performed, referenced to the test procedure

2. listing of actual and expected results, and highlighting of discrepancies

3. summarize positive and negative results

6.4.2 Evaluation of the Ontario Hydro Verification Methodology

The methodology requires free and timely information flow from the developer to the verifier. Where this is possible, it appears to provide an effective means of conducting software verification for safety critical functions. As of the writing of this report (October 1993) the methodology has not been applied in its entirety to a significant product, and hence the reservation "appears to ...". The methodology described here represents excerpts from the reference document and omits substantial portions that the authors of this report considered to be of lower priority than those selected. The selected portions provide adequate coverage of the areas of concern enumerated in Chapter 4 as shown in the following table.

Table 6.4-2 Areas of Concern addressed by the OH Documents

Area of Concern	Addressed in Document*
1. Normal Service	All documents
2. Failure Modes	DID 5, SRS 4,6 SDD 5, DVR 5, CVR 5,
3. Unsafe Actions	SDD 2, HAR 1,2,3
4. Human Interfaces**	DID 3, UTP 3
5. Isolation	DID 1
6. Test	DVR 9, CVR 6, UTP 1,2,3
7. Attributes	DID 2, SRS 5, DVR 1

* This column lists document abbreviations followed by the applicable item under the document

** The documents refer to interfaces in general; an addendum that requires emphasis on human interfaces is required

6.4.3 Life Cycle Activities for the U. S. Environment

The object of the verification is assumed to be a program for a safety function for which a functionally diverse alternate exists, or a simple safety function (single or very few inputs and outputs) without an alternate. As mentioned previously, current verification methodology is not considered adequate for a complex safety function for which there is no creditable back-up. The listing of activities is primarily directed at the case where there are no similar installations in service but the supplier is experienced in nuclear protection systems. The possible deletion or compression of activities where there are prior installations depends on (1) differences in the plant proper and the associated I&C systems, (2) the design basis, (3) availability of other safety systems that may act as back-up for the system to be installed or that depend on back-up from the new system, and (4) differences in the configuration of the system to be installed from that of its predecessors. It is therefore difficult to propose general rules for these circumstances.

6.4.3.1 Requirements Phase

Despite the fact that the design of the system is complete, all activities listed for the OH approach in Section 6.4.1.1 are applicable. The reasons for recommending the full treatment of requirements are:

- it provides a systematic review of the needs and constraints to be met at the target location

- it is essential for the preparation of acceptance test plans and procedures, and for the evaluation of test results

- it permits an evaluation of alternatives for furnishing the required service

6.4.3.2 Design Verification Report

The design phase activities should use the OH documentation requirements as a general guide (Section 6.4.1.2) but place greatest emphasis on

- existence of a design description that is responsive to the concerns described in Chapter 4, particularly Section 4.2.

- traceability of all safety critical functions in the design to the requirements

- identification of design features that do not directly implement requirements

- existence of complete interface specifications between software components, software to database, and software to hardware.

The Design Verification Report shall at least provide evidence that the design documentation meets the four criteria identified above.

6.4.3.3 Code Verification Report

The coding phase activities should use the OH documentation requirements as a general guide (Section 6.4.1.3) but place greatest emphasis on

- traceability of the code to the design
- conformance of the code to applicable language standards
- existence of supplier coding standards and compliance of the code with these
- readability of the code, including adequacy of comments.

The Code Verification Report shall at least provide evidence that the code meets the four criteria identified above.

6.4.3.4 Hazards Analysis Report

The code hazards analysis shall be performed as described for the OH life cycle. The Hazards Analysis Report shall meet all requirements described in Section 6.4.1.4.

6.4.3.5 Unit Test Procedure and Report

The unit test activities described in Section 6.4.1.5 shall be performed as part of the verification for all safety critical code utilized by the system. Unit test shall also establish that non-safety critical code cannot disable or cause interference with safety critical code or the variables utilized by it.

6.4.4 Verification of Isolated Non-Critical Segments

A reduction of verification requirements is possible for non-critical segments, such as diagnostics, self-test, and limited range calibration, where these functions are isolated from critical code as defined in Chapter 4, Section 4.2.5. All isolation provisions must be regarded as part of the critical code and receive full coverage as described above.

Where these conditions are met, the code verification report and the hazards analysis report are not required. The design verification report can be restricted to documenting the calling structure, data usage, and external interfaces of the non-critical modules.

6.4.5 Verification by Reverse Engineering

Where reverse engineering tools are available, significant portions of the verification activities can be omitted or simplified. The following assumes that the ECT and CATS tools described in Section 6.3.3 (or equivalent tools) are available, and that requirements are stated in a format consistent with the reverse engineering output of the source code analyzer.

Requirements verification is conducted in full (Section 6.4.1.1), but design and code reviews can be restricted to verification that an accepted software engineering methodology has been applied consistently, and that the reverse engineering output corresponds to the requirements. The analysis of the reverse engineering output of the object code tool (CATS or equivalent) satisfies the requirements of the Hazards Analysis Report. The unit test report must establish that the required structural coverage has been obtained. All other testing can be deferred to the validation phase.

6.5 SPECIAL VERIFICATION CONCERNS

6.5.1 Commercial and Reused Software

This heading discusses software not specifically developed for the application under consideration but incorporated in it or intended to be incorporated in it (these programs are collectively referred to as *non-developed software*[9]). Commercial software used in support functions (compilers and tools) is covered in later headings. For the purpose of verification it is necessary to distinguish two forms of non-developed software, depending on whether development background and source code are (a) provided or (b) not provided. Minimum information about the development background includes the software specification, design description, and evidence of quality assurance and configuration management during development. It is practically impossible to conduct meaningful verification activities on category (b), and its use in safety grade software must be restricted to portions of the application that are not safety critical and well isolated from safety critical tasks. This conclusion is in agreement with the non-mandatory Appendix D of IEEE Std. 7-4.3.2, Draft 7 (1993), which states that development process steps must be identified by at least the following documents:

- system requirements and acceptance criteria
- software requirements
- software design documentation
- evidence of verification and validation by the developer
- evidence of integrated hardware and software testing
- configuration management procedures and reports.

With regard to category (a), in addition to the documentation identified above, at least the requirements and code hazard analysis activities (see Sections 6.4.2.1 and 6.4.2.4) should be conducted. If the non-developed software has seen extensive use and there is positive evidence of failure-free operation (for an interval commensurate with the requirements of the intended application), other verification activities may not be required. If these premises do not hold, all steps of the verification described in Section 6.4.2 are required. The entire approach for the U. S. environment described in that section is based on software which could not be accessed by verifiers during development, and thus it is essentially suitable for any non-developed software. The reason for skipping some steps where there is evidence that the software has operated satisfactorily in an environment representative of the intended application is to conserve

---

[9] A contraction of "not developed for the specific application under discussion"; in spite of being possibly misleading, this phrase has gained wide acceptance.

resources, and because it is unlikely that design or coding deficiencies uncovered during verification will be corrected unless they have a direct bearing on the safety or performance of the software.

Any failure of non-developed software in another environment will invalidate the verification for the intended application. Therefore there must be assurance from the source of the software that the user in the intended application will be notified of known failures. The notification is required even if the failure in the other environment was not a cause for a change in the code.

6.5.2 Compilers

The benefits of coding the program in a high level language are so great that the inherent disbenefit, the need for a compiler, is sometimes forgotten. The better-known ways in which compilers can introduce faults in the executable code, even though the source code was fault-free, include

- generating an incorrect operation code (or sequence of operation codes) for the operation specified in the source

- assigning in incorrect memory reference for a variable declared in the source

- changing the sequence of operations so that a register (or memory location) is read before the correct value is loaded into it

- assigning an incorrect type designation to a variable, causing it to occupy more (or less) memory than intended.

Most compilers undergo a certification process (essentially verification and validation by an independent agency) that checks for the presence of these known failure mechanisms, but as compilers get more sophisticated they can develop more subtle fault mechanisms that are not readily detected in the certification. It is particularly difficult to safeguard against failures that are dependent on a sequence of source statements, e. g., that occur only when an assignment statement to an array is followed by an assignment of an array index for another array.

Because certification of compilers is a very specialized activity, usually assigned to organizations dedicated to that process, it does not appear appropriate or necessary to provide guidelines for compiler certification as part of this document. A requirement to use only compilers certified by an independent agency (or in very wide use) greatly reduces the possibility of compiler generated faults in the executable code, but it does not completely eliminate it. It is not known how many errors were found in code after it had been certified, but this information deserves to be made available. Verification of the executable code, e. g., by CATS as described in Section 6.3.3, is therefore desirable for safety-critical segments of the program.

6.5.3 Tools

The need for verification of tools depends on the use made of them in software development, test, or verification. Tools that furnish output for review by a professional are less in need of verification than those which furnish output that is incorporated in the operational code or is directly accepted as evidence of satisfactory performance of the operational software. Among tools least likely to require verification are:

- static analyzers, dynamic analyzers and set/use table generators because their outputs are usually input to further analysis by professionals who will detect errors or inconsistencies

- code auditors because incorrectly identified deviations will be detected by further analysis, and the probability that deviant code is not identified is small (and non-conformity with coding standards will not directly lead to an operational failure)

These tools should be subjected to a *partial verification* including at least review of design and user documentation to determine their suitability for the intended application. Tools requiring an intermediate level of verification include:

- requirements analyzers because failure to detect missing requirements is not likely to be compensated for by further manual or automated activities

- test data generators because bias (providing insufficient test cases for a given failure mechanism) is not likely to be detected in test reviews

Verification for these tools should in addition to the above include review of requirements documentation and test reports, with emphasis on the areas of vulnerability that were identified for them. The highest level of verification is required for tools such as

- code generators because their output will become part of the operational program

- automated proof generators (used in formal verification) because their intermediate steps are not ordinarily subject to review by professional analysts.

Where these tools are applied to safety critical software they should be verified to the same degree as operational software, e. g., by the procedures described in Section 6.4.2. The hazards analysis should be conducted with regard to consequences of failure of the tool as well as of the operational software on which the tool is used.

All tools, including those identified as requiring minimum verification, must be under configuration management when used on safety critical software. The configuration management extends to the documentation because tools used with inappropriate (obsolete) manuals will develop failures that can be as severe as those caused by faulty code or design.

6.5.4 Process Audits

The primary emphasis in verification is on attributes of the product. But it has already been mentioned in Section 6.3 that audits may also cover process attributes. Process audits can be conducted on phases of the development or on phases of the verification. Audits of the development are appropriate where a developer does not have a track record in the nuclear field, when it is claimed that a process will inherently assure the suitability of the software product, or if process quality is offered as a substitute for a product requirement, e. g., a highly structured development process as a substitute for software diversity. Audits of the verification process are appropriate where the developer has used an internal verification team during the development and the utility or the licensing agency need an independent assessment of the effectiveness of the internal verification. The scenario for an audit is well described in a recent (undated) NRC Draft document "Operating Reactors - Digital Retrofits - Digital System Review Procedures"

The process audit has two objectives: to determine that the process is carried out completely and correctly, and that it yields the claimed results. The first objective is achieved in auditing the *process implementation* and the second by auditing the *process capability*. Auditing the process implementation usually starts with a review of the organization's Software Process and Procedures manual (in the OH nomenclature used in Section 6.4 it is called the Standards and Procedures Handbook, SPH) or Verification and Validation Manual. Where the manual acknowledges that it is based on a published software or verification methodology the audit team should familiarize itself with the source so that deviations and omissions with respect to the original methodology can be discussed. The primary purpose of the implementation audit is to determine whether the documented procedures are adhered to in letter and spirit. Frequently encountered problem areas in the development process are:

- perfunctory peer reviews, as evidenced by uncommented sign-off on forms, positive acknowledgement of steps that had not been performed, and unrealistic scheduling (too many review steps in one day)

- failure to generate test plans and procedures in connection with requirements and design milestones

- inadequate unit testing -- most development methodologies (as well as IEEE/ANS Std. 7-4.3.2 of 1982) require that at least every decision exit be traversed in unit testing but unless a dynamic analysis tool is used this is difficult to document; yet it is an important requirement that should not be skipped

- poor control of unit development folders (the folders are required by practically all methodologies) -- lack of uniformity among individuals, presence of unsigned forms, missing or outdated documents.

The most frequently encountered problems in V&V audits is lack of documentation of activities performed and inadequate substantiation of assumptions. Both of these are due to the fact that the internal verification team shares the "culture" of the development team and finds it unnecessary to document essential information derived from that culture.

At the conclusion of the implementation audit the audit team shall furnish a report which states either that the implementation met the requirements of the source document (except in stated areas), or that cited deficiencies indicate substantial non-compliance.

The capabilities audit should concentrate on a small number of capabilities important to the intended application that are claimed to be achieved by the process. Examples of such capabilities for software development are: stable requirements, functional partitioning of the design, low software failure rate. Because the claims are frequently stated in the qualitative terms employed here, it requires some familiarity with the state of practice to determine whether a significant technical benefit is being achieved. The developer should be able to supply data to substantiate the claimed benefit, but independent substantiation is preferred. Stable requirements should result in few design changes involving external and high level internal interfaces; where such changes constitute more than 10% of all changes it can be presumed that requirements are not stable. Functional partitioning is intended to minimize the propagation of changes (changes in one module necessitating changes in other modules). Where a single fault or failure report results in changes in several modules it can be presumed that at least in this instant the desired capability was not provided. Representative software failure rates at test and development milestones are found in [MUSA87], and these can be compared with those encountered in the process being audited.

Examples of claimed capabilities for verification are that the process has established that the result of design is fully compliant with the documented requirements and that code is fully compliant with the documented design. These capabilities should be evident in systematically arranged verification reports. Where there is a possibility of interaction between requirements (e. g., that the system be put into state A when x occurs, and into state B when y occurs) the verification report shall either establish that being in state A and state B at the same time is admissible, or else that the design will preclude the acceptance of events x and y at the same time.

At the conclusion of the capabilities audit the audit team shall furnish a report that states either that the claimed capabilities are provided (subject to stated exceptions) or that cited deficiencies prevent acceptance of the claims.

6.6 CONCLUSIONS AND RECOMMENDATIONS

In the introduction to this report we mentioned that providing guidelines for verification and validation is difficult because of the lack of a completion criterion. This applies particularly to the the material discussed in this chapter. While there is no lack of publications on verification

methodologies, there is total absence of conclusive evidence of how effective these methodologies have been in reducing failure rates to the level required for the high integrity systems addressed in this report. In addition, the administration of verification and validation in the U. S. nuclear industry differs sharply from that of the aerospace and defense industries where most verification practices originated. In the latter environments the user or customer contracts separately with an independent organization to verify the software products of the developer, whereas in the U. S. nuclear industry the developer is frequently responsible for the conduct of verification (or of substantial portions of it). This makes some widely practiced and standardized verification procedures inappropriate or of limited value.

In these circumstances the verification practices adopted by Ontario Hydro in connection with the licensing of the Darlington reactor protection system (and further developed for other safety systems) offered the best basis for recommendation for the current U. S. nuclear power environment. There is as yet little experience with the reliability of the software developed and verified under the OH procedures, and there is also a significant administrative concern in that OH had continuous and open access to the software products from the earliest development stages, whereas U. S. utilities as well as the NRC typically have access only after most of the development is complete. In spite of these reservations, the OH procedures offer these benefits:

- they have been reviewed by nuclear and software professionals, and are open for examination by any interested party; no significant objections to the procedures are known

- no negative experiences have been reported in the operation of the Darlington plant

- they are specifically tailored for the nuclear power environment.

The life cycle activities derived from the OH procedures are summarized in Table 6.4-1, and the applicability of the resulting products to the special areas of concern discussed in Chapter 4 of this report is presented in Table 6.4-2. The recommended activities for the U. S. environment are discussed in Section 6.4.3. Together with most verification methodologies, the OH procedures place heavy emphasis on the verification of requirements, and subsequent traceability of requirements to the later development stages. This emphasis is quite consistent with the findings of Chapter 3 of this report that most failures in high integrity systems involve rare conditions. The lack of specific requirements for the handling of rare conditions, particularly of multiple rare conditions, is responsible for many of these difficulties. The condition table methodology used by OH to formalize requirements is highly effective in identifying potential sources of these difficulties.

The labor required for verification can be considerably reduced by the use of tools, and these have additional benefits in enforcing a systematic approach and in reducing the possibility of mistakes. Thus, tool use should be encouraged. However, a number of caveats must be recognized:

- tools are frequently language dependent, and selection of some languages may severely restrict the availability of tools.

- tools may themselves contain faults and must therefore be verified (see Section 6.5.3)

- to further reduce the possibility of faults introduced by tools, the verifiers should use tools that are different from those used in the development.

SECTION 7 - VALIDATION GUIDELINES

7.1 OVERVIEW

This chapter responds to paragraph 4.1.5 of the Statement of Work which states in part:

> Develop guidelines to evaluate the adequacy of the validation program. The guidelines should address the (1) amount of systematic, structural, and statistical testing, (2) acceptance criteria for testing ... and techniques to detect unintended functions, (3) acceptance criteria for the validation of diagnostics and fault tolerance ..., (4) error analyses ... and reliability evaluation for validation, and (5) validation of commercial grade software applications.

Validation is here treated as a computer system level activity intended to determine that the integrated hardware and software complies with the requirements for the computer system (see Section 4.3). The benefits of validation are therefore dependent on the quality of the requirements.

7.1.1 Motivation

If the computer system requirements do not completely or correctly translate those at the plant protection level, even a very conscientious validation effort will fall short of assuring that the system will meet user needs, or will provide the intended protection against plant hazards.

A systematic deficiency of most current specifications is that they do not identify which requirements may have to be met at the same time. Because this will directly affect the testing to be conducted as part of validation the guidelines emphasize this potential problem area and suggest remedial measures. A particular area of concern is the handling of multiple exception or failure conditions, e. g., the program being required to recover from a hardware failure at the same time that a severe thunderstorm causes a high rate of data errors. A statistically based criterion has been developed for requiring such multiple rare conditions to be covered by the validation.

Most of this chapter deals with the validation of custom developed plant protection programs. Special requirements for diagnostic software are discussed in Section 7.2.5 and the validation of commercial software is discussed in Section 7.4.

The established practice, implied in the IEEE/ANS Std. 7-4.3.2 definition, is to conduct validation subsequent to the software/hardware integration step. If significant deficiencies in meeting system requirements are detected at that point, extensive and time-consuming rework will be required. It is therefore recommended that an earlier step of requirements validation be added, to be conducted prior to start of design. The technique of animation of requirements, which has been successfully used in Europe for a number of years, can be employed [HALL91].

The primary validation activities are review of requirements, review of documents generated as part of verification, and the conduct of system level tests. Because it represents the final bulwark against acceptance of a faulty system, the emphasis in this chapter is on the latter activity. The recommended test methodology is a combination of functional testing, structural testing, and statistical (random input) testing. Functional testing is based on the requirements; structural testing is based on the structure of the software; statistical testing subjects the system to inputs selected at random from a data population that is intended to represent a severe operating environment.

In this chapter test is partitioned into a reliability growth phase and a reliability assessment phase. During reliability growth it is expected that failures occur, and the correction of the underlying faults reduces the future failure rate. During reliability assessment failures are infrequent, and there may not be a statistically significant reduction in the failure rate. The methodology proposed here looks at the causes of the remaining software problems that are found during testing for reliability assessment. Earlier SoHaR research, described in Chapter 3, has shown that the predominant cause of failures during this phase is multiple exceptions, i. e., the coincidence of two or more rare input or computer states, each one of which may have been a previous test condition by itself, but the combination of which had not been encountered by the program. When test failures consistently occur under multiple rare conditions, probabilistic reasoning permits an assessment of failure rates that are in the range of acceptable risk, and from this finding a test termination criterion can be developed. The formulation of this criterion is an original contribution of this effort.

7.1.2 Structure of this Chapter

An overview of test methodologies is presented in Section 7.2. The major categories discussed are functional, structural, and statistical testing. Advantages and disadvantages are summarized following the individual discussions. The later parts of Section 7.2 describe validation of requirements as an initial step in the total validation process, and the validation of diagnostics.

Section 7.3 discusses termination criteria for each of the major test categories. It is seen that intrinsic termination criteria for functional and structural test are not very meaningful for the validation of high integrity programs. A new approach for termination criteria for statistical testing is presented that holds promise of relieving one of the major drawbacks of this otherwise desirable technique.

The validation of commercial software is discussed in Section 7.4, and conclusions and recommendations are presented in the final section.

7.2 TEST METHODOLOGY

This section addresses the selection of test methodologies as a part of the validation process. The principal methodologies considered are

- functional testing
- structural testing
- random or statistical testing

The first two of these qualify as systematic test methodologies because it is possible, at least in principle, to define goals for "complete" testing by specified criteria. The term *complete* was placed in quotation marks because it refers to satisfaction of a test attribute and not to exhaustive testing or fault removal. The three techniques were selected from a much larger number of possible test strategies [HOWD78, BISH90] because they are established and have been successfully used in the validation of software for nuclear power applications.

7.2.1 Functional Testing

The aim of functional testing is to determine that all required functions are provided by the software under test. There is an implication that it should also establish the absence of functions that are not required (and particularly absence of undesirable functions) are provided, but, as will be discussed later, functional test is not very effective in this respect.

In the typical planning of functional test the plain text requirements are searched on a text processor for the *shall* string, and sentences containing this string are then assigned successive numbers. At least one test case is generated for each numbered requirement. Where the requirement is conditional at least one test case is generated for each condition outcome. Where the conditions pertain to process variables test cases are generated for small increments above and below the specified limit for each variable as well as for values that are well above and below the limit. Good practice also requires test cases for special values, such as zero or negative values where these are not normally expected.

The first sentence under this heading states that "*all* required functions" are to be tested, and this holds out a promise of "complete testing". If there is a list of all required functions and a test case is successfully executed for each of them, is it warranted to claim that complete functional testing has been conducted? In a purely semantic way the answer may be "yes" but for practical purposes it is "no" as will be seen from the following example. In a payroll program the requirements exist "On the last day of the month close timesheets and compute payroll" and "Every Monday start a new timesheet". These requirements can be completely tested (in the semantic sense) by the following test cases: (1) Tuesday, April 30, and (2) Monday, May 6. But any reasonable, practical approach to testing will also require a test case which is both Monday

and the last day of the month, and by this interpretation the two initial test cases do not satisfy the completeness criterion.

From the above example it is seen that meaningful functional testing requires a specification of the level of *coincidence* (of requirements) that is to be tested for. Perhaps the greatest difficulty encountered in functional testing relates to multiple state transitions under exception conditions and the related problem of the length of operator command sequences, all of which are special instances of the *coincidence* problem. A typical state transition is addressed by a requirement "If sensor data increment exceeds X use alternate sensor data", and this can be tested by the methodology described above. And there may be another requirement "If sensor data is zero use previous data value". Complete requirements should address the following questions:

- Is a test case required for the condition where the primary sensor increment exceeds X and the alternate sensor value is zero?

- Are the individual conditions or the joint failure condition to be combined with another state transition in response to a computer (hardware) failure?

- Is the presence (or absence) of operator commands to be used as a test condition?

Current standards and regulatory documents provide very little guidance regarding multiple exception conditions that need to be considered in the requirements and which subsequently have to be validated. The closest to identifying the need for tolerating multiple malfunctions or exception conditions are statements that may be paraphrased as "The safety system must continue to have the capability of safely shutting down the plant in the presence of any single malfunction *together with any creditable malfunction in other parts of the plant.*" Investigations described in Chapter 3, and also [ECKH91] have shown that multiple exception conditions are indeed a very prominent cause of software failures in systems that have undergone thorough testing under "best practice" methodologies. Functional test should be structured to minimize this failure probability by use of the following guidelines; an additional line of defense against multiple rare events failures is statistical testing.

To develop guidelines postulate that a given rare condition will have a limited active period. The active periods are easiest to define for permanent hardware malfunctions, where they extend from the occurrence of the failure until completion of the repair or replacement. For operator induced events the active period normally terminates when a corrective or reset command is issued. For software failures and transient hardware failures the active period is highly variable but it has a maximum bound in the time required to restart the system. In this connection it is important to note that recent research shows that many high impact software failures are actually the result of failures in the hardware or communication functions [TANG92], and the active period of these is therefore governed by the active period of the underlying failure.

Based on the reasoning stated in the following sentences, it is assumed that (a) all rare conditions are detectable, and (b) the repair actions do not involve inaccessible portions of the plant. The

detection of rare conditions is the responsibility of the diagnostic functions that are a part of every digital plant safety system, and the validation of the diagnostics is an important part of the overall validation process. Repair actions may be required in portions of the plant that are not accessible in the operational state, such as internal reactor sensors, but usually these are provided with extensive redundancy so that the active period terminates with switching to a replacement element rather than with the physical replacement.

The conditions described above permit the establishment of approximate quantitative limits for the level of coincident events for functional test. The coincident event A-B occurs when either element A malfunctions within the active period following the malfunction of element B, or element B malfunctions within the active period following a malfunction of element A. For many malfunction types the failure probabilities and times to repair can be estimated with fair accuracy. Examples are sensor, power supply or relay failures, periods of impaired communication (thunderstorms, external events), and common operator errors. For these failures the probability of a second component failing while the first one is affected by an outage can be computed from[10]

$$p_{AB} = p_A\, p_B\, T_B + p_B\, p_A\, T_A = p_A\, p_B\, (T_A + T_B) \qquad (1)$$

where p_X is the probability of occurrence of X during a given period and T_X is the time to repair of element X, expressed as a fraction of the period for which the probability was stated.

Validation must cover all coincident events (not restricted to combinations of two events) for which the probability computed in accordance with equation (1) exceeds a threshold that depends on the classification and associated reliability requirements of the safety system. For safety systems that allow at most a probability of failure of 10^{-6} per year[11], the threshold for a given joint event will typically be selected between 10^{-7} and 10^{-8} per year because there will be a number of joint events that can all contribute to the system failure probability. A specific example of the evaluation of eq. (1) is discussed in connection with test termination criteria in Section 7.3.3. The procedure for use of functional test methodology for high integrity software must specify the level of coincidence (double, triple, etc) and the combination of states to which it should be applied. A rational approach for such a specification can be derived from the quantitative reliability requirement of the system. The above is applicable to coincidence due to random overlap of independent events. Establishing that coincidence is not caused by common or related causes for multiple failure events is assumed to be a requirement of the hazards analysis.

Other decisions that are necessary for a meaningful functional test are

---

[10] p_{AB} is here the probability of both units being out of service for an arbitrarily small interval. This is not equivalent to the joint event AB which implies that they are out over the same interval.

[11] For the purpose of this analysis failure/demand probabilities must be converted to time based probabilities, e. g., by using demands/year as a conversion factor.

- the handling of compound action requirements, e. g., "the printer shall be turned on and a legend displayed to the operator." Are these sequential or simultaneous actions, and within what granularity of time?

- the handling of compound conditions, e. g., "On Monday and on the first of the month do X". Is the *and* actually a logical or, and if so, is it an exclusive or?

- allocation of requirements to operating modes, e. g., segregating requirements that apply in all modes from those that apply only to operational mode, maintenance mode, calibration, etc. Testing is also required to show that improper mode changes will not occur, such as entering operational mode while calibration data are being processed.

- precise definitions of numerical requirements, e. g., "When the water level reaches 1 m initiate X." Does that mean when there is a momentary surge to 1 m, or when it stays above 1 m for a defined period of time? (Actions on maximum or minimum values should always reference a time interval, and whether continuous or average exceedence during the interval is to be used).

- definitions of the averaging interval, e. g., "When the 5 minute average water level exceeds 1 m initiate X". Does this mean a 5 minute interval with an arbitrary start time, or does the interval start when the measurement first exceeds 1 m?

Many of the above problems are really problems of requirements formulation rather than test, but they are encountered in the process of validation planning or review. They are mentioned here because validation is based on requirements and it cannot be conclusive when requirements are missing or are ambiguous.

One area in which functional testing clearly has an advantage over the other methodologies discussed here is that the test outcomes are either directly specified or are easily derived. This contrasts with the need for a test oracle or back-to-back testing of multiple versions that is typically required for structural or statistical testing.

Most of the functional testing in support of validation will normally be carried out at the system level. However, functional testing can also be conducted at lower levels (applied to portions of the system, or to software by itself) with the requirements allocated to that level. Results of lower level functional testing may be accepted for validation if the affected functions are clearly isolated, such as the display interface.

7.2.2 Structural Testing

Structural testing is guided by the structure of the software itself, and it is therefore sometimes called "clear box" testing. Within the overall category of structural testing the following are the dominant techniques:

- statement testing -- test cases are formulated to execute every statement at least once

- branch testing -- test cases are formulated to execute every branch exit at least once

- condition test -- test cases are formulated to execute every condition at least once (this differs from branch testing only where there are compound conditions, such as "If Monday OR Day 1 ..." which requires testing for both conditions here, but only for one of them in branch testing).

- path testing -- test cases are formulated to execute every feasible path from start to exit of a defined program segment at least once

- data flow testing -- test cases are formulated to execute every feasible data flow at least once.

The goals of the structural testing can be modified to require at least 90 or 95 percent of the criteria (statement, branch , etc.) to be completed instead of every one. This relaxation is sometimes justified by the high cost associated with accessing the last 5 or 10 percent of the structural elements. In path testing only a fraction of the structurally identifiable paths are feasible when the semantics of a program are considered, and therefore the above description refers to every *feasible* path. The distinction between a structurally identifiable and a feasible path is shown in the following program segment:

```
If day_of_week = Friday
        Then sum hours for week
        Else continue
.
.
If day_of_week = Saturday
        Then increase rate by 50%
        Else continue
```

The path using both *then* exits is structurally identifiable but it is not a feasible path because the day of the week cannot be Friday and Saturday at the same time. In this example it was easy to demonstrate that the path was not feasible, but in dealing with physical variables much more subtle dependencies are frequently encountered that make it more difficult to enumerate all feasible paths. Where path coverage is a requirement, and where paths remain untested that are not obviously infeasible a listing of such paths must be furnished. This listing should be reviewed by the sponsor of the development and, if the paths are non-critical and small in number, may be accepted in satisfaction of the requirement.

IEEE/ANS 7.4.3.2 - 1982 includes a requirement for testing of all logic branches as part of verification (par. 7.3.3). Standards that are more demanding in structural test include the OH

standard discussed in the preceding Chapter and the U. K. MOD 00-55 (par. 33.2) which requires that test access all of the following:

(a) all statements
(b) all branches for true and false conditions and case statements for each possibility, including "otherwise"
(c) all loops for zero, one and many iterations, covering initialization, typical running, and termination conditions.

The non-mandatory Appendix E of IEC Publication 880 contains language essentially identical to (a) - (c) above and adds path and data flow requirements for module level tests.

In a practical sense software tools (test analyzers, dynamic analysis tools, automated verification systems, collectively referred to as test harnesses) are essential for all structural testing. The availability of tools is dependent on the programming language selected. In prior work the authors of this report have identified languages for which adequate tool support exists [HECH93A], and the developer should be responsible for furnishing equivalent tools if another language is utilized. The test tools generally perform the following functions:

1. Instrumentation -- they insert counters at points in the program for which access is to be determined (after every statement for statement coverage, every branch exit for branch coverage, etc.) and instructions to increment these counters when they are encountered in execution.

2. Run-time analysis -- at the conclusion of every run they furnish reports of the points that have been accessed and those that have not been accessed during this run.

3. Cumulative run analysis -- reporting on the number of times each point (counter) has been accessed during all executions to date (from an arbitrary starting point). Together with this there is usually a list of points that have not been accessed.

Some of the tools also furnish aids for test case generation that will access points not reached in prior runs. Most structural testing is carried out at the module or subprogram level, partly due to tool limitations and partly due to the difficulty of accessing some program functions in a system environment. Once the test tool is installed, any test data set, including those generated for functional testing, can be used as part of the structural test. Indeed, the initial data sets used for structural testing are usually those generated from an analysis of the requirements. Because the activity takes place prior to the specific validation test, structural testing is sometimes considered a component of verification rather than validation. Even where it is not formally a part of the validation process, it is essential that the records of the structural test program be made available to the validation team.

Structural testing can be, and usually is, terminated when the required coverage is reached, such as all branches, or 95% of all branches. Complete testing to a structural test criterion is by no means equivalent to complete program testing. In [HOWD78] complete branch testing found only 6 out of 28 faults, and complete path testing only 18. All structural techniques combined with anomaly analysis and interface analysis found only 25 out of the 28 faults [HOWD78].

The correct results for a given execution in structural testing are usually less obvious than in functional test. One reason is that the conditions that determine the internal path do not always map directly to the external conditions (which determine the expected result). As an example, an external requirement to take action when a threshold is exceeded for 3 continuous seconds might be represented internally by a loop exit condition that does not obviously translate into 3 seconds. Another reason is that most of the structural testing is conducted at the module level where even the external input and output variables are transformations of the overall program variables. For these reasons expected test results must be computed analytically or obtained by an independent system simulation. Another method is to code two versions of the program independently and then compare results under identical test inputs.

Structural testing, particularly path testing, is probably the best methodology for detecting unintended effects because these are usually associated with a particular condition exit or sequence of condition exits (path testing covers sequences of exits). For this reason structural test is considered essential for the validation of programs for high integrity applications. The generation of test data can be based on functional or statistical methodologies. As long as a suitable test tool is in place, the test results will be included in the coverage computed for the cumulative analysis.

7.2.3 Statistical Testing

In statistical testing test data are generated randomly from defined distributions. The execution can be monitored by a test tool and thus contribute to the cumulative coverage against the selected structural criteria. Selection of a statistical test methodology involves many more decisions than the selection of a functional or structural methodology. Typical decisions, some of which are described in IEC Publication 880, include:

1. Scope of test input space: expected operational profile, expected operational profile with k-fold amplification of safety system actuation demands, range of profiles determined by given distance from safety system actuation point (above and below), expected operational profile with m-fold amplification of operator initiated actions (mode changes, power settings, etc)

2. Type of statistical distribution: uniform, Gaussian, bi-modal (peaked at extremes), exponential (decreasing likelihood of data with distance from actuation point). Different distributions may be required for each type of input variable.

3. Sequence of input conditions: are new input conditions selected completely at random, or should the new conditions be reachable in the operational environment from the previous condition? In the latter case the distribution applies not to the variable as a whole but to the increment over the preceding value.

4. Data attribute to which the distribution is to apply: each individual input variable, the vector of all input variables, a required program function, the vector of all program functions, probability of detecting expected program faults. Where vectors are involved an additional decision is required whether the components should be selected truly randomly or with a predetermined correlation (e. g., corresponding to operational usage). Truly random selection can result in data representing operationally infeasible or extremely unlikely conditions. The data attribute may also include internal states of the software, e. g., the average of an external variable over a given number of cycles.

5. Test termination criteria: this may be total number of runs, number of consecutive runs without failure, to reach a given execution time, or to complete a certain number of runs for given conditions (e. g., that result in activation of the safety function).

More sophisticated criteria for test termination are discussed in Section 7.3 and recommendations with respect to the other criteria are also developed there.

The evaluation of test results is more difficult in the case of statistical testing than in structural testing, and much more difficult than in functional testing. It has been recommended that this technique be used only where a test oracle (a simulation or an alternate version) is available [BISH90]. For safety systems many of the test cases may be selected from separate distributions of the activate and non-activate conditions (where outcomes are therefore known), leaving only a fraction of the runs to be selected from initially uncertain input conditions. Also, it may be possible to generate a simulation at reasonable cost, or one may be available in the form of an analog system serving the same function. Where multiple versions are run back-to-back it is of course extremely important that they be truly independent creations to minimize the possibility of both giving the same incorrect result.

The greatest benefits of statistical testing are:

- ease of generating large volumes of test data

- ability to tailor the test to selected operational profiles or anticipated sources of error, and particularly to multiple rare conditions

- reduced possibility of missing rare operating conditions that were not anticipated by either the program designer or the test designer

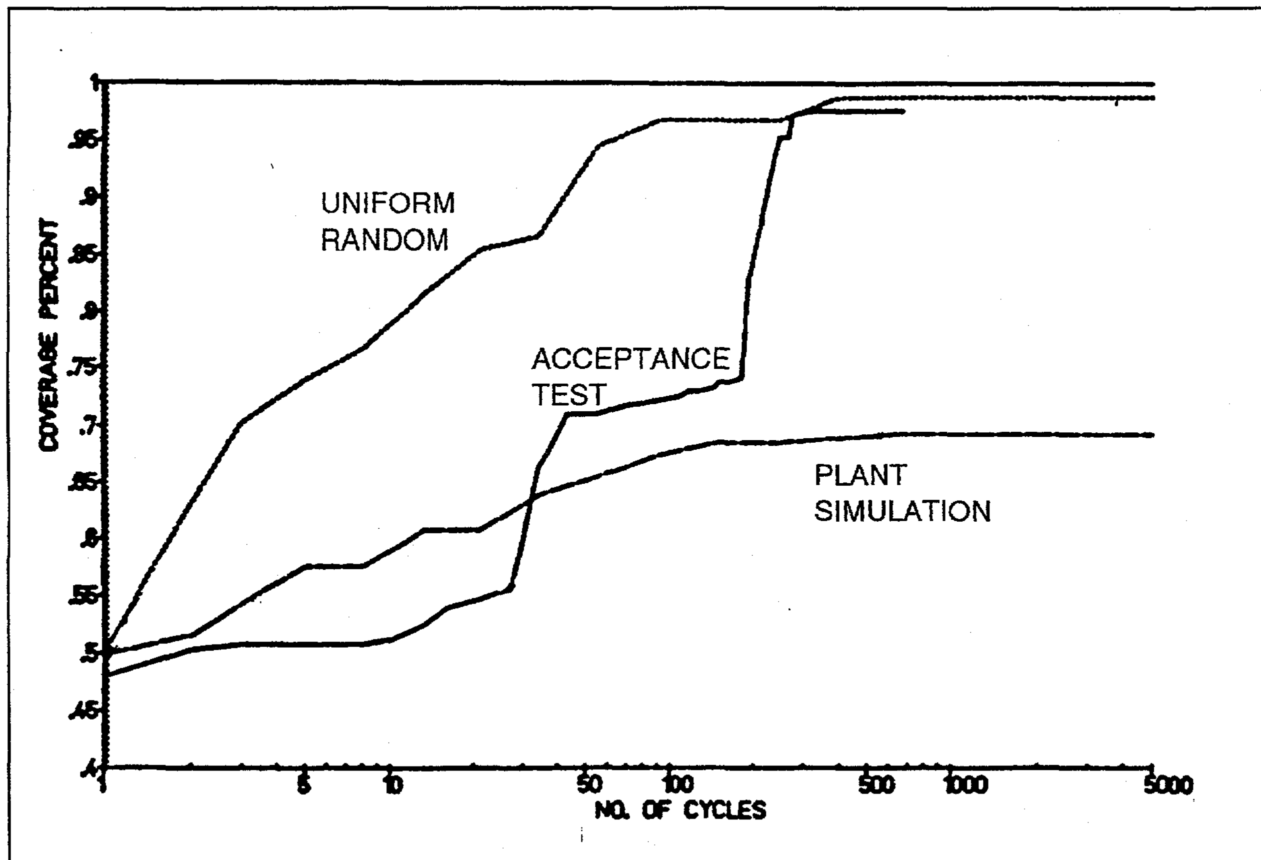


Figure 7.2-1 Branch coverage as a function of test cycles

In addition, statistical testing is a very efficient way of achieving high structural coverage as documented in the Halden experiment [DAHL90]. Because it achieves high structural coverage it is also well suited for finding unintended functions. Figure 7.2-1 shows branch coverage as a function of the number of test cycles for the acceptance test (using functional test methodology), a uniform random data selection over the entire input space, and data generated by a plant simulation. The differences seen in the approach to full coverage are probably due to the following circumstances:

1. The acceptance test deliberately first covered normal plant operating conditions (up to about cycle 30), then transitioned to mildly disturbed conditions (to about cycle 200), and finally to unusual plant conditions. The first few cycles of each transition caused previously inactive branch exits to be executed and thus resulted in the steep increase in coverage.

2. The uniform random data selection caused normal, disturbed, and unusual plant conditions to be accessed in a random manner and thus resulted in the nearly constant slope over the first 100 cycles after which it became much more difficult to find previously inactive branch exits.

3. The plant simulation generated data from normal and mildly disturbed conditions in a random manner which accounts for the nearly uniform slope to cycle 150, and after that essentially no new branch exits were taken because it was not programmed to simulate the unusual conditions.

The distinct advantage of the uniform random data selection over the others is in a large part due to the fact that the latter were not intended for rapid attainment of high structural coverage. The order of conditions in the acceptance test could have been modified so that mildly disturbed conditions were generated after only 5 cycles with normal conditions, and that unusual conditions were generated after only an additional 10 cycles. Similarly, the plant simulator[12] could have been programmed to amplify the proportion of disturbed and unusual conditions. These procedures would have made the alternative test methodologies equivalent or possibly superior to the uniform random test data.

Although the drastic advantage of statistical testing shown in the figure may not prevail under all conditions, the benefits of statistical testing are such as to warrant its inclusion in any validation suite for high integrity software. More about the specific uses of statistical testing is presented in Section 7.3.

7.2.4 Relative Evaluation of the Test Methodologies

The three test methodologies described above are much more complementary than competitive. Attention is called again to the use of functional and statistical methodologies for generating test cases the execution of which can be evaluated for structural coverage. Also, the preparatory activities required for functional and statistical testing are largely the same: definition of admissible states and data ranges for input variables, as well as means for monitoring the resulting output. For functional testing specific values are then selected, whereas for statistical testing the distribution of candidate values is defined.

---

[12] Plant simulators are frequently not suitable for simulating unusual computer conditions.

TABLE 7.2-1 EVALUATION OF TEST METHODOLOGIES

Characteristic	Functional	Struct.	Statistical
Basis for input selection	Req'mts	Code	Req'mts
Test condition selection	Easy	Difficult(1)	Easy
Test data generation	Moderate	Difficult(1)	Easy
Outcome analysis	Easy	Moderate	Difficult
Completeness criterion	Inherent(2)	Inherent (2)	See Sect. 3.3
Finding unintended functions	Difficult	Moderate	Moderate
Typical scope of test	System	Segment	System

Notes: 1. For high test coverage
2. Does not imply that testing as a whole is complete

The relative ranking of the three methodologies for a number of important characteristics is shown in Table 7.2-1. Test condition selection refers to the general conditions for a test case or series of cases, such as: temperature above x, only one pump available. Test data generation refers to specific values that implement these conditions.

One of the uses of this table is to investigate the characteristics for which a *difficult* rating is found. As indicated by Note 1, test condition selection and test data generation become difficult only as high coverage is approached. At that point several alternatives are available:

(i) determine whether the structural outcomes for which test cases could not be generated are indeed feasible. If not, remove these from the base for coverage calculation.

(ii) attempt to reach the required outcomes by statistical testing. The range of test data can be restricted to values that are likely to produce the desired outcomes

(iii) prevent access to untestable sections of the code by means of assertions that produce a safe outcome on failure (i. e., if access is attempted)

The difficulty in validating the absence of undesired outcomes by functional testing can be overcome by using one of the other methodologies. The probability of undesired outcomes is minimized when the program has a low structural complexity and does not use shared memory.

Means of simplifying outcome analysis for statistical testing have already been discussed in the preceding heading: use of simulation or of alternate software versions for the same function.

7.2.5 Validation of Requirements

The *verification* of requirements is concerned with the appropriate and correct allocation of higher level (primarily safety function) requirements to the computer system, and from the computer system to the software. Compliance with applicable government, voluntary and organizational standards will also be evaluated as part of verification. The *validation* is concerned with the evaluation of the requirements from the operational point of view. Examples of specific questions that should be addressed by requirements validation include:

- do the functional requirements cover the entire operating range?
- are interactions with the operator clearly identified, and is there protection against incorrect, missing, or delayed response?
- do the requirements restrict the acceptance of specified operator inputs (or sequences of inputs) under some plant conditions?
- do the requirements provide defense in depth against unexpected plant states, operator actions, and combinations thereof?
- are all pertinent timing and sequencing requirements identified, and are they (a) adequate under limiting adverse conditions, and (b) feasible for the proposed implementation?
- are reliability, maintainability, and fault tolerance requirements adequate, feasible and verifiable?

One means for validation of requirements is animation. The requirements are formulated in a machine readable form, and the response of the requirements to input scenarios is evaluated either as natural or emulated computer outputs, or in terms of response from a plant simulator. Making the requirements machine readable can be achieved by formulating them either in a specification language (which can be transformed into an executable form) or as rules of an expert system. The specification language has the advantage that this is also a very suitable format for verification and possibly for automated proof or correctness. Disadvantages are the need for translation from the plain language text (effort and possible mistakes), and the lack of formalism for some requirements (timing, reliability). The primary advantage of the expert system is that it can accept plain text with minimal modification, and the disadvantage is that the manipulation of the text by the expert system shell may introduce unintended effects (the expert system itself is difficult to validate).

Validation at the requirements level is not expected to be conclusive. It is a necessary but not sufficient step in the overall verification and validation process. Therefore the limitations that

were mentioned in the preceding paragraph are not intended to detract from the benefits that the mentioned approaches (and possibly others) can offer: early recognition of problems in the formulation of requirements or in their implementation, and the ability to observe the effect of modifying, removing, or adding requirements on the ability of the plant protection system to respond to challenges.

7.2.6 Validation of Diagnostics

One of the primary advantages of a digital system over an analog one is the ease with which concurrent diagnostics and calibration can be automated, thus reducing maintenance requirements and making the safety of the plant less dependent on personnel actions and skills. Requirements for diagnostics are included in IEC Publication 880 par. 4.8 under the heading of self-supervision (in other documents the term self-monitoring is used). Excerpts from these requirements are presented below:

> The computer software shall continuously supervise both itself and the hardware. This is considered a primary factor in the achievement of the overall system reliability requirements.
>
> Self-supervision shall meet the following requirements, unless it is proved that other means furnish the same degree of safety:
>
> a) no single failure shall be able to block directly or indirectly any function dedicated to the prevention of the release of radioactivity;
>
> b) those parts of the memory that contain code or invariable data shall be monitored to prevent any changes.

Subsidiary clauses deal with execution of the diagnostics during normal plant operations and require that they not interfere with intended system functions. There are also requirements for periodic off-line testing. Notes in Appendix A of IEC 880, par. 2.8 include guidance in the following areas:

- failures shall be identified to a narrow environment
- fail-safe output shall be guaranteed as far as possible
- if such guarantee is impossible, only less essential safety requirements shall be violated
- remedial procedures such as fall-back, retry, and system restart should be considered
- failures should be signaled to the operators

- intermediate reasonableness tests shall be provided

- software and functional diversity may be integrated into the diagnostics

This guidance is particularly vague in the important area of reasonableness tests and should be supplemented by the following:

(1) reasonableness tests shall be performed on all sensor signals for compliance with (a) expected increments from last reading, (b) consistency with correlated sensor readings, and (c) physical laws or constraints, such as conservation of mass, energy or momentum.

(2) reasonableness tests on operator commands shall assure at least that the command is (a) not spurious (e. g., a sequence of keystrokes that may result from unintended operation of the input device), (b) proper for the current state of the plant and program, and (c) authorized for the position from which it originated.

(3) reasonableness on program operation shall assure that (a) all prior or pending high priority diagnostic instructions have been executed, (b) prerequisites for the current program step have been met (e. g., fresh inputs are available), and (c) the entry into the current module is from a program step for which such a transition is authorized.

(4) reasonableness test shall preclude processing of conditions which are outside the scope of the diagnostics (this protects against possible amplification of the effects of multiple rare condition failures by inadequate diagnostics).

Where specific requirements for diagnostics are not provided, those contained in IEC Publication 880, modified by (1) - (3) above, are a suitable basis for validation of the diagnostics.

7.3 TEST TERMINATION CRITERIA

Functional and structural methodologies have implicit test termination criteria but these do not translate into complete satisfaction of validation requirements. Means of dealing with these problems are discussed in the first two headings of this section. Termination criteria must be externally supplied for statistical testing, and a number of approaches for generating suitable values are presented in the last heading.

7.3.1 Test Termination for Functional Test

The minimum test termination criteria for functional testing are to establish that each requirement is

(1) implemented when required conditions exist

(2) is not implemented when required conditions do not exist.

These minimum requirements can be satisfied with two test cases but that does not provide assurance that the requirement is implemented for all required conditions, or that it is not implemented for all conditions where it is not intended. Test techniques that explicitly show conformance to all (positive and negative) conditions for a requirement are not practicable (or, for continuous variables, not feasible). This dilemma has given rise to the search for means of partitioning the input domain such that within each partition one test case will demonstrate correct operation for any input within that domain. This technique can be made to work with small modules that have a limited number of inputs. For realistic program components, leave alone complete programs, it breaks down because the number of domains becomes too large and sometimes unbounded. From a practical point of view an equivalence domain (within which all inputs are assumed to be processed in the same manner) is best defined by a path in the program structure. The problem of partitioning of the input domain is therefore resolved when functional test is augmented by structural test.

A related problem that arises in defining termination criteria for functional test is that of coincidence which has already been mentioned in Section 7.2.1. It is concerned with the number of conditions that have to be combined in an individual test case. Assume that the requirements document contains the following statements:

1. when clock A indicates 59.99 minutes all exceptions shall be logged to disk

2. when a high sensor reading is encountered in sensor S1, the immediately preceding reading shall be used for one cycle and thereafter the alternate sensor shall be used.

3 - 5. equivalent statements for sensors S2, S3, and S4.

6. when two or more alternate sensors are in use send an alarm to the operator

How many test cases are required to test the compliance of the program with these conditions? With regard to statement 1 the system can be in two states (logging or non-logging); for each of the statements 2 - 5 it can be in three states (normal reading, high for first cycle, and high for subsequent cycles); and for statement 6 it can be in 16 states. Thus complete coincidence testing for this simple set of requirements calls for $2 \times 3^4 \times 16 = 2592$ test cases. Rationales can be developed for partitioning the scenarios so that fewer tests will suffice, but the most practical resolution is again to rely on structural testing with the reasoning that all combinations that involve different processing will be accessed in path testing.

In practice, functional test is therefore frequently restricted to testing for individual requirements, and structural and statistical testing are depended on to access multiple requirements conditions.

7.3.2 Test Termination for Structural Test

The principal software attributes tested by structural test are statement, branch, condition, path, and data flow. After reasonable conventions for handling of loop tests are introduced, a finite list of test conditions can be developed for each attribute, and when all tests for that list have been successfully passed the testing for that attribute can be terminated. Although these lists are finite, they can be uncomfortably large, particularly for path testing. This problem can be overcome by partitioning as shown in Figure 7.3-1. In a simplified but otherwise representative model of a safety actuation system the software is divided into two sections: Sensor processing and Actuation processing. In the sensor processing part suspect readings are identified and calibration and smoothing is applied to good data. In the Actuation part the calibrated and smoothed data from the sensor part are used to determine the state of the plant and to position the actuators to furnish an appropriate response. Each half of the program has four paths, and if complete path coverage is required, sixteen paths will have to be tested. By partitioning the program at the interface between the sensor processing and actuation processing sections only four paths in each, or a total of eight, will have to be tested. A technical requirement to permit this type of partitioning is that the instructions coming down the single path between the two programs are of the same type, regardless of which path in the sensor processing they originated.

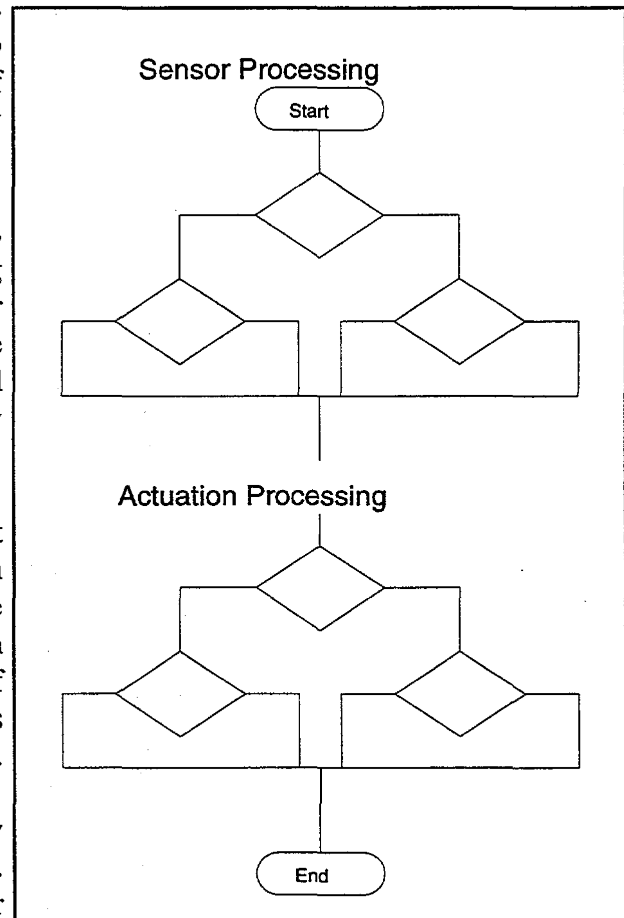


Figure 7.3-1 Partitioning a program

Statement, branch, and condition testing is not as greatly simplified by partitioning as path testing is. However, the ability to access all branches or conditions can be significantly improved in the partitioned environment. The test cases required for data flow testing can also be significantly reduced by appropriate partitioning.

Partitioning brings with it an obligation to test and review the interfaces very thoroughly. In particular, it must be determined that the instructions being accepted at the interface can be handled in exactly the same manner, or, if different processing is required, that this information is supplied by associated parameters.

Partitioning is best performed where only a few different data types flow across the interface. Practical partitions tend to be considerably larger than those shown in Figure 7.3-1, and therefore routine path testing of all partitions of a program

may be impractical. The use of random testing to identify sensitive partitions, and other desirable interactions between structural and statistical testing are described in the next heading.

In general, statement testing provides very low assurance of correct operation of a program. Branch testing signifies that at least all single conditions provided in the design can be processed under favorable circumstances. Path testing signifies that under favorable circumstances all reasonable combinations can be processed. None of the structural tests are effective in finding data or data flow dependent errors. Additional testing under data flow based scenarios will therefore be helpful, but similar objectives can also be achieved by integrating structural and statistical tests.

7.3.3 Test Termination for Statistical Testing

In the previous headings it has been shown that the apparent implicit termination criteria for functional and structural testing actually did not provide any assurance that testing was sufficiently complete to permit a quantitative assessment of the reliability. For statistical testing there is no inherent termination criterion, and yet it will be shown that criteria can be established that permit a quantitative estimate of the probability of failure to be formulated.

It is assumed that statistical testing is carried out as the final step of an acceptance test program and has been preceded by functional testing at least to the extent of determining positive and negative compliance with each requirement, and by structural testing at least to the extent of complete branch coverage for all code associated with the operation of safety functions and by path testing for critical modules. These conditions are important to minimize modification of code for fault removal during statistical testing so that the testing is conducted on a stable software product.

It is desirable to retain the test harness (see Section 7.2.2) employed for structural testing during statistical testing because this facilitates (a) evaluation of the correct result for each run, and (b) the identification of the path in which a failure was encountered. Alternatives for the evaluation of the correct result are: other versions of the same program (these can be restricted to the safety critical functions), reverse mapping (determining the input space for which a given output condition should prevail), and plant simulators which include equivalent safety algorithms. Some failures manifest themselves by very obvious deviations from the desired output, such as computer crash, overrunning of a time limit, or illegal computer operations, and for these none of the alternatives discussed here are required. The inclusion of strong reasonableness tests (see Section 7.2.6) greatly increases the probability that a deviation from normal program operation will become obvious. The identification of the path in which a failure was encountered is important in order to recognize the mechanism by which it occurred. Possibly the same mechanism may cause failures in other parts of the program that should be immediately investigated when the first failure is encountered.

For the purpose of discussing test termination criteria, the test interval may be divided into the reliability growth phase and the reliability assessment phase. In the former established reliability growth models can be used to monitor progress in fault removal. During the assessment phase an alternative approach which is still in the experimental stage may be more appropriate and details on this are presented shortly. The division between reliability growth and assessment cuts across project management boundaries. In some projects reliability growth may be complete well before entering acceptance test whereas in others the termination may occur during the acceptance test. A practical but admittedly arbitrary criterion for terminating the reliability growth phase is when no failures are experienced during two successive data collection intervals or when the average number of failures for five successive intervals is one or less (whichever occurs first). The rationale for this recommendation will become clear from the following discussion of the basics of reliability growth modeling.

Software reliability growth models assume that removal of faults should result in a reduction of the failure rate [GOEL79, MUSA87, SCHN75]. During the test phase it is assumed that faults are identified as a result of failures (this is not universally true but is accepted as a good approximation of actual events). The differences between individual models arise from the specific relation between failure rate and fault removal (proportionality or a more complex function), and from assumptions about the effectiveness and lag of the fault removal (not all faults are completely corrected, and the corrections may not show up until a much later period). An excellent survey of software reliability models is available [FARR83], and the author of that report is continuing to provide computer based aids to the utilization of most of the popular models under the acronym SMERFS (Statistical Modeling and Estimation Functions for Software) [FARR85]. Results generated by SMERFS using the Schneidewind model on a set of actual data are shown in Figure 7.3-2. The solid curve in the left part of the figure shows the model estimate of the number of failures in an interval while the black squares represent actual failures. It is seen that the model becomes less relevant as the average number of failures during an interval decreases. While a difference of two failures between the estimate and actuals may still

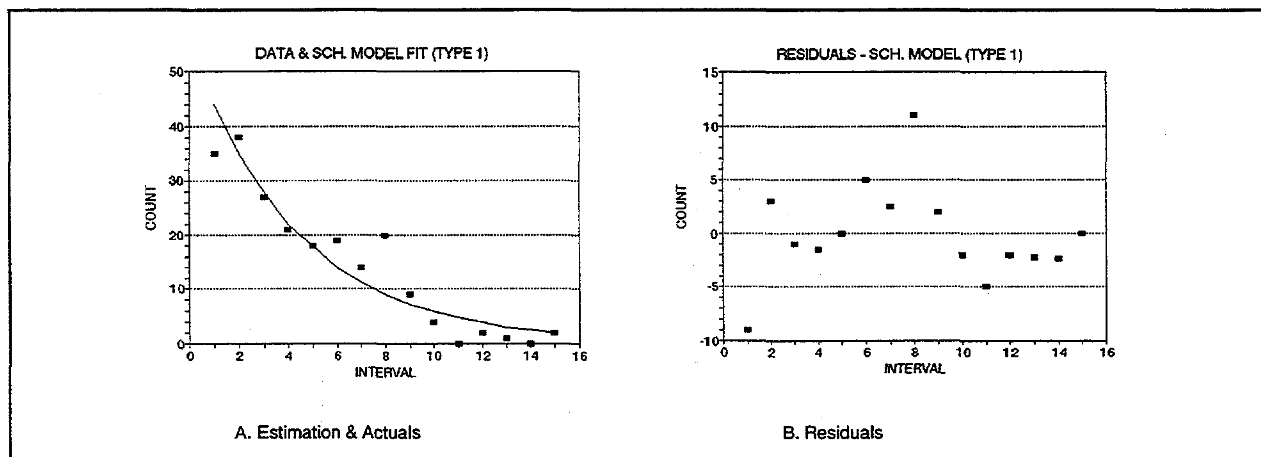


Figure 7.3-2 Reliablility Model

be considered a good fit when ten or more failures are encountered during an interval, the same difference detracts significantly from the value of the model when the actual number of failures is one or less. Therefore further use of this model has been discontinued after interval 15.

The right side of the figure shows the difference between the actuals and the model estimate. This type of representation is valuable for validation in identifying unusual events, such as the large positive deviation at interval 8. The investigation of such events may shed light on underlying causes of failures (e. g., integration of new modules or changes in requirements) or it may indicate that they are irrelevant (e. g., several reports about a single fault).

During the reliability growth phase all faults in frequently executed portions of the program and most single faults in rarely executed faults of the code should have been identified and corrected. The reliability assessment phase is therefore primarily concerned with failures which occur when multiple rare conditions are encountered during the execution of a program. The transition between failure modes that occur when software is subjected to an extensive test program is shown in Figure 7.3-3. At the beginning of test practically all failures occur under routine conditions (in frequently executed portions of the program). As testing progresses an increasing fraction will be found due to single rare conditions and in the final stages practically all will be due to multiple rare conditions. In the middle of the figure the scale for the vertical axis is changed to show better what happens in the right tail of the curve. Obviously, reliability growth continues there, but because the failure rate is already very low it is difficult to evaluate this growth by the conventional means. Instead, the suggested approach depends on the qualitative change in the failure modes, and particularly on entering a region in which the predominant failure type is due to at least two rare conditions.

Rare conditions for the purpose of this discussion are hardware or software exceptions that cause the program to enter code that had not previously been executed, and where there is therefore a much higher probability of uncovering a fault than in previously executed segments. Rare conditions can be caused by:

- hardware failures: computer, voter, bus, mass memory, sensors, I/O processing
- environmental effects: high data error rates (e. g., due to lightning), excessive workloads, failures in the controlled plant
- operator actions: sequences of illegal commands that saturate the error handling, non-sensical command sequences (not necessarily illegal), logical or physical removal of a required component.

The test data for statistical testing should provide a population that is rich in individual rare conditions and the random process should be organized to make it likely that multiple rare conditions will be encountered. This may be difficult but is considered essential for the highest integrety levels by the authors of this report. An example is that each test case is comprised of four events, representing, respectively, success (routine operation) or failure (a rare event) for temperature sensors, radiation sensors, computer channels, and output devices, respectively. Assume that four random numbers are generated to represent the individual events and that the boundaries for routine and failure outcomes are selected so that for each individual events there

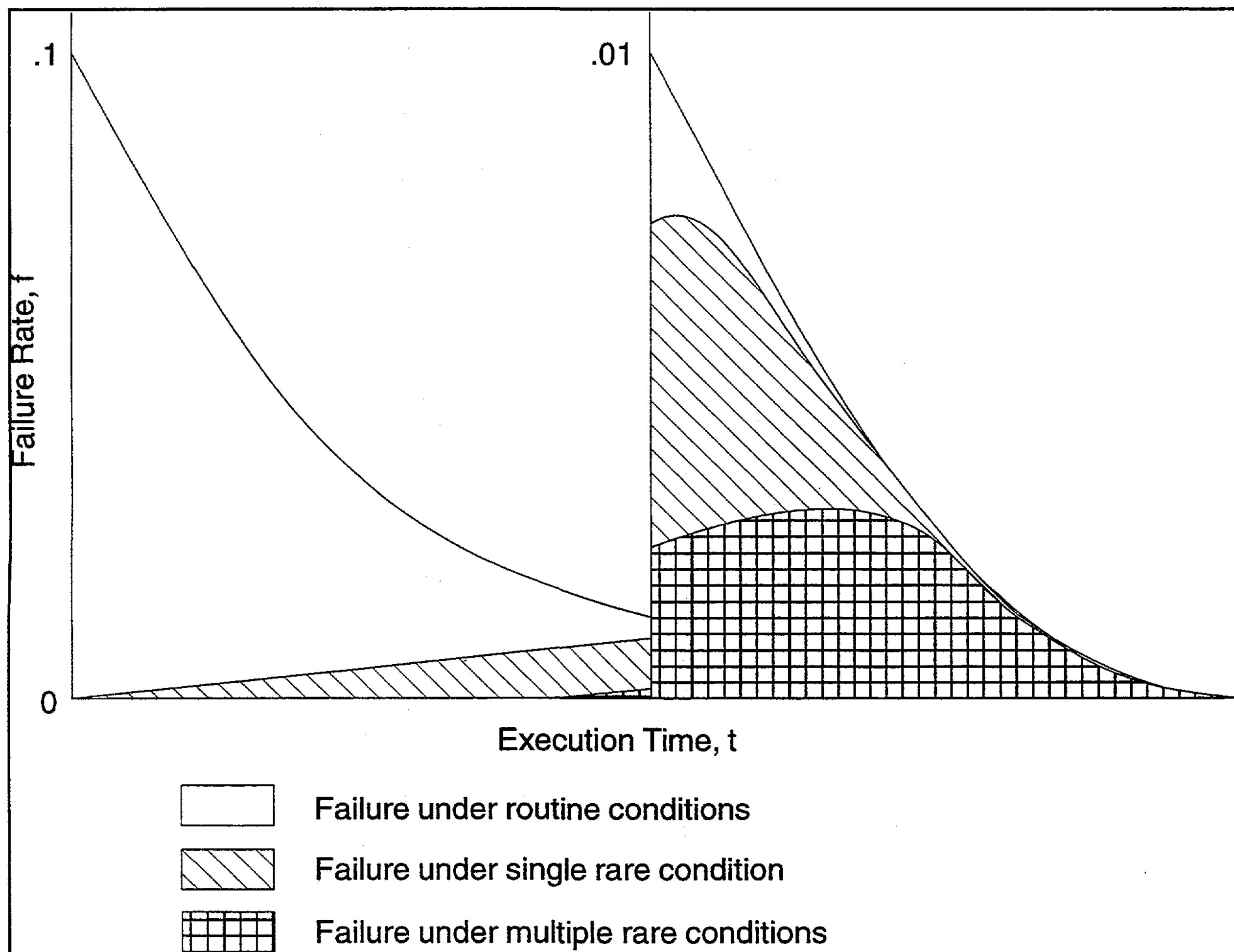


Figure 7.3-3 Progression of failure types during test

is 0.8 probability of success (routine operation). The probability of encountering rare events in a test case under these conditions is shown in Table 7.3-1.

Table 7.3-1 Probability of Rare Events
Four simultaneous events, each 0.8 probability of success

No. of Rare Events	Probability
0	0.4096
1	0.4096
2	0.1536
3	0.0257
4	0.0016

The basis for declaring the reliability assessment phase successful is that the joint probability of encountering the multiple rare conditions that cause failure is less than the allowable failure probability of the software. The following example will show how this can be demonstrated for a given test case:

The simulated conditions that caused failure are: (a) failure of a temperature sensor and (b) failure of a computer I/O channel. In operation the temperature sensor failure is estimated to be encountered no more often than once in 2 years and computer channel failure has occurred at a rate of one per year. Since these conditions occurred on different components it is accepted that they are independent. Replacement of the temperature sensor takes 12 hours (0.0014 years), while replacement of the computer channel can be effected in 4 hours (0.0005 years). The failure rates and repair times have been assumed at higher values than are expected in order to avoid example data with too many zeros. Actual joint event probabilities are expected to be several orders of magnitude smaller than those discussed in the following.

The joint event is represented by (a) the sensor failing while the I/O channel is being replaced, or (b) the I/O channel failing while the sensor is disabled. By applying eq. (1) from Section 7.2 the probability of the joint event during a given year is computed as $0.0005/2 + 0.0014 \approx 0.0017$. This figure represents the probability of a particular failure the cause for which is actually being corrected once it has been found. However, if the latest failures that are being experienced during a period of statistical testing are all due to multiple rare events with a joint probability of at most p per year, then it can be argued that the total failure probability of the software is of the order of p, as is explained below.

Assume that the random test case generation produces five test cases with routine or single rare conditions for each test case with multiple rare conditions (approximately the distribution shown in Table 7.3-1). If the probability of failure under multiple rare conditions is assumed to be equal to that of failure under routine or single rare conditions, this situation can be represented by drawing black (single) or white (multiple) balls from an urn that contains five black balls and one white ball. The probability of drawing a white ball at the first drawing is 1/6 or 0.17, of

successive white balls in two drawings (with replacement[13]) is 0.0277, and for three successive white balls it is less than 0.005. If three successive failures due to multiple rare events have been observed, it can then be concluded that the probability of failure under single and multiple rare events is not equal, and there is a basis for assigning a chance of less than 1 in 200 that the next failure will be due to a routine or single rare event[14]. To be statistically valid this experiment must be started at a random time (if it is started on seeing a failure due to multiple rare events, then that failure cannot be counted as being the first one in the sequence).

This reasoning is not claimed to be a rigorous test termination criterion, but it can be used as a practical guide and in that sense represents a significant advance in establishing a software test methodology for high integrity systems. Further research and experimentation in this area can provide substantial benefits for arriving at an objective validation technique. The criterion is self-adjusting to the allowable failure rate. An extremely low allowable failure rate will require more testing because it will require that the multiple rare events encountered in test failures have a low joint failure probability.

7.4 VALIDATION OF COMMERCIAL SOFTWARE

The correctness of commercial software with respect to system requirements and design documents is established as part of verification. The primary questions to be answered by the validation of commercial software are:

- what evidence is provided that the inherent failure probability of the commercial product will constitute only a small increment of the maximum estimated failure probability of the overall plant protection system?

- is the configuration control (including version control) of the commercial software adequate for precluding the existence of inconsistent and untested combinations of commercial and developed software?

- are all modifications to the standard commercial software (including values for user selectable parameters) that are required for its use in this application documented, and have tests been conducted to establish their safety?

- are there safeguards against (a) errors in the developed software propagating to cause failures in the commercial software, and (b) errors or unexpected responses from the commercial software propagating to cause failures in the developed software?

---

[13] Drawing with replacement is used because the probability of further failures is unknown. Under these conditions, the correction of one fault cannot be assumed to reduce the prevailing failure probability .

[14] The *a priori* probability of encountering routine and rare events is computed as in the example shown in Table 7.3-1.

The last two of these questions have to be answered by test, and the types and scope of test are essentially those discussed in the preceding two sections. Specific test cases for validation of the commercial product must address the modifications and safeguards. The overall operation of the commercial software as part of the computer system will be validated in the system test as described in Sections 7.2 and 7.3, and specifically including statistical testing.

The validation of the failure rate (reliability) and configuration control can be conducted on the basis of documentation furnished by the vendor. The documentation must establish:

a. that it pertains to the commercial software product to be used as part of the computer system. Differences (e. g., experience on an earlier release) have to be identified, and the applicability of the data has to be justified.

b. that it is the latest available information. Vendor certification with a date close to that of the start of system validation is acceptable.

c. that failure data resulted from a positive response (such as a statement of the number of failures observed over a time interval) rather than from lack of a negative response (not having heard of any failures)

d. that the vendor has agreed to notify the user of (a) all significant failure reports, and (b) all new releases and of the reasons for these.

7.5 CONCLUSIONS AND RECOMMENDATIONS

This chapter considered validation as being conducted at the system level (computer system or higher), with end-to-end testing being the major activity. This is the last bulwark against placing an inadequate or faulty system into operation. Validation is a comparison of system capabilities against the requirements. Everything that has been said about the importance of verification of requirements in the preceding chapter applies here also. In addition, validation of requirements by means of animation or simulation is a valuable stepping stone that can reduce the probability of encountering serious problems in the system level validation. Validation uses the products of the verification process to establish that the system development has been carried out in accordance with an acceptable process, and that discrepancies discovered during reviews and pre-system testing have been corrected.

A combination of functional, structural, and statistical testing is recommended. Preferably all tests are carried out with a test harness that permits measurement of structural coverage and that identifies untested paths in critical portions of the program and at least branches and conditions in non-critical parts. Functional testing is primarily aimed at establishing that all requirements are implemented, structural testing identifies paths or branches that have not been accessed during functional test (and that could lead to unwanted actions), and statistical testing is conducted to establish the reliability of the system, and as further safeguard against unintended functions.

The most significant issue in validation is to determine how much test is required, i. e. to identify a criterion for test termination. The implicit termination criteria for functional and

structural test (e.g., to access every requirement or every branch) are not sufficient for high integrity computer systems because they do not include testing for coincident requirements or combinations of branch conditions. To overcome these limitations, statistical testing in an environment that generates test cases corresponding to multiple rare conditions has been recommended, and a test termination criterion for this type of test has been developed in Section 7.3.3. This criterion, while not rigorous, provides an objective means of establishing that the goals of validation have been attained. Further research and experimentation on the criterion and on the integrated approach to use of the three test methodologies is recommended.

Since regulators are rarely in a position to conduct tests themselves, the key activities are

- review of test plans: provision for functional, structural, and statistical testing

- test termination criteria: consistent with the recommendations of Section 7.3

- test reports: compliance with approved plans and test specifications, use of appropriate tools, identification of difficulties encountered and explanation of their potential effect on plant safety, assurance of adequate retest after modification of any part of the software (including requirements through code and documentation).

It is reasonable to insist that all documentation furnished in connection with validation be understandable by a person not familiar with the specific development and test techniques or tools used by the performing organization.

SECTION 8 - STANDARDS FRAMEWORK

8.1 OVERVIEW

This chapter responds to paragraph 4.1.2 of the Statement of Work which reads in part:

> Review existing V&V methods, guidelines, and standards in the United States as well as in other countries. (For example, review RG1.152 "Criteria for Programmable Digital Computer System Software in Safety Related Systems of Nuclear Power Plants" and ANSI/IEEE Std. 1012-1986 "Software Verification and Validation").... Study the results of a recent effort conducted by the National Institute for Standards and Technology (NIST) on high integrity software standards. Prepare a framework based on [the investigations contracted for here] and the NIST work that forms the basis for verification and validation guidelines..

In this chapter the term *standards* (lower case) includes recommended practices and guideline documents issued by standards organizations. The use of standards for defining verification and validation activities and products is highly desirable because they

- represent consensus among the interested parties
- promote uniformity of practice, thereby reducing familiarization effort and permitting transfer of experience from one application to the next
- reduce dependence on *ad hoc* requirements and regulations, thus limiting the risk about acceptability of proposed procedures.

For these reasons standards have been extensively referred to in the preceding chapters of this report. The purpose of the present chapter is to investigate the feasibility of a framework that clearly propagates the statutory and operational safety requirements into verification and validation practices. The most desirable outcome of this investigation is to identify a few standards of broad scope that in turn reference more detailed standards for individual activities and documents. This goal was not attained, and, on the contrary, the finding is that current standards represent a patchwork with considerable gaps, overlaps, and inconsistencies.

The section immediately following describes requirements for the top level of a standards framework and outlines the major shortcomings of present documents vis-a-vis the requirements. Section 8.3 presents detail topics that should be included at the lower levels of the framework. Sections 8.2 and 8.3 together form the conclusions of this chapter, and a separate conclusions section is therefore not provided.

8.2 TOP LEVEL OF THE FRAMEWORK

8.2.1 Requirements for the Top Level

The requirements for the top level of the framework arise from three areas:

- implementation of statutory provisions from 10CFR50
- conformance with best prevailing software practices as represented by standards
- the need for economical procurement and operation of digital protection systems on the part of the user (utilities).

The provisions of 10CFR50 do not address digital implementations of plant protection functions, and thus provide only very general guidance for the acceptability of software or integrated hardware/software products. Current software standards do not specifically address the needs of high integrity applications, such as nuclear plant protection systems. And the economic environment in which utilities operate has caused the transition from analog to digital protection systems to be undertaken in a piecemeal fashion in which at a given time practically every plant represents a different configuration of operational and protection equipment. These circumstances combine to create considerable difficulties in arriving at a standards framework that clearly implements statutory requirements, makes use of accepted commercial software practices, and is widely applicable to the prevailing state of the power plants.

The alternatives that have been considered for arriving at a top level framework are:

1. an integrated, self-contained, top level document, directly traceable to 10CFR50 requirements
2. acceptance of a suitable existing standard, with tailoring at the more detailed levels
3. tailoring of a suitable existing standard at the top level.

A model of the first alternative is the Ontario Hydro "Standard for Software Engineering of Safety Critical Software", which is a technically suitable document from which substantial excerpts have been incorporated in this report, particularly in Chapter 6. The standard is a company document and does not represent consensus, a deficiency that seems inherent in any attempt to generate a self-contained standard. The generation of the document at OH was facilitated by its application to a single known plant configuration and implementation by a single vendor. Any attempt to use this approach in the U. S. environment will have to face many difficulties because of the diversity of plants, major equipment suppliers, and specialty equipment suppliers. It is therefore not likely to be successful.

The second alternative is attractive because it builds on a consensus document while providing considerable freedom in the selection of features to be standardized. A number of current or soon to be issued standards claim to conform to the key provisions of 10CFR50. The Nuclear

Regulatory Commission's Regulatory Guide 1.152 strongly implies that ANSI/IEEE-ANS 7-4.3.2-1982 meets statutory requirements. This alternative also has the potential of giving access to the current software practices via tailoring of subsidiary documents, but it falls very significantly short of meeting the user's need for economical procurement and operation of protection systems. The principal shortcomings in that respect of all currently accepted standards is failure to establish reduced verification requirements for diverse implementations and for isolated segments not directly involved in the protection service, and to provide specific guidance for commercial dedication, issues that are discussed previously in this report, particularly in Chapter 2.

8.2.2 Recommended Structure

The third alternative, tailoring of a suitable existing standard, requires more time and effort than the first two, but holds promise of overcoming the difficulties outlined above and is therefore recommended. Non-exclusive examples of existing or pending standards that may serve as a baseline are the Draft IEEE 7-4.3.2 (1993) and IEC 1226. The key provisions that need to be tailored in are definitions of diversity and isolated segments, and reduced verification requirements for these as well as for dedication of commercial functions that have an established reliability history. It is not believed that such provisions will be in conflict with the interests of any group that currently participates in the related standards efforts. The issues need to be raised and considered, and effort will be required to achieve consensus, but ultimately this should result in gains (or, at least, no losses) to the affected parties.

If this recommendation is adopted, classification will become a part of the top level framework. Because there are no existing standards for commercial dedication, it appears desirable to add several pertinent topics to the top level framework. The principal provisions affect definition of the service to be performed, service experience, and procurement concerns. The issues to be covered in these areas are outlined below.

Definition of Service

The definition of service is essential to provide the proper environment for verification and validation of the digital system. A separate definition is required for each plant safety service (e. g., one definition for control rod actuation and a separate one for emergency feedwater supply). All functions that do not directly affect a plant safety service should be separately defined. The definition(s) can be tabular or text, supplemented by timing diagrams and listing of allowed state transitions. If previous service experience is claimed, the definition(s) should identify differences between the previous service and that proposed to be provided. Required topics are:

- Method and frequency of invocation (cyclic, by event, operator command, etc.)

- Possible states of the controlled plant at time of invocation

- Consequences of failure of service

- Mechanism for detecting failure

- Redundancies and other means for mitigation of failure

Service Experience

This heading is intended to provide assurance that the proposed system will not degrade the safety of the plant below current levels, and that its attributes are at least comparable to those of equivalent installations. Required data are:

- How is this service currently provided at this plant and failure history
- How is this service currently provided at similar U. S. plants and history
- How is this service currently provided at similar foreign plants and history
- Significant differences of the proposed service from those above

Procurement Concerns

In the procurement of established products, including off-the-shelf items, at least the following provisions should be included.

- Acceptance test -- the acceptance should establish unambiguously that the item meets all user requirements; test results should be documented
- Configuration control -- there should be assurance that all procured items are exactly like the one for which service experience is claimed and on which the acceptance test was run.
- Vendor internal quality control -- sufficient information should be obtained to assess the level of internal quality control maintained by the vendor.
- Notice of discrepancies or failures -- agreement should be obtained from the vendor for prompt notification of any discrepancies in the product that were found either in its own activities or were reported to it by outside sources.
- Use of discrepant products -- there should be a written procedure on use of a product for which discrepancies had been reported.
- Staffing levels -- the project plan should identify staffing for monitoring activities, and these should be compared to (a) similar recent efforts by the developer, and (b) industry norms for critical software. The professional qualifications of the monitoring staff should be equivalent to those of a design team for a comparable product.

8.2.3 Additional Information to be Supplied

Voluntary standards organizations frequently cannot achieve consensus on specific requirements for a process or product and then restrict documents to generic or planning topics. Examples of this practice that are pertinent to high integrity systems are the following IEEE Standards

730 Software Quality Assurance Plans

828 Software Configuration Management Plans

1012 Verification and Validation Plans

1228 Software Safety Plans

These standards provide a desirable structure for the conduct for important assurance activities *but they do not define requirements.* Therefore at the top level, or in a subsidiary document, specific objectives and levels of attainment for the controlled activities must be specified.

8.3 LOWER LEVEL STANDARDS FRAMEWORK

This heading discusses a standards framework for system and software attributes or practices in areas where high integrity systems can make use of established documentation. Most of the sub-headings therefore refer to suitable documents and make recommendations for tailoring or supplementation.

8.3.1 Life Cycle Phases

For software development a recursive (spiral) life cycle is highly desirable (see Chapter 5). From the user's or regulator's point of view the phases usually collapse into an unconventional sequence of

- Requirements formulation
- Procurement (development or product evaluation)
- Licensing activities
- Acceptance testing

IEEE Standard 1074-1991 "Developing Software Life Cycle Processes" can be used for general guidance. Additional information can be found in IEEE 1058.1-1987 "Software Project Management Plans".

8.3.2 System and Software Requirements

The requirements must consider both function and attributes. Suitable references are IEEE Std. 830-1984, "Guide for Software Requirements Specifications" and par. 3.4.2 "Software Safety Requirements Analysis" of IEEE Std. 1228 "Software Safety Plans". Specific requirements for this phase discussed in Section 6.4.1.1 of this report will fit into this framework.

Requirements documentation should be under configuration control (see below).

8.3.3 Software Development or Procurement

For software design documentation, reference should be made to IEEE Std. 1016-1987 "Recommended Practice for Software Design Descriptions" which is suitable for both previously and newly developed software. Guidance on safety aspects can be found in par. 3.4.3 "Software Safety Design Analysis" of IEEE Std. 1228. Specific requirements for this phase applicable to newly developed software and discussed in Section 6.4.1.2 of this report fit into this framework.

The design should be under configuration control.

The source code for high integrity systems should be written in a standardized language for which tool support is available. It should be produced in accordance with the developer's Standards and Procedures Handbook (or equivalent). Unit test should be conducted in accordance with IEEE Std. 1008-1987 "Standard for Software Unit Testing". The code should be reviewed prior to undergoing system test in accordance with IEEE Std. 1028-1988 "Standard for Software Reviews and Audits". Safety critical segments of newly developed code should be subjected to par. 3.4.4 of IEEE Std. 1228 "Software Safety Code Analysis". Specific requirements for code discussed in sections 6.4.1.3 and .4 of this report fit into this framework.

The code should be under configuration control.

8.3.4 Licensing Activities

In support of licensing software quality assurance should be conducted in accordance with a plan that conforms to IEEE Std. 730-1989 "Standard for Software Quality Assurance Plans". Configuration management should be conducted in accordance with IEEE Std 828-1990 "Standard for Software Configuration Management Plans" and IEEE Std. 1042-1987 "Guide to Software Configuration Management". Verification and Validation should be conducted in accordance with IEEE Std. 1012-1986 "Standard for Software Verification and Validation Plans". All of these standards need tailoring and specific information in order to be applicable to high integrity systems.

A suitable framework for software security could not be identified. In lieu of this the following is suggested:

Measures to protect the software against negligence or pranks among authorized personnel, and against malicious acts of both authorized personnel and outsiders should be in place.. Protection

against outsiders is primarily concerned with access control. Protection against harmful acts by insiders, whether intended or not, depends on management supervision of all critical operations. The following are minimum required safeguards:

- Protection against unauthorized physical or functional access -- this implies access checklists
- Security and safety functions embedded in the code must not be bypassable
- Protection of data against loss, tampering, and unauthorized access
- Witnessing of all operations that can compromise security by at least one member of management -- this implies checklists of critical operations

8.3.5 Acceptance Testing

Testing of newly developed software should conform to par. 3.4.5 "Software Safety Test Analysis" of IEEE Std. 1228. The validation requirements discussed in Chapter 7 of this report fit into that framework. Testing of non-developed software should be conducted in accordance with par. 3.3.11 of IEEE Std. 1228 "Previously Developed or Purchased Software". Software test documentation should comply with IEEE Std. 829-1983 "Standard for Software Test Documentation"

8.3.6 Other Activities

User documentation should conform IEEE Std. 1063-1987 "Standard for Software User Documentation".

Software changes after acceptance testing should conform to par. 3.4.6 "Software Safety Change Analysis" of IEEE Std. 1228.

A Software Safety Hazards Analysis should be furnished that complies with par. 50.2.12 of MIL-STD-882B "System Safety Program Requirements".

SECTION 9 - SUMMARY CONCLUSIONS AND RECOMMENDATIONS

The major conclusion of this report is that verification and validation are open-ended activities without natural completion criteria. Where a limited set of tasks has been defined, this has been based on experience or subjective evaluation of the decisions maker, on resource limitations, or on a combination of these. While methodologies and tools are probably available to verify the absence of any one cause of safety impairment, there is no practicable set that will cover all possible causes. In this environment it is very important that all possible feedback mechanisms be utilized to improve our knowledge of (a) causes of failures, (b) effectiveness of specific methodologies against these causes, and (c) resource requirements of the methodologies. At present the state of knowledge in the nuclear power field with respect to each of these topics leaves much to be desired.

In this report causes of failures in other high integrity applications have been identified, and appropriate detection and correction methodologies have been described. But there are many unique factors at work in nuclear protection systems, and the reliance that can be placed on the imported data is therefore limited. Systematic collection of data is recommended on failures or discrepancies in digital systems for nuclear power plants, together with identification of the environment in which they occur, and of the resources that were used for development and verification. The background on these activities is provided in Chapter 3.

One of the most significant differences between nuclear power applications of high integrity systems and those in other environments is that in the latter the user typically funds and controls verification and validation, while in the nuclear field much of this has in the past been left to the developer. As a result truly independent reviews of the software occur late in the development stage and are frequently restricted to an audit of the verification undertaken by the developer. This approach obviously restricts the visibility of the independent team, as well as the scope of possible corrective actions. These difficulties are described in Chapters 6 and 7. Backward reconstruction techniques are discussed which have the potential of comprehensive product verification at late stages in the development. These techniques and the associated tools are emerging from research and cannot be immediately applied to arbitrary software products. But because of the very specific suitability for the nuclear power environment further investigations of their use are recommended.

Because of the open-ended nature of the verification process, and the specific circumstances in the nuclear field, it is very difficult to generate broadly applicable criteria for practical verification, and the past practice of individual assessment of each case has caused uncertainty about outcomes and scheduling of the review which have a negative impact on further development of digital protection systems for nuclear plants. The limitations of formal methods for verification of practical high integrity software are discussed in Section 6.3.4. Specificaly, no verifiable claims for reduction of failures in the operational environment could be found. At the present state of knowledge about potential causes of failure, and about adequate and practicable verification methodologies, it may at times be cost effective to employ functionally diverse systems, if each one can be qualified by a more limited verification process, and if the

independence of the implementations can be assured. This avenue has not been widely pursued, partly because current classification practices make no distinction between a single protection system and two diverse ones. The issues and possible solutions are discussed in Chapter 2 and Appendix A (which is referenced there).

The verification methodologies discussed in sections 6.4 and 6.5 can establish with reasonable effort that software products intended for high integrity applications are reviewable and materially in conformance with requirements. The validation methodologies discussed discussed in sections 7.2, 7.3, and 7.4 can establish operational compliance with requirements and demonstrate with reasonable test effort and currently available test tools that the probability of failure on demand is not greater than 10^{-3} or (with greater effort and advanced tools) 10^{-4}. Where the dependability requirements exeed these values the use functionally diverse approaches appears to be the most economical approach.

Validation of digital systems depends heavily on test, and the dictum that you can prove the presence of bugs by test but never their absence has not yet been repealed. However, in Chapter 7 a methodology has been developed that holds promise of showing by test that the failure probability of a digital system is below a selected threshold. The methodology departs from exclusively statistical reasoning by also making use of the types of failures that are being encountered. Specifically, when the only failure occurrences are due to multiple rare conditions in the input data or the computer environment, a much smaller number of test cases than would be required by pure dependence on statistics, can provide assurance of meeting reasonable reliability goals.

REFERENCES

ADAM84	D.M. Adams and J.M. Svoboda, "Interim Criteria for the Use of Programmable Digital Devices in Safety and Control Systems", EG&G Report to U.S. Nuclear Regulatory Commission, NUREG/CR-4017, EG&G-2348, December, 1984.
AIAA92	American Institute of Aeronautics and Astronautics, "Recommended Practice - Software Reliability", ANSI/AIAA R-013-1992.
AMOR73	W. Armory and J. A. Clapp, "A Software Error Classification Methodology", generated by The MITRE Corporation under USAF Contract F19628-73-C-0001, June 1973.
AVIZ82	Algirdas Avizienis, "The Four-Universe Information System Model for the Study of Fault Tolerance", *Digest of Papers 12th Fault Tolerant Computing Symposium,* IEEE Cat 82CH1760-8, pp. 1157-1191.
AVIZ87	A. Avizienis, "A Design Paradigm for Fault Tolerant Systems", *Proceedings of the AIAA Computers in Aerospace VI Conference,* Wakefield, Mass., October 1987.
BAIL81	C. T. Bailey and W. L. Dingee, "A software study using Halstead metrics," in Proceedings of the 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, Vol. 10, pp. 189-197, March. 1981.
BIER69	H. Bierman, C. P. Bonini, and W. H. Hausman, *Quantiative Analysis for Business Decisions,* Richard D. Irwin, Inc., Homewood IL 1969
BISH90	P. G. Bishop, ed., *Dependability of critical computer systems 3 -- Techniques directory,* Elsevier Applied Science, ISBN 1-85166-544-7, 1990.
BOEH73	B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. McLeod, and M. J. Merritt, *Characteristics of Software Quality,* TRW-SS-73-09, TRW Systems, Systems Engineering and Integration Division, Redondo Beach CA, December 1973.
BOEH81	Barry W. Boehm, *Software Engineering Economics,* Prentice-Hall Inc., 1981.
BOWE85	T. P. Bowen, G. B. Wigle, and J. T. Tsai, "Specification of software quality attributes," Final Technical Report RADC-TR-85-37, Rome Air Development Center, New York, February. 1985 (three volumes).

BROW91 R.A. Brown, "Guideline for the Categorization of Software in Ontario Hydro's Nuclear Facilities with respect to Nuclear Safety", Memo to Distribution, Ontario Hydro, Toronto, Ontario, Canada, July, 1991.

CONT86 S. D. Conte, H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models,* Benjamin/Cummings, Menlo Park CA, 1986

COOP93 "Proceedings of the Cooperstown I Workshop - Creating a National Vision and Force in Software through Software Measurement", obtainable from the Data and Analysis Center for Software (DACS), USAF Rome Laboratory, August 1993.

CRAI93 Dan Craigen, Susan Gerhart, and Ted Ralston, "An International Survey of Industrial Applications of Formal Methods", (2 vols.), NIST GCR 93/626, March 1993

DAHL90 Gustav Dahll, Mel Barnes, and Peter Bishop, "Software Diversity -- A Way to Enhance Safety?", *Proc. Second European Conference on Software Quality Assurance,* Oslo, Norway, May 30 - June 1, 1990.

DORF90 M. Dorfman and R. H. Thayer, "Standards, Guidelines, and Examples on System and Software Requirements Engineering", IEEE Computer Society Press, 1990.

ECKH91 D. E. Eckhardt, A. K. Caglayan, J. C. Knight, et al., "An experimental evaluation of software redundancy as a strategy for improving reliability", *IEEE Trans. Software Engineering,* Vol 17, No 7, July 1991, pp. 692 - 702.

FARR83 William H. Farr, "A Survey of Software Reliability Modeling and Estimation", NSWC TR82-171, Naval Surface Weapons Center, Dahlgren VA, 22448, September 83.

FARR85 W.H.Farr and O.D.Smith, "Statistical Modeling and Estimation of Reliability Functions for Software", Users Guide, NSWC TR 84-373, April 1985.

FENT93 Norman Fenton. "How effective are software engineering methods?", *Journal of Systems and Software,* Vol. 23, No. 2, pp. 141 - 146, August 1993

FISC91 H.D. Fischer, A. Graf, and U. Mertens, "Siemens-KWU works toward digital I&C for safety systems", *Nuclear Engineering International,* February, 1991, p. 35.

GIFF84 David Gifford & Alfred Spector, "The TWA Reservation System", *Communications of the ACM*, pp. 650-665, Vol 2, No. 27, July 1984.

GLAS81 R. G. Glass, "Persistent Software Errors", *IEEE Transactions on Software Engineering*, SE-7 No. 2, March 1981.

GOEL79 Amrit Goel and K. Okumoto, "Time-Dependent Error Detection Rate for Software Reliability and Other Performance Measures", *IEEE Transactions on Reliability*, Vol R-28 No. 3, pp. 206-211, 1979.

HALL91 J. V. Hall, *Software Development Methods in Practice*, Elsevier, 1991

HALS77 M. H. Halstead, *Elements of Software Science.* New York: Elsevier North-Holland, 1977.

HAML90 Dick Hamlet, "Partition Testing Does not Inspire Confidence", *IEEE Trans. Software Engineering,* Vol 16, No. 12, pp. 1402 - 1411, December 1990.

HECH86 Herbert Hecht and Myron Hecht, "Software Reliability in the System Context", *IEEE Transactions on Software Engineering*, January 1986.

HECH92 M. Hecht, S. Chau and G. Dinsmore, "Analysis of 1991 AAS PTRs", Vol 1., prepared under FAA Systems Engineering and Technical Assistance Contract DTFA01-90-C-00013 by SoHaR Incorporated, December 1992.

HECH93 Herbert Hecht, "Rare Conditions -- An Important Cause of Failures", *Proc. COMPASS'93,* June 1993.

HECH93A H. Hecht, A. Tai, and K. S. Tso, *Class 1E Digital System Studies,* NUREG/CR-6113, July 1993

HENR81 S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Software Engineering*, Vol. SE-7, pp. 510-518, September, 1981.

HERS93 A. Hersh, "Where's the Return on Process Improvement?", *IEEE Software*, Vol 10, No. 4, July 1993, p. 12

HOWD78 W.E. Howden, "An Evaluation of the Effectiveness of Symbolic Testing", *Software-Practice and Experience,* Vol 8, pp. 381-397, John Wiley & Sons, 1978.

HUGH92 G. Hughes and R. S. Hall, "Recent Development in Protection and Safety-Related Systems for Nuclear Electric's (UK) Power Plant", *Tokyo Conference of Computers in Nuclear Power Plants*, May 1992.

IAEA84 International Atomic Energy Agency, "Safety Related Instrumentation and Control Systems for Nuclear Power Plants", IEAE Safety Guide NO. 50-SG-D8, Vienna, 1984.

ICHB79 J. D. Ichbiah et al., "Rationale for the design of the Ada programming language," ACM SIGPLAN Notices, Vol. 14, June 1979.

JENS85 H. A. Jensen and K. Vairavan, "An experimental study of software metrics for real-time software," IEEE Trans. Software Engineering, Vol. SE-11, pp. 231-234, Feb. 1985.

JOAN93 P. Joannou, "Pickering NGS-B, Digital Trip Meter, Procedure for the Specification of Safety Critical Software", Ontario Hydro Ltd., January 1993.

KANO87 K. Kanoun and T. Sabourin, "Software Dependability of a Telephone Switching System", *Digest, FTCS-17,* June 1987.

LEVE86 N.G. Leveson, "Software Safety: What, Why and How", *ACM Computing Surveys*, No.2, Vol.18, PP.125, June 1986.

LIND89 R. K. Lind and K. Vairavan, "An experimental investigation of software metrics and their relationship to software development effort," IEEE Trans. Software Engineering, Vol. SE-15, pp. 649-653, May 1989.

MCCA76 T. J. McCabe, "A complexity measure," IEEE Trans. Software Engineering, Vol. SE-2, pp. 308-320, December 1976.

MCCA77 J. A. McCall, P. K. Richards, and G. F. Walters, *Factors in Software Quality,* RADC-TR-77-369 (3 Vols.), November 1977.

MUSA87 John Musa, Iannino and Okumoto, "Software Reliability Measurement Prediction, Application", McGraw-Hill Co. 1987.

NEUM93 Peter G. Neumann, "Myths of Dependable Computing: Shooting the Straw Herrings in Midstream", *Proc. COMPASS'93,* pp. 1 - 4, Gaithersburg, MD, June 1993.

OVIE80 E. Oviedo, "Control flow, data flow and program complexity," in Proceedings of COMPSAC'80, pp. 146-152, 1980.

PARN91 D. L. Parnas, G. J. K. Asmis, and J. Madey, "Assessment of Safety-Critical Software", *Proc. Ninth Annual Software Reliability Symposium,* Colorado Springs CO, May 1991.

PARN93 David L. Parnas, "Some theorems we should prove", Telecommunications Research Institute of Ontario (TRIO), Technical Report, June 1993.

PAUL91 M. Paulk et al., "Capability maturity model for software, "Technical Report CMU/SEI-91 TR-24, Software Engineering Institute, August, 1991.

RUSH92 John Rushby, "Formal Methods for Dependable Real-Time Systems", *Proc. International Symposium on Real-Time Embedded Processing for Space Applications,* Les Saints-Maries-de-la-Mer, France, November 1992.

SCHN75 Norman F. Schneidewind, "Analysis of Error Processes in Computer Software", *Sigplan Notices*, Vol. 10 No. 6, pp. 337-346, October, 1975.

TANG92 D. Tang and R. K. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems", *IEEE Transaction on Computers,* Vol. 41, No.5, pp. 567 - 577, May 1992.

THAY76 T. A. Thayer, et al, "Software Reliability Study", RADC-TR-76-238, Final Technical Report, pp. 5-13, 61-49, August 1976.

TSO91 K. S. Tso, "Complexity metrics for avionics software," Final Report under WL/AAAF-3 Contract F33615-91-C-1753, SoHaR Incorporated, Beverly Hills, CA, October, 1991.

VELA84 P. Velardi and R. K. Iyer, "A Study of Software Failures and Recovery in the MVS Operating System", *IEEE Transactions on Computers, Special Issue on Fault Tolerant Computing*, Vol. C-33 No. 7, July 1984

VIFT93 (Team Authored Report) "Report of the Voice Switching and Control System (VSCS) Independent Fault Tolerance Analysis Team (VIFTAT)", prepared for the Federal Aviation Administration, January 1993.

VOAS92 Jeffrey M. Voas and Keith W. Miller, "Improving the Software Development Process using Testability Research", *Proc. Third International Symposium on Software Reliability Engineering*, Research Triangle Park, NC, October 1992

WEBE91 C. V. Weber, M. C. Paulk, C. J. Wise, and J. V. Withey, "Key practices of the capability maturity model," Technical Report CMU/SEI-91-TR-25, Software Engineering Institute, August. 1991.

STANDARDS REFERRED TO

Standards Symbol (Issuer)(first use only)	Title of Standard
ANS-50.1* (American Nuclear Soc.)	"Nuclear Safety Design Criteria for Light Water Reactors", January 1993
ANS-58.14*	"Safety and Pressure Integrity Classification Criteria for Light Water Reactors", January 1993
ASME NQA-2a (Am. Soc. of Mechanical Engineers)	"Quality Assurance Requirements for Nuclear Facility Applications", 1990
IEC 987 (International technical Commission)	"Programmed Digital Computers Important to Safety of Nuclear Power Stations", 1989
IEC 1226	"Nuclear Power Plants -- Instrumentation and Control Systems Important to Safety -- Classification", 1993
IEC 65A(Sec)94*	"Software for Computers in the Application of Industrial Safety-Related Systems," 6 December 1989.
IEC Publication 880	"Software for Computers in the Safety Systems of Nuclear Power Stations," IEC Publ. 880, 1986.
IEEE Std 279-1980** (Inst. of Electrical & Electronic Engineers)	"Criteria for Protection Systems for Nuclear Power Generating Stations"
IEEE Std 308-1980	"Criteria for Class 1E Electric Systems for Nuclear Power Generating Stations"
IEEE Std 323-1983	"Qualifying Class 1E Equipment for Nuclear Power Generating Stations"
IEEE Std 379-1977	"Application of the Single Failure Criterion to Nuclear Power Generating Station Class 1E Systems"
IEEE Std 500-1984	"Guide to the Collection and Presentation of Electrical, Electronic, Sensing Component, and Mechanical Reliability Data for Nuclear Power Generating Stations"
IEEE Std 603-1991	"Criteria for Safety Systems for Nuclear Power Generating Stations"

IEEE Std 730-1989	"Software Quality Assurance Plans"
IEEE Std 828-1990	"Software Configuration Management Plans"
IEEE Std 829-1983	"Software Test Documentation"
IEEE Std 830-1984	"Guide for Software Requirements Specification"
IEEE Std 1008-1987	"Software Unit Testing"
IEEE Std 1012-1986	"Software Verification and Validation Plans"
IEEE Std 1016-1987	"Recommended Practice for Software Design Descriptions"
IEEE Std 1028-1988	"Software Reviews and Audits"
IEEE Std 1042-1987	"Guide to Software Configuration Management"
IEEE Std 1058.1-1987	"Software Project Management Plans"
IEEE Std 1063-1987	"Software User Documentation"
IEEE Std 1074-1991	"Developing Software Life Cycle Processes"
IEEE Std 1228-1992	"Software Safety Plans"
IEEE/ANS 7-4.3.2-1993	"Criteria for Digital Computer Systems in Safety Systems of Nuclear Power Generating Stations"
MIL-STD-882B	Military Standard, "System Safety Program Requirements", (Dept. of Defense) AMSC Number F3329, March 1984
MIL-STD-1629A	"Procedures for Performing a Failure Mode, Effects and Criticality Analysis", 24 November 1980 (w/Notices 1 & 2)
NIST Special Pub 500-204 (National Institute of Standards & Technology)	"High Integrity Software Standards Guidelines"
Ontario Hydro Standard	"Standard for Software Engineering of Safety Critical Software," Ontario Hydro/AECL CANDU, issued for one year trial use, 21 December 1990.
RTCA DO-178B (Requirements & Technical Concepts for Aviation)	"Software Considerations in Airborne Systems and Equipment Certification", 1992

UK MOD 00-55 (Ministry of Defence Standard)	"The Procurement of Safety Critical Software in Defence Equipment," 5 April 1991.
UK MOD 00-56	"Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment", 5 April 1991.
10 CFR Part50 (U. S. Government)	"Domestic Licensing of Production and Utilization Facilities" Code of Federal Regulations, Vol 10 (Energy), Part 50, (1986)

NRC FORM 335 (2-89) NRCM 1102, 3201, 3202

U.S. NUCLEAR REGULATORY COMMISSION

BIBLIOGRAPHIC DATA SHEET

(See instructions on the reverse)

1. REPORT NUMBER (Assigned by NRC. Add Vol., Supp., Rev., and Addendum Numbers, if any.)

NUREG/CR-6293
Vol. 1

2. TITLE AND SUBTITLE

Verification and Validation Guidelines for High Integrity Systems

Main Report

3. DATE REPORT PUBLISHED

MONTH	YEAR
March	1995

4. FIN OR GRANT NUMBER

L2448

5. AUTHOR(S)

H. Hecht, M. Hecht, G. Dinsmore, S. Hecht, D. Tang

6. TYPE OF REPORT

Technical

7. PERIOD COVERED (Inclusive Dates)

8/27/92-12-31/93

8. PERFORMING ORGANIZATION — NAME AND ADDRESS (If NRC, provide Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address; if contractor, provide name and mailing address.)

SoHaR Incorporated
8421 Wilshire Boulevard Ste 201
Beverly Hills, CA 90211-3204

Under Contract to:
Harris Corporation
Information Systems
P.O. Box 9800, Melbourne, FL 32902

9. SPONSORING ORGANIZATION — NAME AND ADDRESS (If NRC, type "Same as above"; if contractor, provide NRC Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address.)

Division of Systems Technology
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

10. SUPPLEMENTARY NOTES

11. ABSTRACT (200 words or less)

High integrity systems include all protective (safety and mitigation) systems for nuclear power plants, and also systems for which comparable reliability requirements exist in other fields, such as in the process industries, in air traffic control, and in patient monitoring and medical systems. Verification aims at determining that each stage in the software development completely and correctly implements requirements that were establsihed in preceding phase, while validation determines that the overall performance of a computer system completely and correctly meets system requirements. Volume I of the report reviews existing classifications for high integrity systems and for the verification and validation procedures, based on assumptions about the environment in which these systems will be utilized. The final chapter of Volume I deals with a framework for standards in this field. Volume II contains appendices dealing with specific methodologies for system classification, for dependability evaluation, and for two software tools that can automate otherwise very labor intensive verification and validation activities.

12. KEY WORDS/DESCRIPTORS (List words or phrases that will assist researchers in locating the report.)

Verification
Validation
Computer System Integrity
Software Test
Software Reliability
Error Classification

13. AVAILABILITY STATEMENT

Unlimited

14. SECURITY CLASSIFICATION

(This Page)

Unclassified

(This Report)

Unclassified

15. NUMBER OF PAGES

16. PRICE