NIST Special Publication 500-234

Computer Systems Technology

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology

Reference Information for the Software Verification and Validation Process

PB96188164

Dolores R. Wallace
Laura M. Ippolito
Barbara B. Cuthill

REPRODUCED BY: NTIS
U.S. Department of Commerce
National Technical Information Service
Springfield, Virginia 22161

The National Institute of Standards and Technology was established in 1988 by Congress to "assist industry in the development of technology . . . needed to improve product quality, to modernize manufacturing processes, to ensure product reliability . . . and to facilitate rapid commercialization . . . of products based on new scientific discoveries."

NIST, originally founded as the National Bureau of Standards in 1901, works to strengthen U.S. industry's competitiveness; advance science and engineering; and improve public health, safety, and the environment. One of the agency's basic functions is to develop, maintain, and retain custody of the national standards of measurement, and provide the means and methods for comparing standards used in science, engineering, manufacturing, commerce, industry, and education with the standards adopted or recognized by the Federal Government.

As an agency of the U.S. Commerce Department's Technology Administration, NIST conducts basic and applied research in the physical sciences and engineering, and develops measurement techniques, test methods, standards, and related services. The Institute does generic and precompetitive work on new and advanced technologies. NIST's research facilities are located at Gaithersburg, MD 20899, and at Boulder, CO 80303. Major technical operating units and their principal activities are listed below. For more information contact the Public Inquiries Desk, 301-975-3058.

Office of the Director
- Advanced Technology Program
- Quality Programs
- International and Academic Affairs

Technology Services
- Manufacturing Extension Partnership
- Standards Services
- Technology Commercialization
- Measurement Services
- Technology Evaluation and Assessment
- Information Services

Materials Science and Engineering Laboratory
- Intelligent Processing of Materials
- Ceramics
- Materials Reliability[1]
- Polymers
- Metallurgy
- Reactor Radiation

Chemical Science and Technology Laboratory
- Biotechnology
- Chemical Kinetics and Thermodynamics
- Analytical Chemical Research
- Process Measurements
- Surface and Microanalysis Science
- Thermophysics[2]

Physics Laboratory
- Electron and Optical Physics
- Atomic Physics
- Molecular Physics
- Radiometric Physics
- Quantum Metrology
- Ionizing Radiation
- Time and Frequency[1]
- Quantum Physics[1]

Manufacturing Engineering Laboratory
- Precision Engineering
- Automated Production Technology
- Intelligent Systems
- Manufacturing Systems Integration
- Fabrication Technology

Electronics and Electrical Engineering Laboratory
- Microelectronics
- Law Enforcement Standards
- Electricity
- Semiconductor Electronics
- Electromagnetic Fields[1]
- Electromagnetic Technology[1]
- Optoelectronics[1]

Building and Fire Research Laboratory
- Structures
- Building Materials
- Building Environment
- Fire Safety
- Fire Science

Computer Systems Laboratory
- Office of Enterprise Integration
- Information Systems Engineering
- Systems and Software Technology
- Computer Security
- Systems and Network Architecture
- Advanced Systems

Computing and Applied Mathematics Laboratory
- Applied and Computational Mathematics[2]
- Statistical Engineering[2]
- Scientific Computing Environments[2]
- Computer Services
- Computer Systems and Communications[2]
- Information Systems

---

[1]At Boulder, CO 80303.

[2]Some elements at Boulder, CO 80303.

NIST Special Publication 500–234

Reference Information for the Software Verification and Validation Process

Dolores R. Wallace
Laura M. Ippolito
Barbara B. Cuthill

Information Systems Architecture Division
Computer Systems Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899-0001

April 1996

PROTECTED UNDER INTERNATIONAL COPYRIGHT
ALL RIGHTS RESERVED.
NATIONAL TECHNICAL INFORMATION SERVICE
U.S. DEPARTMENT OF COMMERCE

U.S. Department of Commerce
Michael Kantor, Secretary

Technology Administration
Mary L. Good, Under Secretary for Technology

National Institute of Standards and Technology
Arati Prabhakar, Director

Reports on Computer Systems Technology

The National Institute of Standards and Technology (NIST) has a unique responsibility for computer systems technology within the Federal government. NIST's Computer Systems Laboratory (CSL) develops standards and guidelines, provides technical assistance, and conducts research for computers and related telecommunications systems to achieve more effective utilization of Federal information technology resources. CSL's responsibilities include development of technical, management, physical, and administrative standards and guidelines for the cost-effective security and privacy of sensitive unclassified information processed in Federal computers. CSL assists agencies in developing security plans and in improving computer security awareness training. This Special Publication 500 series reports CSL research and guidelines to Federal agencies as well as to organizations in industry, government, and academia.

National Institute of Standards and Technology Special Publication 500-234
Natl. Inst. Stand. Technol. Spec. Publ. 500-234, 90 pages (April 1996)
CODEN: NSPUE2

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1996

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, DC 20402

ABSTRACT

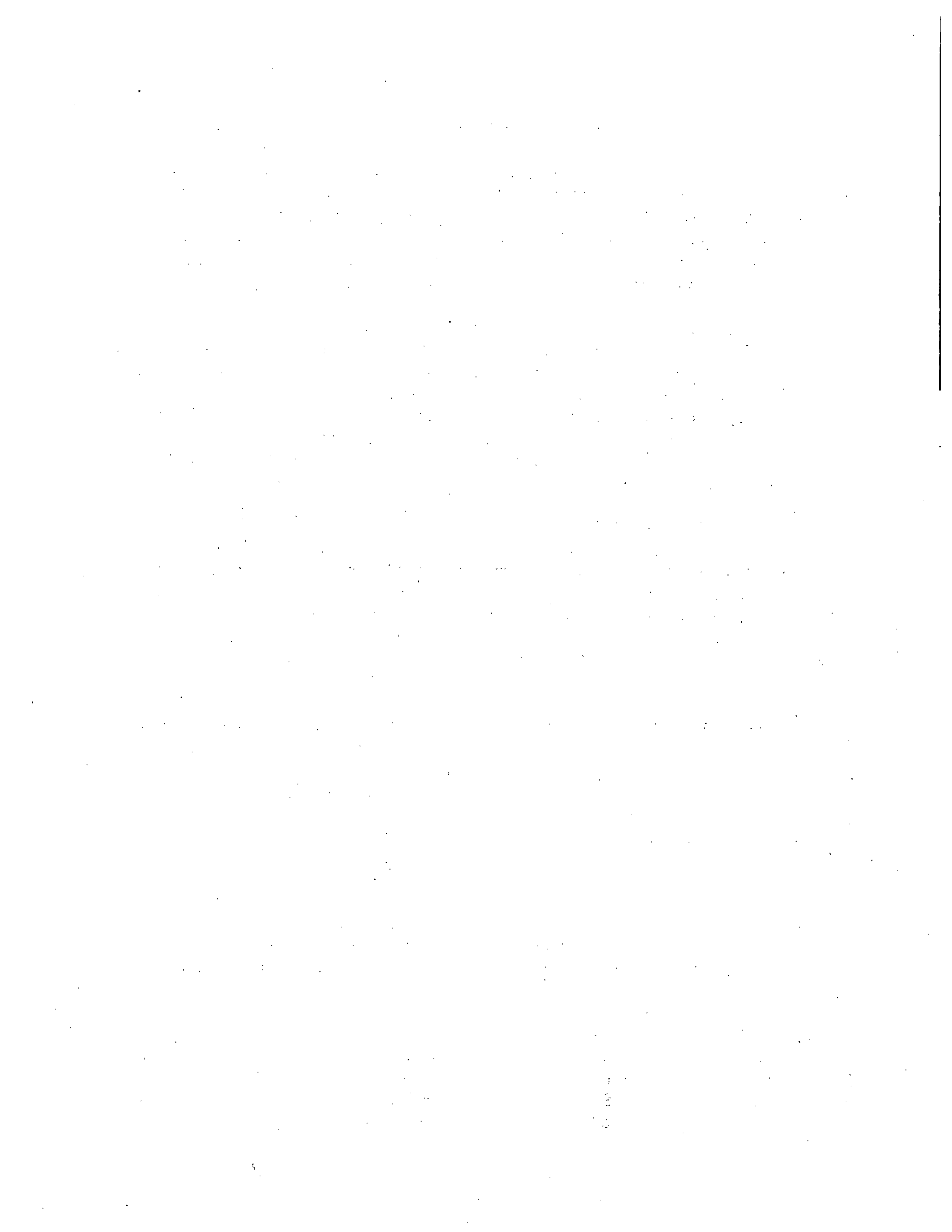
Computing systems may be employed in the health care environment in efforts to increase reliability of care and reduce costs. Software verification and validation (V&V) is an aid in determining that the software requirements are implemented correctly and completely and are traceable to system requirements. It helps to ensure that those system functions controlled by software are secure, reliable, and maintainable. Software V&V is conducted throughout the planning, development and maintenance of software systems, including knowledge-based systems, and may assist in assuring appropriate reuse of software.

KEYWORDS

Health care; independent verification and validation; knowledge-based systems; software reuse; software development; software diagnostic tools; software verification and validation.

ACKNOWLEDGMENTS

This report was funded by the Advanced Technology Program (ATP) of the National Institute of Standards and Technology (NIST) under Solicitation 94-04, Information Infrastructure for Health Care.



EXECUTIVE SUMMARY

Like many other industries in the United States, the health care industry is turning to computing systems to reduce administrative overhead, control escalating costs, and improve accuracy of stored information. New technology is affecting the form and usage of patient information, diagnostic tools, and the tools which provide treatment. In particular, the application of information technology is a promising enabler for transferring gains in medical science research to patient benefit, for ensuring appropriate availability of patient information, and for managing the billing processes.

Computing systems may be employed in the health care environment in efforts to increase reliability of care and reduce costs. Software verification and validation (V&V) is an aid in determining that the software requirements are implemented correctly and completely and are traceable to system requirements. (Software V&V does not verify the correctness of the system requirements, only that the software requirements can be traced to the system requirements.) It helps to ensure that those system functions controlled by software are secure, reliable, and maintainable. It uses a structured approach to analyze and test the software: It evaluates software against its requirements for quality attributes such as performance. Software V&V is conducted throughout the planning, development, and maintenance of software systems.

The major objective of the software V&V process is to determine that the software performs its intended functions correctly, ensure that it performs no unintended functions, and provide information about its quality and reliability. Software V&V evaluates how well the software is meeting its technical requirements and its safety, security and reliability objectives relative to the system. It also helps to ensure that software requirements are not in conflict with any standards or requirements applicable to other system components. Software V&V tasks analyze, review, demonstrate or test all software development outputs.

The guidelines in this report address V&V issues related to the recognition that different health care systems may:

- execute in real time (e.g., implantable medical devices and treatment devices);
- rely heavily on existing software;
- contain many units contributing to overall system complexity;
- incorporate knowledge-based systems (KBS) (e.g., diagnostic systems).

The software V&V process is tightly integrated with the software development process. For each activity in software development there is a corresponding software V&V activity to verify or validate the products of those activities. This report explains these relationships, the software V&V tasks supporting each activity, and the types of techniques that may be used to accomplish specific software V&V tasks.

Software V&V has long been employed on new development projects. Today, more and more systems are built using commercial off-the-shelf (COTS) software products, software components from sources external to the developer, and software from a previous version of a similar product built by the same organization. Some of the issues concerning software V&V for systems reusing any of these software types are addressed in this document.

Preceding page blank

The health care industry has been interested in, and made use of, artificial intelligence (AI) techniques by developing KBSs to understand the complex medical and patient data used for diagnosis. This interest has grown as the scale of the problem of managing data and knowledge has grown in the health care industry. While there are techniques available for V&V of the KBS which employ AI techniques, the V&V and AI communities still need to do more research especially in the areas of making knowledge maintenance easier and more reliable. These guidelines provide an overview of the issues in using V&V and KBS techniques.

ACRONYMS

AI	Artificial Intelligence
ATP	Advanced Technology Program
CASE	Computer-Aided Software Engineering
CBR	Case-Based Reasoning
COTS	Commercial Off-The-Shelf
FSM	Finite State Machines
IA	Intelligent Agents
IV&V	Independent Verification and Validation
KBS	Knowledge-Based System
KADS	KBS Analysis and Design Support
LOC	Lines Of Code
NIST	National Institute of Standards and Technology
SFMECA	Software Failure Mode, Effects, and Criticality Analysis
SPC	Statistical Process Control
SQA	Software Quality Assurance
SVVP	Software Verification and Validation Plan
SVVR	Software Verification and Validation Report
V&V	Verification and Validation

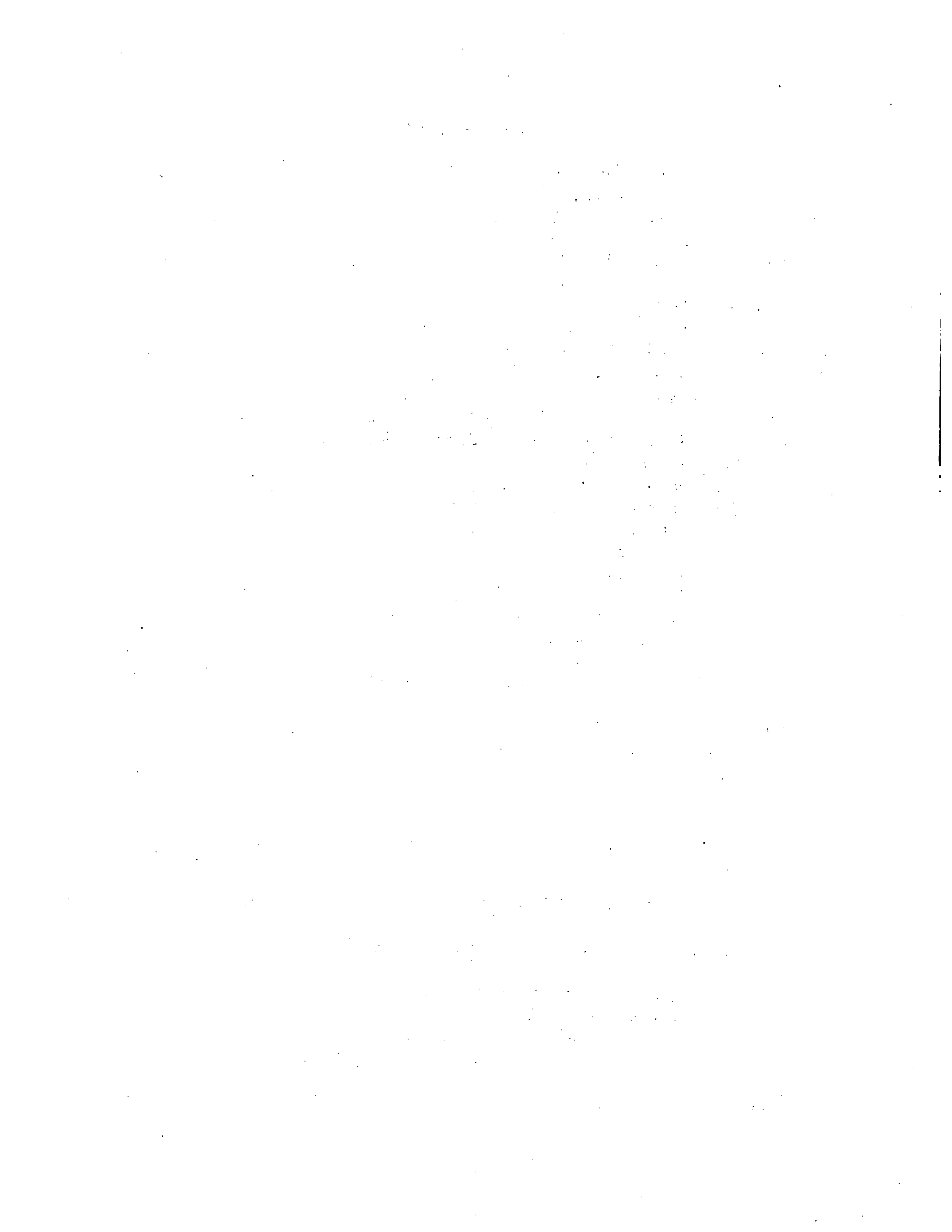


TABLE OF CONTENTS

Page

ABSTRACT . . . iii

ACKNOWLEDGMENTS . . . iii

EXECUTIVE SUMMARY . . . v

ACRONYMS . . . vii

1 INTRODUCTION . . . 1

2 SOFTWARE VERIFICATION AND VALIDATION (V&V) . . . 3
- 2.1 Independent V&V . . . 5
- 2.2 Software V&V Management . . . 6
- 2.3 Software V&V Activities . . . 9
  - 2.3.1 Software Requirements V&V . . . 10
  - 2.3.2 Software Design V&V . . . 13
  - 2.3.3 Code Verification . . . 15
  - 2.3.4 Unit Test . . . 16
  - 2.3.5 Software Integration Test . . . 18
  - 2.3.6 Software System Test . . . 19
  - 2.3.7 Software Installation Test . . . 21
  - 2.3.8 Software Operation and Maintenance V&V . . . 21

3 SOFTWARE V&V TECHNIQUES . . . 23
- 3.1 Strategies for Choosing Techniques . . . 23
- 3.2 Descriptions of Techniques . . . 25

4 REUSE . . . 39
- 4.1 Software Reuse Concerns . . . 39
- 4.2 Assessing Software for Reuse . . . 40

5 KNOWLEDGE-BASED SYSTEMS (KBS) . . . 45
- 5.1 KBS and Agents . . . 45
- 5.2 Differences and Similarities between KBSs and Other Systems . . . 46
- 5.3 KBS Development . . . 47
  - 5.3.1 KBS Analysis and Design Support . . . 47
  - 5.3.2 KBS Development Process . . . 48
- 5.4 Issues for Real Time KBS . . . 49
- 5.5 Reuse and KBS . . . 50

6 REFERENCES . . . 51

Preceding page blank

APPENDIX A. SOFTWARE V&V MEASUREMENT 59
A.1 Metrics 59
A.1.1 General Metrics 60
A.1.2 Software Requirements Metrics 61
A.1.3 Software Design Metrics 62
A.1.4 Code Metrics 64
A.1.5 Test Metrics 65
A.1.6 Software Installation Metrics 67
A.1.7 Software Operation and Maintenance Metrics 67
A.2 Statistical Process Control Techniques 68
A.2.1 Control Charts 69
A.2.2 Run Chart 71
A.2.3 Bar Graph 72
A.2.4 Pareto Diagram 72
A.2.5 Scatter Diagram 73
A.2.6. Method of Least Squares (Regression Technique) 75
A.3 Software Reliability Estimation Models 76

Tables

Table 2-1. Major Software V&V Activities 4
Table 3-1. Software V&V Techniques 35
Table 5-1. KADS Activities, Tasks, and Products 48
Table 5-2. Development Process Mapping 49
Table A-1. Types of Control Charts 70

Figures

Figure A-1 np Control Chart. 71
Figure A-2 Bar Chart. 73
Figure A-3 Pareto Chart. 74
Figure A-4 Scatter Diagram 75

1 INTRODUCTION

Like many other industries in the United States, the health care industry is turning to computing systems to control escalating costs and improve the quality of service. New technology is affecting the form and usage of patient information, diagnostic tools, and the tools which provide treatment. In particular, the application of information technology is a promising enabler for transferring gains in medical science research to patient benefit, for ensuring appropriate availability of patient information, and for managing the billing processes.

In response to the increasing dependence of the health care industry on information technology, the Advanced Technology Program (ATP) at the National Institute of Standards and Technology (NIST) issued Solicitation 94-04, Information Infrastructure for Health Care. The recognition by the ATP that the computer-based systems used in health care must be of high integrity[1] resulted in one element of that solicitation being technology for verification and validation (V&V). This report is the result of an effort funded by the ATP to produce guidance for the software V&V of computer-based health care systems. Software V&V helps to ensure and assess the quality of software-based systems.

Computing systems may be employed in the health care environment in efforts to increase reliability of care and reduce costs. To achieve these benefits, those functions controlled by software in health care systems must be secure, reliable, and maintainable. Software V&V will help to provide all these assurances. Software verification and validation (V&V) is an aid in determining that the software requirements are implemented correctly and completely and are traceable to system requirements. (Software V&V does not verify the correctness of the system requirements, only that the software requirements can be traced to the system requirements.) It uses a structured approach to analyze and test the software. It measures software against its requirements for quality attributes such as performance, safety[2], and computer security. Software V&V includes activities[3] to determine that the software system performs its intended functions correctly, to ensure that it performs no unintended functions, and to provide information about its quality and reliability.

The guidelines in this report address V&V issues related to the recognition that different health care systems may:

- execute in real time (e.g., implantable medical devices and treatment devices);
- rely heavily on existing software;
- contain many units contributing to overall system complexity;
- incorporate knowledge-based systems (KBS) (e.g., diagnostic systems).

---

[1]High integrity systems are those which can and must be trusted to work dependably [NIST190].

[2]Throughout this document, the word *safety* is used in the context of this definition from [LEVESON95]: Software system safety implies that the software will execute within a system context without contributing to hazards.

[3]This document adopts the terminology used in [ISO12207]; i.e., a "process" is made up of "activities" which contain "tasks." For example, the software development process includes a software requirements activity and the software V&V process includes, among others, the software requirements V&V activity which contains, among others, a task called software requirements evaluation. ([NIST223] only used the terms "process" and "activity," e.g., software requirements V&V process contains the activity called software requirements evaluation.)

Software V&V has long been employed on new development projects. Today, more and more systems are built using commercial off-the-shelf (COTS) software products, software components from sources external to the developer, and software from a previous version of a similar product built by the same organization. Some of the issues concerning software V&V for systems reusing any of these software types are addressed in this document. This particular aspect of software V&V for reused software requires additional research from the reuse and V&V communities.

The health care industry has been interested in and made use of artificial intelligence (AI) techniques by developing KBSs to understand the complex medical and patient data used for diagnosis. This interest has grown as the scale of the problem of managing data and knowledge has grown in the health care industry. While there are techniques available for V&V of the KBS which employ AI techniques, the V&V and AI communities still need to do more research especially in the areas of making knowledge maintenance easier and more reliable. These guidelines provide an overview of the issues in using KBS techniques on systems requiring high reliability and on some of the techniques for V&V of KBS especially KBS which employ expert systems.

The guidance in this report is generally applicable to most software systems and is compatible with the following existing NIST guidance documents:

- "A Study on Hazard Analysis in High Integrity Software Standards and Guidelines" [NIST5589]
- "A Framework for the Development and Assurance of High Integrity Software" [NIST223]
- "Quality Characteristics and Metrics for Reusable Software" [NIST5459]
- "Software Error Analysis" [NIST209]
- "Software Quality Assurance: Documentation and Reviews" [NIST4909]
- "Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards" [NIST165]
- "Guideline for Software Verification and Validation Plans" [FIPS132]

The overview of software V&V in section 2 of this report describes considerations for determining who performs software V&V and provides details on the management of software V&V. Section 2 also discusses the scope, objectives, and tasks of software V&V. Section 3 explains the categories of techniques supporting V&V. It also presents short descriptions of the more common techniques, the problems they help to uncover, and the other tasks they may support. Sections 4 and 5 address issues regarding reused software and KBS. In both cases, more research is needed to provide a comprehensive approach for software V&V. Appendix A addresses software metrics, statistical processes, and reliability estimation models that may be applied to the collective findings of software V&V.

2 SOFTWARE VERIFICATION AND VALIDATION (V&V)

Software verification and validation (V&V) is an aid in determining that the software requirements are implemented correctly and completely and are traceable to system requirements. (Software V&V does not verify the correctness of the system requirements, only that the software requirements can be traced to the system requirements.) The major objective of the software V&V process is to comprehensively analyze and test the software during development to determine that the software performs its intended functions correctly, ensure that it performs no unintended functions, and provide information about its quality and reliability [NIST165]. Software V&V evaluates how well the software is meeting its technical requirements and its safety, security, and reliability objectives relative to the system. It also ensures that software requirements are not in conflict with any standards or requirements applicable to other system components. Software V&V tasks analyze, review, demonstrate or test all software development outputs.

Software verification examines the products of each development activity (or increment of the activity) to determine if the software development outputs meet the requirements established at the beginning of the activity. The scope of each software development activity is defined by software program management. A software design may consist of many small increments for each iteration of the total system. Hence, V&V tasks can be performed on small outputs. Validation that the software is a correct implementation of the system requirements for which the software is responsible is conducted concurrently with, and at the end of, *all* software development activities.

The software V&V process produces a software verification and validation plan (SVVP), individual plans and reports for tasks, summary reports, anomaly reports, and a final software verification and validation report (SVVR). Software V&V planning is conducted against system requirements at the highest level of planning, and then on the software requirements, which should be traceable to the system requirements. Many software V&V tasks, such as planning for software system test, are actually performed in early development activities. The software system test plan is developed concurrently with the software requirements activity. The plan is updated with additions or changes in details as the project progresses. While different management and technical staff may be responsible for different types of test, staff who perform verification of the software requirements may be staff who prepare preliminary plans for software system tests. The development of the test plans and designs may lead to discovery of software requirements errors because of the analysis needed to plan tests.

One issue that often arises in planning a project and its software V&V effort is how to ensure the objectivity of the staff performing software V&V tasks. Independent V&V (IV&V) for software grew out of this concern. Software IV&V is the performance of software V&V tasks by a team that is separate from the software development group. IV&V is described in section 2.1.

This guideline is intended for use with any software development methodology. The software V&V process comprises the software V&V management activity and software V&V technical activities. Each activity consists of several tasks, shown in Table 2-1. These tasks are defined in [FIPS132] and expanded in [WALLACE94]. Software V&V management is described in section 2.2. It ensures that task selection is appropriate for achieving the software V&V objectives; ensures the

Table 2-1. Major Software V&V Activities

ACTIVITY	TASKS
Software V&V Management	-Planning -Monitoring -Evaluating results, impact of change -Reporting
Software Requirements V&V	-Review of concept documentation (if not performed prior to software requirements development) -Traceability Analysis -Software Requirements Evaluation -Interface Analysis -Initial Planning for Software System Test -Reporting
Software Design V&V	-Traceability Analysis - Software Design Evaluation -Interface Analysis -Initial Planning for Unit Test -Initial Planning for Software Integration Test -Reporting
Code V&V	-Traceability Analysis -Code Evaluation -Interface Analysis -Completion of Unit Test Preparation -Reporting
Unit Test	-Unit Test Execution -Reporting
Software Integration Test	-Completion of Software Integration Test Preparation -Execution of Software Integration Tests -Reporting
Software System Test[4]	-Completion of Software System Test Preparation -Execution of Software System Tests -Reporting
Software Installation Test	-Installation Configuration Audit -Reporting
Software Operation and Maintenance V&V	-Impact-of-Change Analysis -Repeat Management V&V -Repeat Technical V&V Activities

[4]This document treats acceptance test as a function of the acquirer of the software system, while acknowledging that the acquirer may sometimes work with V&V staff from the software requirements V&V through software installation test to develop acceptance test. Tasks for acceptance test parallel those for software system test. Differences may exist in the specific objectives, which may influence test requirements.

performance and quality of the V&V effort; selects appropriate metrics and techniques applied to the V&V results; and, conveys results of the V&V "tasks" to appropriate places.

The software V&V technical activities each have several tasks. Each task is accomplished by applying one or more techniques. A specific technique, such as control flow analysis, focuses on finding a specific type of problem, for example, a logic error. An aggregate of techniques is usually necessary to achieve the objectives of a task. Section 2.3 discusses the tasks for each activity (sec. 3 describes techniques and the problem areas related to those techniques).

2.1 Independent V&V

Some software V&V activities may be performed by two different groups. The use of a different organization (other than the software development group) for software V&V is called independent verification and validation (IV&V). The following is summarized from the chapter on IV&V in [WILEY].

Technical independence requires that members of the IV&V team (organization or group) may not be personnel involved in the development of the software. This team must have some knowledge about the system design or have related experience and engineering background enabling them to understand the system. The IV&V team must not be influenced by the development team when the IV&V team is learning about the system requirements, proposed solutions for building the system, and problems encountered. Technical independence is crucial in the team's ability to detect the subtle software requirements, software design, and coding errors that escape detection by development testing and SQA reviews.

The technical IV&V team may need to share tools from the computer support environment (e.g., compilers, assemblers, utilities) but should execute qualification tests on these tools to ensure that the common tools themselves do not mask errors in the software being analyzed and tested. The IV&V team uses or develops its own set of test and analysis tools separate from the developer's tools whenever possible.

Managerial independence means the responsibility for IV&V belongs to an organization outside the contractor and program organizations that develop the software. While assurance objectives may be decided by regulations and project requirements, the IV&V team independently decides the areas of the software/system to analyze and test, techniques to conduct the IV&V, schedule of tasks (within the framework of the system schedules), and technical issues to act upon. The IV&V team provides its findings in a timely fashion simultaneously to both the development team and the systems management who acts upon the reported discrepancy and findings.

Financial independence means that control of the IV&V budget is retained in an organization outside the contractor and program organization that develop the software. This independence protects against diversion of funds or adverse financial pressures or influences that may cause delay or stopping of IV&V analysis and test tasks and timely reporting of results.

The extent that each of these parameters is vested in the IV&V team's responsibilities defines the degree of independence achieved. Based on the definitions of IV&V and how much IV&V a specific project requires, some software V&V activities may be conducted by both the developer and another organization. For example, unit test by one organization may focus on demonstrating that specific objectives (e.g., safety objectives relative to the system), which may differ from the objectives of the developer (e.g., logic structure, test coverage), have been met [IEEEP1059].

2.2 Software V&V Management

The process of software V&V needs to be managed and performed comprehensively over the entire software development process. Management tasks, spanning all of the software development activities, are to:

- Plan and maintain the software V&V process.
- Coordinate and interpret performance and quality of the software V&V effort.
- Report discrepancies promptly to the user or development group.
- Identify early problem trends and focus software V&V tasks on them.
- Provide a technical evaluation of the software performance and quality attributes at each major software program review (so a determination can be made of whether the software product has satisfied its set of software requirements well enough to proceed to the next activity).
- Assess the full impact of proposed software changes.

An SVVP contains the information necessary to manage and perform software V&V. Major steps in developing an SVVP are to:

- Define (or confirm, if already provided) the quality and performance objectives (e.g., verify conformance to specifications, verify compliance with safety and computer security objectives relative to the system, assess efficiency and quality of software, and assess performance across the full operating environment).
- Characterize the types of problems anticipated in the system and define how they would be manifested in the software.
- Select the software V&V analysis and testing techniques to effectively detect the system and software problems.
- Select the metrics and techniques applied to V&V results to measure and predict the quality of the software.

The SVVP may include details for acquiring tools and for training personnel. The SVVP is revised as knowledge accumulates about the characteristics of the system, the software, and the problem areas in the software and in software V&V activities.

The software V&V process could be tailored to specific applications; however, the risk of the software failing and the subsequent consequences must be considered when selecting software V&V activities.

One software V&V management task is to monitor the software V&V technical progress and quality of results. During each software V&V activity, planned software V&V tasks are reviewed and new ones are added to focus on the critical performance/quality functions of the software and its system. The monitoring task includes formal reviews of software V&V discrepancy reports and technical evaluations to provide a check of their correctness and accuracy. Internal monitoring of the quality and accuracy of software V&V results is essential because the development group must make the necessary software changes as indicated by the software V&V results. If the software V&V results are erroneous, or of poor quality, the development group wastes its time and resources in attempting the changes, and more importantly, loses confidence in the effectiveness and helpfulness of the software V&V results. Software V&V studies [RADATZ] have shown that responding to discrepancy reports and software V&V evaluation reports consumes the largest portion of a development group's interface time with the software V&V group.

Boehm and Papaccio [BOEHM] report that the Pareto effect, that is, 20% of the problems cause 80% of the rework costs, applies to software. They recommend that software V&V "focus on identifying and eliminating the specific high-risk problems to be encountered by a software project." This does not mean that software V&V should examine only 20% of the software. Rather, software V&V needs to examine all the software. This includes: identifying potential hazards or threats to the safety and security of the system, prioritizing the software functions by criticality, and allocating software V&V analysis resources to those areas of the software which contain critical[5] functions and high-risk problems (i.e., more error-prone). Identifying and focusing on critical and high-risk areas of the software can be addressed by these software V&V methods:

- examination of early program deliveries to software V&V staff;
- use of software hazard (or threat) analysis; and
- conduct of a "criticality analysis" to identify the most critical functions of the software.

Various approaches in development can provide early product information to software V&V. These include: prototypes, incremental software development, and handing over each unit or subfunction following development unit testing. Incremental software development is an effective method of providing early product information to software V&V. The early deliveries reinforce the systematic analysis and test approach used by software V&V to examine the software in smaller pieces while progressively evaluating larger software pieces as each new piece is integrated. High-risk software areas are easier to identify by using the incremental approach because the software V&V can:

---

[5] A critical function is a function that must be performed, correctly and reliably; otherwise the system fails in a manner that may have serious consequences.

- Provide an early lead time to evaluate each engineering solution and allow time to suggest alternative solutions which can be incorporated in subsequent incremental deliveries without adversely impacting the schedule.

- Isolate each new set of requirements and evaluate their impact on the system performance.

- Provide early indications of system performance so that adjustments can be made to refine the desired performance.

- Develop trend information about software anomalies and risk issues to allow time to adjust the development and software V&V resources and planning to accommodate evolving software risk issues.

In incremental development, a software build (or early product) represents a basic program skeleton including draft documentation containing portions of the full software capabilities. Each successive build integrates additional functions into the skeleton. Based on discrepancy or progress reports from software V&V, software program management can make the technical and management decisions to refocus the software V&V and development team onto the program's specific problem areas of the software.

Two related analyses, criticality and hazard, can help focus the V&V effort on those parts of the program whose consequence of failure are most severe. A hazard is an (unsafe) "condition that may lead to an unintended event that causes an undesirable outcome" [WALLACE91]. For example, a driver of a car ignores warning lights at a railroad crossing and drives the car onto the tracks. The hazard is the presence of the car and train on the track at the same time. The unintended event (mishap) is the train colliding with the car. The undesirable outcome is the probable loss of life and damage to the car and train. The term "hazard" generally is used to refer to safety problems; the term "threat" generally is used to refer to security problems. In this document, the methods and issues related to hazard analysis are also applicable to security issues; the terms "threat" and "security" could be used in place of "hazard" and "safety" respectively.

Criticality analysis locates and reduces high-risk problems and is performed at the beginning of a project. It identifies the functions and units which are required to implement critical program functions or quality requirements (e.g., safety, computer security). The steps of the analysis are:

- Develop a block diagram or control-flow diagram of the system and its software. Each block or control-flow box represents a system or software function (unit).

- Trace each critical function or quality requirement through the block or control flow diagram.

- Classify all traced software functions (units) as critical to either the proper execution of critical software functions or the quality requirements.

- Focus additional analysis on these traced software functions (units).

- Repeat criticality analysis for each activity to observe whether the implementation details shift the criticality emphasis to other or additional functions (units).

System hazard analysis is used to identify potential events and circumstances that might lead to problems of varying degrees of severity, from critical failures resulting in loss of life or national security problems, to less serious malfunctions in the system. Software hazard (or threat) analysis focuses on the role of software relative to the hazards, or threats. Specific techniques that can be used for hazard analysis are included in section 6 with the V&V techniques; these include event tree analysis, software fault tree analysis, Petri-nets, and software failure mode, effects, and criticality analysis. (Hewlett-Packard's Medical Systems Unit has developed a software hazard avoidance process utilizing some aspects of these techniques [CONNOLLY].)

When identification of high risk areas from early deliveries, criticality analysis, and hazard (or threat) analysis are used together, the software V&V approach can focus on the most critical areas of the early software products. Software V&V results, obtained early enough in the software development process, can have significant impact on the quality and performance of the system under development.

2.3 Software V&V Activities

Software V&V should begin when the project begins. Usually the first software V&V tasks are conducted during the software requirements V&V activity. One V&V task is to examine the early project documentation, often called concept documents, to verify that the system to be built is not only feasible but will use the rules, conventions, algorithms, and practices appropriate to the application domain of the system. *Software requirements V&V* is performed to ensure that the specified software requirements are correct, complete, consistent, accurate, readable, and testable, and will satisfy the system requirements. Poorly specified software requirements (e.g., incorrect, incomplete, ambiguous, or not testable) contribute to software cost overruns and problems with reliability. Even when software fully meets its requirements upon delivery, there may be problems in the maintenance activity because general requirements (e.g., maintainability, quality, and reusability) were not accounted for during the original development. Identifying software requirements is difficult because the complexity of the problems being solved causes uncertainty in developing the intended system performance requirements. The occurrence of changes in requirements (e.g., to incorporate new technologies, new missions, new knowledge, new interfacing systems, new people coming on the scene) throughout the software development process adds significantly more chance for error. Software requirements V&V is intended to prevent these problems from occurring.

Design errors can be introduced by misrepresentation of the functional requirements and by implementation constraints relating to timing, data structures, memory space, and accuracy. *Software design V&V* provides assurance that software requirements are not misrepresented or incompletely implemented; that extraneous software requirements are not designed into the solution by oversight; that software requirements are not left out of the software design; and that other constraints are managed correctly.

Clerical and syntactical errors have been greatly reduced through use of structured programming, reuse of code, adoption of programming standards and style guides, availability of more robust computer languages, better compiler diagnostics and automated support, and, finally, more

knowledgeable programmers. Nevertheless, problems still occur in translating design into code and *code V&V* continues to be an important software V&V activity.

Test management is an important part of the software V&V activity in which all testing needed for the software system is considered and planned. Software V&V test planning begins with software requirements and spans almost the full range of activities. Test planning tasks encompass different types of testing--unit test, software integration test, and software system test. The planning activities result in documentation for each test type consisting of test plan, test design, test case, and test procedure documents.

Unit test verifies the design and implementation of software units. *Software integration test* verifies functional requirements as the software units are integrated together. Special attention is focused on software, hardware, and operator interfaces. *Software system test* validates the entire software program against system requirements and software performance objectives. Software system tests validate that the software executes correctly within its stated operating environment. The software's ability to deal properly with anomalies and stress conditions is emphasized. Software V&V tests are not intended to duplicate or replace the user and development group's test responsibilities, but instead test behavior not normally checked by the user or development group.

Software installation test validates that the software operates correctly with the operational hardware system and with other software, as specified in the interface specifications. It verifies that the installation procedures are correct and adequate, that the software is the same as the executable code delivered for installation, and that all supporting software products are the proper versions. Software installation test verifies that the software has been accurately tailored for site-dependent parameters and that the configuration of the delivered product is correct.

In *software operation and maintenance V&V*, when a software change is made, all software V&V activities are considered and possibly repeated to ensure that nothing is overlooked. Software V&V activities include examining the impact of the change throughout the system to understand what software V&V activities are needed. Software V&V activities are added or deleted to address the type of software change made. In many cases, an examination of the proposed software change shows that software V&V needs to repeat its activities on only a small portion of the software. Also, some software V&V activities, such as verifying the original concepts, require little or no effort to verify a small change. Small changes can have subtle but significant side-effects in a software program; for this reason, change analysis (a software operation and maintenance V&V task) is significant in preventing unintended functions and problems from making their way into new versions of the system.

2.3.1 Software Requirements V&V

The software requirements V&V activity checks that the allocation of system requirements to software is appropriate and correct, and how well the software requirements have been specified (e.g., correct, complete, nonambiguous, testable). It should be structured to ensure that the software objectives have been met. Verification of the software requirements should include an examination of documentation produced earlier in system development (e.g., initial feasibility studies, concepts on which the system has been designed) if this examination has not already been performed. If the

assumptions, algorithms, and physical rules imposed on the software requirements previously have not been verified to be appropriate for this project, then software V&V should perform those checks. Inputs to the software requirements V&V activity may be documents written in natural or formal mathematical languages and may include graphics and charts. When formal mathematical languages are used, other forms of representations may be provided to different users of the specifications. Software requirements verification must ensure fidelity among the forms of representation. [NIST223]

Concurrently with software requirements V&V, software system test planning in initiated. Software V&V examines all the proposed testing for the system to ensure that comprehensive testing and appropriate resources are planned. Each type of testing (unit, software integration, software system) is discussed more fully in this report. When the system requirements and the software requirements have been specified and reuse of software is identified, reuse issues identified in section 4 must be checked to ensure the software is suitable for the application domain and the operating environment.

The remainder of this section elaborates on software requirements V&V for general V&V tasks, for V&V tasks specifically designed for reused software, and those for knowledge-based systems (KBS).

<u>General</u>[6]

- Conduct a concept documentation evaluation.
  - Evaluate the defined concept to determine whether it satisfies user needs and project objectives in terms of system performance requirements, feasibility (e.g., compatibility of hardware capabilities), completeness, and accuracy.
  - Identify major constraints of interfacing systems and constraints/limitations of the proposed approach and assesses the allocation of system functions to hardware and software, where appropriate.
  - Assess the criticality of each software item defined in the concept.
- Begin test planning.
- Conduct a software traceability analysis - Trace software requirements to system requirements (and vice versa) and check the relationships for accuracy, completeness, consistency, and correctness; check that allocation is appropriate and complete.
- Conduct a software requirements evaluation.
  - Evaluate the software requirements for accuracy, completeness, consistency, correctness, testability, and understandability.

[6] V&V tasks related to testing are discussed in sections 2.3.4 through 2.3.6.

- Measure completeness by verifying existence and correctness of defining properties: initiator of action, action, object of action, conditions, constraints, source, destination, mechanism, and reason.

- Verify correctness and appropriateness of software requirements and assertions (executable statements that may be required in the software as fault tolerance protections for the system safety and computer security objectives (e.g., checking algorithms, states and integrity of system and the responses to unfavorable results of the assertions)). Verify the operation of the assertions will not adversely impact system performance.

- Verify correctness and appropriateness of fault tolerance requirements. Verify that the operation of the assertions will not adversely impact system performance.

▸ Assess how well the software requirements accomplish the system and software objectives.

▸ Identify critical areas of software by assessing criticality of software requirements.

▸ Evaluate software requirements for compliance to software requirements standards and software engineering practices.

- Conduct a software interface analysis - Evaluate software requirements with hardware, user, operator and software interface requirements for accuracy, completeness, consistency, correctness, and understandability.

<u>Reuse-Specific</u>

- Evaluate the reused software for conformance to its performance goals, to identify constraints of interfacing systems, to allocation functions to hardware and software, and to assess criticality of each software item.

- Conduct software interface analysis to evaluate reused software to new requirements for accuracy, completeness, consistency, correctness, and understandability, relative to the operating environment of both the reused and the new software and to the application domain. When COTS is considered for use in a new system, this task is especially significant for ensuring that the COTS will match the system interfaces in the operating environment.

- Compare the new software system objectives to the content of the reused documentation and the reused code to ensure the:

▸ availability of all necessary files;

▸ adequacy of user manual (compare to the requirements for the user manual in software development); and,

- compatibility of the software, hardware, and system environment (e.g., was the old system designed for a 16 bit machine and will now be on a 32 bit machine?).

- If the reused software is COTS, consider whether any functions of the software are to be blocked out from usage; if the consequences of any functions are unknown; and, the operational history of the COTS relative to failure.

KBS-Specific

- Verify the scope and complexity of the proposed domain for the KBS.

- Verify the correctness and appropriateness of the requirements on accuracy and completeness of the expected results (e.g., is the system supposed to perform like a student or an expert?).

- Verify that the selected tools can implement a domain model of the expected scope and complexity.

- Determine how accuracy of the system will be evaluated and against what standard it will be evaluated.

- Determine the volatility of the domain model and strategy for updating the knowledge base.

2.3.2 Software Design V&V

The software design V&V activity occurs after the software requirements have undergone the software V&V process and the software design, or an increment of the software design, has been completed.[7] The software V&V tasks of traceability, evaluation, and interface analysis provide assurance that software requirements are not misrepresented, incompletely implemented, or incorrectly implemented. By verifying that the software design meets its software requirements, the software design V&V activity also supports validation that the software design meets system requirements. There may be several instantiations of the software requirements and software design verification before the entire system is verified. [NIST223]

When the software system is designed, decisions may be made to incorporate existing software. Again, the issues identified in section 3 must be considered by software V&V to ensure that the reused software is appropriate, and that the software design takes into account any changes that must be made to the reused software to accommodate the operating environment and the application domain. The tasks and techniques are the same as for the software being developed, but the objectives and issues are specific for reuse.

The remainder of this section elaborates on software design V&V for general V&V tasks, for V&V tasks specifically designed for reused software, and those for KBSs.

---

[7]According to the model used for development, the software V&V process may be exercised on the entire software design or software design increments, but always traceable back to the software requirements.

General

- Conduct a software design traceability analysis - Trace software design to software requirements, and vice versa. Check the relationships for accuracy, completeness, consistency, and correctness.

- Conduct a software design evaluation.

  - Evaluate the software design for accuracy, completeness, consistency, correctness, and testability.

  - Evaluate software design for compliance with software design standards; language standards if appropriate; and software engineering practices.

  - Assess software design against assigned quality attributes.

- Conduct a software design interface analysis - Evaluate software design for accuracy, completeness, consistency, and correctness of hardware, operator and software interface requirements.

- Verify that the software requirements for assertions, responses to assertions and other required system algorithm and integrity checks or fault tolerance protections have been designed into the software. Check that the software design is complete and accurate and will not adversely affect system performance.

- Coordinate with software integration test planning.

Reuse-Specific

- Conduct an evaluation of the original software design documentation for compliance to software design requirements of the new system. Verify interface requirements. Generate any needed software design information or justify the use of the software without the required information. This determination should be based on recognized risk (safety, cost of modifications, impact of various degrees of uncertainty on the project) and coordinated with the user.

- If any modifications are needed, evaluate whether or not the software and documentation are adequate to support the modification (e.g., for change analysis, testability). If not, the needed information should be obtained or developed. If this is not prudent, modifications should not be made when they cannot be supported by adequate software design information.

KBS-Specific

- Verify that the domain model:
  - is complete and consistent; and,
  - represents the domain knowledge.

- Verify that the domain model addresses, at the required level of accuracy and completeness, the range of expected problems.

- Verify that the domain model operates in the specified scope.

2.3.3 Code Verification

The code verification activity verifies correct implementation of software design into code. Often this activity requires tedious checking of details within the code; automation provides protection against human error in gathering the code information for analysis and also can speed the process. Code verification is the last opportunity to find and remove errors that could cause unnecessary costs and delays from advancing poor code into any of the test activities. Code validation is accomplished through unit test which is described in section 2.3.4. [NIST223]

At this point in the software development process, the reuse issues should have been examined and the decision made to reuse or not to reuse the software. In the case that changes are to be made to the code, or if there is a possibility changes will be needed in a future version of the software system under development, some software V&V tasks may be needed.

The knowledge base should be internally consistent and reflect the domain model. In its simplest form, maintaining knowledge base consistency (or integrity) means not allowing a fact and its negation to both be part of the knowledge base. More extensive consistency checks can disallow rules that would, potentially, infer both a fact and its negation. Knowledge consistency is a key issue. A consistent domain model and a consistent representation of that model is critical. This is especially true for domains representing physical structures or controlled equipment. The model of the equipment and the physics controlling the behavior of the equipment must be consistent for computer controllers to function properly. In other domains, expert disagreement over the interpretation of a set of facts may be normal and expected. For example, legal disputes frequently involve the interpretation of the facts themselves. Probabilities can be one way to handle conflicting knowledge.

The remainder of this section elaborates on code verification for general tasks, for verification tasks specifically designed for reused software, and those for KBSs.

General

- Conduct a source code traceability analysis - Trace source code to software design, and vice versa. Check the relationships for accuracy, completeness, consistency, and correctness.

- Conduct a source code evaluation.
    - Evaluate the source code for accuracy, completeness, consistency, correctness, and testability.
    - Evaluate source code for compliance with code standards, language standards if appropriate, and software engineering practices.
    - Assess source code against assigned quality attributes.
- Conduct a source code interface analysis - Evaluate the source code for accuracy, completeness, consistency, and correctness with respect to the hardware, operator, and software interfaces.
- Evaluate draft code-related documents (e.g., user manual, commentary within the code) with source code for completeness, consistency, and correctness.
- Coordinate with unit test[8].

Reuse-Specific

- If the source code is available, compare it to known design specifications. Evaluate for correctness, consistency, completeness, and accuracy. Assess the interfaces for consistency with the system in which the reused code will be placed. Assess source code quality. (This task may be needed in instances where the history of the code is not well-known.)
- Evaluate source code for testability. Evaluate code-related documentation received from the source for suitability for any future modifications.

KBS-Specific

- Conduct a logical verification of the structure of the knowledge and rules in the knowledge base for consistency, completeness, etc.
- Verify that the knowledge base implements the domain model accurately.

2.3.4 Unit Test

Unit test is the test of the software elements at the lowest level of development. Units may be aggregates of software elements. Planning for unit test should occur concurrently with the software design activity. Reused software will probably not undergo unit test; unless changes were made to the units. Then, appropriate testing is performed as in regression testing.

[8]Unit test is actually a part of code V&V.

The remainder of this section elaborates on unit test for general V&V tasks, for V&V tasks specifically designed for reused software, and those for KBSs.

General

- Test planning - Establish the objectives of the unit test, the strategies to be employed, the coverage requirements, reporting and analysis, and close-out of anomalies.
- Generate, monitor, and update the unit test plan to accomplish objectives.
- Trace test design, cases, procedures, and execution results to the unit designs.
- Confirm that anomalies during test are software anomalies, and not problems detected for other reasons.
- Generate test cases and procedures - Develop test cases and procedures for unit test and continue tracing as required by software test plans.
- Perform unit test - Check individual software units for typographical, syntactic, and logic errors to ensure that each correctly implements the software design and satisfies the software requirements; execute the test cases; analyze results to verify anomalies; recommend changes to software design or code; and conduct retesting as necessary.
- Document test activities and results.

Reuse-Specific

- Evaluate existing test cases and reports for suitability for intended use.
- Prepare test cases and test procedures if any modifications are made to the reused software.
- Follow the criteria for unit test.

KBS-Specific

- Evaluate the knowledge and rules in the knowledge base against the domain knowledge.
- Establish objective for testing portions of domain knowledge.
- Plan tests for accuracy and completeness of domain model.
- Define test procedures to test for expected performance level of the system.

2.3.5 Software Integration Test

The software integration test activity is performed to examine how units interface and interact with each other with the assumption that the units and the objects (e.g., data) they manipulate have all passed unit tests [BEIZER]. Software integration tests check the interaction with other software (e.g., libraries) and hardware. The software integration test schedule depends upon the development and integration schedules for software units, hardware, and other components. For large systems, software integration test planning may require close coordination among all system personnel to ensure that the overall test objectives are achieved by the selected test strategy. For each major integration that has successfully undergone interface and interaction testing, functional tests may be developed and executed [BEIZER]. When all system components have been integrated and have successfully undergone software integration tests, then the system moves into software system test. During software integration test, reused software units are integrated into the system. It is critical to test that the interfaces are correct, and that the resulting software meets operating requirements.

The remainder of this section elaborates on software integration test for general V&V tasks, for V&V tasks specifically designed for reused software, and those for KBSs.

General

- Test planning - Establish the objectives of the software integration test, the strategies to be employed, the coverage requirements, reporting and analysis, and close-out of anomalies. Ensure that interface testing of reused software to other system software is planned.

- Generate, monitor, and update a software integration test plan to accomplish identified objectives.

- Trace test design, cases, procedures, and execution results to software requirements.

- Generate test cases and procedures - Develop test cases and procedures for unit test and continue tracing as required by software test plans.

- Perform software integration test.

    - Check the inter-unit communication links and test aggregate functions formed by groups of units.

    - Confirm that anomalies during test are software anomalies, and not problems detected for other reasons.

    - Ensure any changes to software requirements, software design, or code are made. Conduct retesting as necessary.

    - Conduct functional, structural, performance, statistical, and coverage testing of successfully integrated units after each iteration of software integration and successful testing of interfaces and interactions.

- Document test activities and results.

Reuse-Specific

- Perform software integration test in accordance with test procedures.
- Analyze results to determine if the software implements the intended use requirements and known design and that the software units function correctly together.
- Conduct interface tests of reused units with other system components.
- Conduct tests of reused units with other system components to validate performance requirements.
- Evaluate existing test cases and reports for suitability for intended use.
- Prepare test cases and test procedures if any modifications are made to the reused software.
- Follow the criteria for software integration test.

KBS-Specific

- Evaluate knowledge base for completeness and consistency.
- Verify that the knowledge base represents the full scope of the domain model.

2.3.6 Software System Test

Software system test, in the context of software V&V, involves the conduct of tests to execute the completely integrated system. Software system test is the validation that the software meets its requirements. Validation of the complete system may involve many tests involving all system components. The software system tests exercise those system functions that invoke software to determine whether the software behaves as intended relative to complete system performance. These tests must be conducted in such a manner as to stress the system based on software responses to system inputs (e.g., from sensors, operators, databases). Tests and data collected from the tests are designed to provide an operational profile of the system which support a statistical analysis of the system reliability [MUSA87, MUSA89, BUTLER]. This section of the report addresses only the tests that validate that the software implements the system requirements; other tests for other components and perspectives are necessary for complete system validation.

While software system tests are conducted after the system has been built, it is imperative that planning for these tests is conducted concurrently with the software requirements activity because:

- Analyzing the software requirements for test requirements may result in finding software requirements errors and/or discovery of untestable requirements.

- Establishing test facilities (e.g., model of operational environment) and Computer-Aided Software Engineering (CASE) tools (e.g., test case generators, test data base) may require as much time as development of the system.

For reused software, software system test is performed to assure that the software is correct, consistent with prior documentation, complete for use and/or modification, and accurate. At the system level, reused software should be considered part of the system. Tests are in accordance with test procedures. Results are documented and traced as required by the software system test plan.

The remainder of this section elaborates on software system test for general V&V tasks, for V&V tasks specifically designed for reused software, and those for KBSs.

General

- Test planning - Establish the objectives of the software system test, the strategies to be employed, the coverage requirements, reporting and analysis, and close-out of anomalies.

- Generate, monitor, and update a software system test plan to accomplish objectives.

- Trace system and software requirements to test software design, cases, procedures, and execution results.

- Generate test cases and procedures - Develop test cases and procedures for unit test and continue tracing as required by software system test plans.

- Test the operation of the software as an entity (sometimes a simulated environment may be used); confirm that anomalies during test are software anomalies, not problems detected for other reasons; ensure any changes to software (software requirements, software design, code, or test cases) have been made; and conduct retesting as necessary.

- Document test activities and results.

Reuse-Specific

- Evaluate existing test cases and reports for suitability for intended use.

- Prepare test cases and test procedures if any modifications have been made to the reused software.

- Follow the criteria for software system test within the boundaries of the known and documented software design.

KBS-Specific

- Define procedures for testing the system according to the expected knowledge of the end user.

2.3.7 Software Installation Test

The software installation test activity is the final step before launching full customer acceptance testing. The purpose of installation test is to demonstrate that the correct software has been delivered and that the software interfaces are correct relative to any interfaces at the installation site. Acceptance testing, which involves the user/customer, is outside the scope of this document.

The remainder of this section elaborates on installation test for general V&V tasks, for V&V tasks specifically designed for reused software, and those for KBSs.

General

- Conduct an installation configuration audit.
  - Determine that all software outputs needed to operate the system are present.
  - Check that the software installed in the system is the software that underwent software V&V.
- Develop and execute tests that will examine and stress site-unique parameters (e.g., printer interface, operating system interface, monitor interfaces).
- Generate applicable documentation.
- Generate an SVVR (or generate it at the end of the software V&V process).

Reuse-Specific

- Conduct an installation configuration audit to verify that any reused software that has not been modified is the current version.

KBS-Specific

- Ensure that data and updates to the knowledge base which are supplied from external sources are in an acceptable form.

2.3.8 Software Operation and Maintenance V&V

The software operation V&V activity requires periodic checks that the integrity of the system has been maintained, that any changes to the system which affect its operation have been documented, and operators have received training in new or changed procedures. The software maintenance V&V activity requires planning for software V&V based on the extent of the maintenance (e.g., adaptive, corrective, perfective [FIPS106]), and hence a revisit of all the software development activities to identify to what extent each software V&V activity must be performed.

If software V&V has not been performed during software development, then the V&V during software operations and maintenance must consider performing a selected set of tasks from the software V&V activities related to earlier development activities. Some activities may include generating software requirements or software design information from source code, an activity known as reverse engineering. While costly and time consuming, it is necessary when the need exists for a rigorous software V&V effort.

The remainder of this section elaborates on software operation and maintenance V&V for general V&V tasks, for V&V tasks specifically designed for reused software, and those for KBSs.

General

- Conduct an anomaly evaluation - Evaluate the severity of anomalies during software operation and their effect on the system.

- Conduct a proposed change assessment - Assess proposed changes to the software and their effect on the system to determine software V&V activities from earlier development to be repeated. Conduct them.

- Develop an SVVP.

Reuse-Specific

Follow the guidance for reuse in section 4.

KBS-Specific

- Plan for update of knowledge base including domain model.

- Determine mechanisms used for updating knowledge base.

3 SOFTWARE V&V TECHNIQUES

The conduct of software V&V tasks to fulfill the requirements of the V&V activities generally involves techniques selected from three major classes: static, dynamic, and formal analysis. Static analysis techniques are those which directly analyze the form and structure of a product without executing the product [FIPS101]. Reviews, inspections, audits and data flow analysis are examples of static analysis techniques. Static analysis techniques are traditionally applied to software requirements, software design and source code. They may also be applied to test documentation, especially test cases, to verify their traceability to the software requirements, their adequacy to fulfill test requirements, and their accuracy.

Dynamic analysis techniques involve execution, or simulation, of a development activity product to detect errors by analyzing the response of a product to sets of input data [FIPS101]. For these techniques, the output values, or ranges of values, must be known. Testing is the most frequent dynamic analysis technique. Prototyping, especially during the software requirements V&V activity, can be considered a dynamic analysis technique; in this case the exact output is not always known but enough knowledge exists to determine if the system response to the input stimuli meets system requirements.

Formal analysis is the use of rigorous mathematical techniques to analyze the algorithms of a solution [FIPS101]. Sometimes the software requirements may be written in a formal specification language (e.g., VDM, Z) which can be verified using a formal analysis technique like proof-of-correctness. The term *formal* often is used to mean a formalized process, that is, a process that is planned, managed, documented, and is repeatable. In this sense, all software V&V techniques are formal, but do not necessarily meet the definition of the mathematical techniques involving special notations and languages.

Table 3-1, at the end of this section, lists the software V&V techniques addressed in this report and indicates under which V&V activities these techniques can be applied. This report does not necessarily address all software V&V techniques.

3.1 Strategies for Choosing Techniques

Some software V&V techniques used during software requirements V&V tasks are control flow analysis, data flow analysis, algorithm analysis, and simulation. Control and data flow analysis are most applicable for real time and data driven systems. These flow analyses transform logic and data requirements text into graphic flows which are easier to analyze than the text. PERT, state transition, and transaction diagrams are examples of control flow diagrams. Algorithm analysis involves re-derivation of equations or evaluation of the suitability of specific numerical techniques. Simulation is used to evaluate the interactions of large, complex systems with many hardware, user, and other interfacing software units.

Some software V&V techniques used during software design V&V tasks include algorithm analysis, database analysis, sizing and timing analysis, and simulation. Algorithm analysis examines the correctness of the equations or numerical techniques as in the software requirements activity, but also examines truncation and round-off effects, numerical precision of word storage and variables (e.g.,

single- vs. extended-precision arithmetic), and data typing influences. Database analysis is particularly useful for programs that store program logic in data parameters. A logic analysis of these data values is required to determine the effect these parameters have on program control. Sizing and timing analysis is useful for real-time programs having response time requirements and constrained memory execution space requirements.

Some software V&V techniques used during code V&V tasks are control flow analysis, database analysis, regression analysis, and sizing and timing analysis. For large code developments, control flow diagrams showing the hierarchy of main routines and their subfunctions are useful in understanding the flow of program control. Database analysis is performed on programs with significant data storage to ensure common data and variable regions are used consistently between all call routines. Data integrity is enforced and no data or variable can be accidentally overwritten by overflowing data tables. Data typing and use are consistent throughout all program elements. Regression analysis is used to reevaluate software requirements and software design issues whenever any significant code change is made. This technique ensures project awareness of the original system requirements. Sizing and timing analysis is done during incremental code development and compared against predicted values. Significant deviations between actual and predicted values is a possible indication of problems or the need for additional examination.

Another area of concern to software V&V is the ability of compilers to generate object code that is functionally equivalent to the source code, that is, reliance on the correctness of the language compiler to make data dependent decisions about abstract programmer coded information. For critical applications, this problem is solved by validating the compiler or by validating that the object code produced by the compiler is functionally equivalent to the source.

Code reading is another technique that may be used for source code verification. An expert reads through another programmer's code to detect errors. In an experiment conducted at the National Aeronautics and Space Administration Goddard Space Flight Center, code reading was found to be more effective than either functional testing or structural testing [BASILI]. The reason was attributed to the expertise of the readers who, as they read the code, were simulating its execution and were able to detect many kinds of errors.

Other techniques commonly used are walkthroughs, inspections and reviews. These tasks occur in interactive meetings attended by a team which usually includes at least one member from the development group. Other members may belong to the development group or to other groups involved in software development. The duration of these meetings is usually no more than a few hours in which code is examined on a line-by-line basis. In these interactive sessions, it may be difficult to examine the code thoroughly for control logic, data flow, database errors, sizing, timing and other features which may require considerable manual or automated effort. Advance preparation for these activities may be necessary and includes code analysis techniques. The results of these techniques provide appropriate engineering information for discussion at meetings where code is evaluated. Regardless of who conducts or participates in walkthroughs and inspections, software V&V analyses may be used to support these meetings.

A comprehensive test management approach to testing recognizes the differences in strategies and in objectives for unit, software integration, and software system test. Unit test verifies the design

and implementation of software units. Software integration test verifies functional requirements as the software units are integrated. Special attention is focused on software, hardware, and operator interfaces. Software system test validates the entire software program against system requirements and software performance objectives. Software system tests validate that the software executes correctly within its stated operating environment. The software's ability to deal properly with anomalies and stress conditions is emphasized. These tests are not intended to duplicate or replace the user and development group's test responsibilities, but instead supplement the development testing to test behavior not normally tested by the user or development group.

Effective testing requires a comprehensive understanding of the system. Such understanding develops from systematically analyzing the software's concept, requirements, design, and code. By knowing internal software details, software V&V testing is effective at probing for errors and weaknesses that reveal hidden faults. This is considered structural, or white-box, testing. It often finds errors for which some functional, or black-box, test cases can produce the correct output despite internal errors.

Functional test cases execute part or all of the system to validate that the user requirement is satisfied; these test cases cannot always detect internal errors that will occur under special circumstances. Another software V&V test technique is to develop test cases that violate software requirements. This approach is effective at uncovering basic design assumption errors and unusual operational use errors. The process of planning functional test cases requires a thorough examination of the functional requirements. An analyst who carefully develops those test cases is likely to detect errors and omissions in the software requirements. In this sense test planning can be effective in detecting errors and can contribute to uncovering some errors before test execution.

The planning process for testing must take into account the specific objectives of the software V&V for the software and the impact of different test strategies in satisfying these objectives. Frequently, the most effective strategy may be to combine two or more strategies. More information and references on software testing may be found in [WILEY].

Criticality analysis may be used to identify software V&V techniques to address high-risk concerns. The selection of V&V techniques for use on each critical area of the program is a method of tailoring the intensity of the software V&V against the type of risk present in each area of the software. For example, software V&V would apply algorithm analysis to critical numerical software functions, and techniques such as sizing and timing analysis, data and control flow analysis and interface analysis to real-time executive functions.

3.2 Descriptions of Techniques

The following are summary descriptions of techniques taken from [BAHILL], [BEN], [EWICS3], [KIRANI], [NBS93], [NGUYEN], [NIST209], [NIST5589], [NUREG6316], [OKEEFE], [OLEARY], [TURING],[VOAS91,92,95], [WALLACE94], and [WILEY]. Issues (in italics at the end of each description) include the types of errors the technique may find, the tasks the technique supports, and other related techniques (to or from which supporting information is provided).

- Algorithm analysis examines the logic and accuracy of the software requirements by translating algorithms into some language or structured format. The analysis involves rederiving equations or evaluating the suitability of specific numerical techniques. It checks that algorithms are correct, appropriate, stable, and meet all accuracy, timing, and sizing requirements. Algorithm analysis examines the correctness of the equations and numerical techniques, truncation and rounding effects, numerical precision of word storage and variables (single vs. extended-precision arithmetic), and data typing influences. *Issues: accuracy; algorithm efficiency; correctness; consistency in computation; error propagation; numerical roundoff; numerical stability; space utilization evaluation; system performance prediction; timing.*

- Analytic modeling provides performance evaluation and capacity planning information on the software design. It represents the program logic and processing of some kind of model and analyzes it for sufficiency. *Issues: accuracy; algorithm efficiency; bottlenecks; error propagation; feasibility; modeling; numerical roundoff; numerical stability; processing efficiency; system performance prediction.*

- Back-to-back testing detects test failures by comparing the output of two or more programs implemented to the same specification. The same input data is applied to two or more program versions and their outputs are compared to detect anomalies. Any test data selection strategy can be used for this type of testing, although random testing is well suited to this approach. Also known as comparison testing. *Issues: anomalies or discrepancies between versions.*

- Boundary value analysis detects and removes errors occurring at parameter limits or boundaries. The input domain of the program is divided into a number of input classes. The tests should cover the boundaries and extremes of the classes. The tests check that the boundaries of the input domain of the specification coincide with those in the program. The value zero, whether used directly or indirectly, should be used with special attention (e.g., division by zero, null matrix, zero table entry). Usually, boundary values of the input produce boundary values for the output. Test cases should also be designed to force the output to its extreme values. If possible, a test case which causes output to exceed the specification boundary values should be specified. If output is a sequence of data, special attention should be given to the first and last elements and to lists containing zero, one, and two elements. *Issues: algorithm analysis; array size; inconsistencies between limits; specification error.*

- Code reading involves an expert reading through another programmer's code to detect errors. The individual is likely to perform a pseudo-execution (mentally) of the code to pick up errors before compilation. *Issues: correctness; misuse of variables; omitted functions; parameter checking; poor programming practices; redundancy.*

- Control flow analysis transforms text describing software requirements into graphic flows where they can be examined for correctness. It checks that the proposed control flow is free of problems (e.g., unreachable or incorrect software design). Control flow analysis is used to show the hierarchy of main routines and their subfunctions and checks that the proposed control flow is free of problems (e.g., unreachable or incorrect code elements). It detects

poor and potentially incorrect program structures. *Issues: assertion testing/violations; bottlenecks; boundary test cases; branch and path identification; branch testing; cell structure of units; correctness; software design evaluation; error propagation; expected vs. actual results; file sequence error; formal specification evaluation; global information flow and consistency; hierarchical interrelationship of units; inaccessible code; software integration tests; inter-unit structure; loop invariants; path testing; processing efficiency; retest after change; system performance prediction; test case preparation; unit tests.*

- Coverage analysis measures how much of the structure of a unit or system has been exercised by a given set of tests. System level coverage measures how many of the unit parts of the system have been called by a test set. Code coverage measures the percentage of statements, branches, or lines of code (LOC) exercised by a test set. *Issues: unit tests, software integration tests, software system tests.*

- Critical timing/flow analysis checks that the process and control timing requirements are satisfied by modeling those aspects of the software design. *Issues: modeling; synchronization; timing.*

- Database analysis ensures that the database structure and access methods are compatible with the logical design. It is performed on programs with significant data storage to ensure that common data and variable regions are used consistently between all calling routines; that data integrity is enforced and no data or variable can be accidentally overwritten by overflowing data tables; and that data typing and use are consistent throughout the program. *Issues: access protection; data characteristics and types; software design evaluation; file sequence error; global information flow; processing efficiency; space utilization evaluation; unit tests.*

- Data flow analysis is important for designing the high level (process) architecture of applications. It can check for variables that are read before they are written, written more than once without being read, and written but never read. *Issues: assertion testing/violations; bottlenecks; boundary test cases; branch and path identification; branch testing; cell structure of units; data characteristics; environment interaction; error propagation; evaluation of program paths; expected vs actual results; file sequence error; global information flow and consistency; hierarchical interrelationship of units; inter-unit structure; loop invariants; processing efficiency; retest after changes; software design evaluation; software integration tests; system performance prediction; test case preparation; uninitialized variables; unused variables; variable references.*

- Decision (truth) tables provide a clear and coherent analysis of complex logical combinations and relationships. This method uses two-dimensional tables to concisely describe logical relationships between boolean program variables. *Issues: logic errors.*

- Desk checking involves the examination of the software design or code by an individual, usually an expert other than the author, for obvious errors. It can include looking over the code for obvious defects, checking for correct procedure interfaces, reading the comments to develop a sense of what the code does and then comparing it to its external specifications, comparing comments to software design documentation, stepping through with input

conditions contrived to "exercise" all paths including those not directly related to the external specifications, and checking for compliance with programming standards and conventions. *Issues: anachronistic data; calls to subprograms that do not exist; data fields unconstrained by data boundaries; failure to implement the design; failure to save or restore registers; improper nesting of loops and branches; improper program linkages; improper sequencing of processes; incomplete predicates; incorrect access of array components; inefficient data transport; infinite loops; initialization faults; input-output faults; instruction modification; inverted predicates; mismatched parameter lists; missing labels or code; missing validity tests; misuse of variables; prodigal programming; unauthorized recursion; undeclared variables; unreachable code; unreferenced labels.*

- Error seeding determines whether a set of test cases is adequate by inserting ("seeding") known error types into the program and executing it with the test cases. If only some of the seeded errors are found, the test case set is not adequate. The ratio of found seeded errors to the total number of seeded errors is an estimation of the ratio of found real errors to total number of errors, or

$$\frac{NumberSeededErrorsFound}{TotalNumberSeededErrors} = \frac{NumberRealErrorsFound}{TotalNumberRealErrors}$$

One can solve for the total number of real errors, since the values of the other three are known. Then, one can estimate the number of errors remaining by subtracting the number of real errors found from the total number of real errors. The remaining test effort can then be estimated. If all the seeded errors are found, this indicates that either the test case set is adequate, or that the seeded errors were too easy to find. *Issues: test case adequacy.*

- Event tree analysis uses a bottom-up approach to model the effects of an event that may have serious repercussions. The initiating event is the root of the event tree. Two lines are drawn from the root, depicting the positive and negative consequences of the event. This is done for each subsequent consequence until all consequences are considered. *Issues: hazard analysis; safety; threat analysis; timing.*

- Finite state machines (FSM) check for incomplete and inconsistent software requirements by modeling the software in terms of its states, inputs and actions. A system in state S_1 receives an input I, then carries out action A, and moves to state S_2 is an example. FSMs can check that there is an action and new state for every input in every state, and that only one state change is defined for each state and input pair. *Issues: incomplete software requirements specification; inconsistent software requirements; modeling.*

- Functional testing executes part or all of the system to validate that the user requirement is satisfied. *Issues: boundary test cases; branch and path identification; branch testing; file sequence error; path testing; program execution characteristics; retest after change; statement coverage testing; system performance prediction; software system tests; test case preparation; test thoroughness; unit test; uninitialized variables; unused variables; variable references; variable snapshots/tracing.*

- Inspections are evaluation techniques whereby the software requirements, software design, or code are examined by a person or group other than the author to detect faults, violations of development standards, and other problems. An inspection begins with the distribution of the item to be inspected (e.g., a specification). Each participant is required to analyze the item on his own. During the inspection, which is a monitored meeting of all the participants, the item is jointly analyzed to find as many errors as possible. All errors found are recorded, but no attempt is made to correct the errors at that time. However, at some point in the future, it must be verified that the errors found have actually been corrected. *Issues: accuracy; checklists (software requirements, software design, code); effective forerunners to testing; formal specification evaluation; go-no-go decisions; information flow consistency; logic errors; loop invariants; manual simulation; retest after change; space utilization evaluation; technical reviews; status reviews; syntax errors; uninitialized variables; unused variables.*

- Interface analysis is a static analysis technique. It is used to demonstrate that the interfaces of subprograms do not contain any errors that lead to failures in a particular application of the software. Interface analysis is especially important if interfaces do not contain assertions that detect incorrect parameter values. It is also important after new configurations of pre-existing subprograms have been generated. The types of interfaces that are analyzed include external, internal, hardware/hardware, software/software, software/hardware, and software/database. *Issues: actual and formal parameters mismatch; inconsistencies between subroutine usage list and called subroutine; inconsistency of attributes of global variables; inconsistency between COTS parameter usage relative to other system parameters; incorrect assumptions about static and dynamic storage of values; incorrect functions used or incorrect subroutine called; input-output description errors.*

- Interface testing is a dynamic analysis technique. Similar to interface analysis, except test cases are built with data that tests all interfaces. Interface testing may include the following: testing all interface variables at their extreme positions; testing interface variables individually at their extreme values with other interface variables at normal values; testing all values of the domain of each interface variable with other interface variables at normal values; testing all values of all variables in combination (may be feasible only for small interfaces). *Issues: actual and formal parameters mismatch; inconsistencies between subroutine usage list and called subroutine; inconsistency of attributes of global variables; inconsistency between COTS parameter usage relative to other system parameters; inconsistent interface parameters; incorrect assumptions about static and dynamic storage of values; incorrect functions used or incorrect subroutine called; input-output description errors.*

- Mutation analysis determines the thoroughness with which a program has been tested, and in the process, detects errors. This procedure involves producing a large set of versions or "mutations" of the original program, each derived by altering a single element of the program (e.g., changing an operator, variable, or constant). Each mutant is then tested with a given collection of test data sets. Since each mutant is essentially different from the original, the testing should demonstrate that each is in fact different. If each of the outputs produced by the mutants differ from the output produced by the original program and from each other,

then the program is considered adequately tested and correct. *Issues: boundary test cases; branch and path identification; branch testing; retest after change; test case preparation.*

- Performance testing measures how well the software system executes according to its required response times, CPU usage, and other quantified features in operation. These measurements may be simple to make (e.g., measuring process time relative to volumes of input data) or more complicated (e.g., instrumenting the code to measure time per function execution). *Issues: memory allocation; synchronization; timing.*

- Petri-nets model systems to assure software design adequacy for catastrophic-failure and other safety problems. The system (including software systems) is modeled using conditions and events represented by state transition diagrams. Petri-nets consist of places (conditions--represented by circles), transitions (events--represented by bars), inputs (pre-conditions--represented by arrows originating from places and terminating at transitions), outputs (post-conditions--represented by arrows originating from transitions and terminating at places), and tokens (indication of true condition--represented by dots). Petri-nets can be "executed" to see how the software design will actually work under certain conditions. Specifically, Petri-nets can be used to determine all the states (including hazardous states) the system can reach, given an initial condition. *Issues: hazard analysis; modeling; safety; threat analysis; timing.*

- Proof of correctness (formal verification) involves the use of theoretical and mathematical models to prove the correctness of a program without executing it. Using this method, the program is represented by a theorem and is proved with first-order predicate calculus. *Issues: correctness; proof of critical sections.*

- Prototyping helps to examine the probable results of implementing software requirements. Examination of a prototype may help to identify incomplete or incorrect software requirements and may also reveal if any software requirements will not result in desired system behavior. It can be used as an aid in examining the software design architecture in general or a specific set of functions. For large complicated systems prototyping can prevent inappropriate software designs from resulting in costly, wasted implementations. *Issues: behavior; omitted functions (from software requirements); incomplete software requirements specification; user interface.*

- Regression analysis and testing is used to reevaluate software requirements and software design issues whenever any significant code change is made. It involves retesting to verify that the modified software still meets its specified requirements. This analysis ensures awareness of the original system requirements. It is performed when any changes to the product are made during installation to verify that the basic software requirements and software design assumptions affecting other areas of the program have not been violated. *Issues: software integration tests; retest after change; software system tests; unit tests.*

- Requirements parsing involves examination to ensure that each software requirement is defined unambiguously by a complete set of attributes (e.g., initiator of an action, source of the action, the action, the object of the action, constraints). *Issues: accuracy; assertion testing/violations; checklists; completeness; consistency; environment interaction;*

feasibility; formal specification evaluation; hierarchical interrelationship of units; information flow consistency; software integration tests; inter-unit structure; path testing; proof of correctness; software requirements evaluation; software requirements indexing; software requirements to design correlation; retest after change; standards check; statement coverage testing; software system tests; unit tests.

- Reviews are meetings at which the software requirements, software design, code, or other products are presented to the user, sponsor, or other interested parties for comment and approval, often as a prerequisite for concluding a given activity of the software development process. Reviews check the adequacy of the software requirements and software design according to a set of criteria and procedures. *Issues: effective forerunners to testing; logic errors; syntax errors.*

- Sensitivity analysis is a prediction of the probability that a software testing scheme will make programmer faults observable during testing. It allows different testing strategies to be ranked, compared, and evaluated. Sensitivity analysis is useful for assessing which regions of code are most likely to be affected during software maintenance (code modifications). It can be twisted into an assessment of how fault-tolerant a program is to software programmer faults (logic errors). *Issues: correctness; logic errors; reliability; test case adequacy.*

- Simulation is used to evaluate the interactions of large, complex systems with many hardware, user, and other interfacing software units. Simulation uses an executable model to examine the behavior of the software. Simulation is used to test operator procedures and to isolate installation problems. *Issues: assertion testing/violations; behavior; boundary test cases; branch and path identification; branch testing; environment interaction; execution monitoring, sampling, support; feasibility; file sequence error; inter-unit structure; path testing; program execution characteristics; retest after change; statement coverage testing; system performance prediction; software system tests; uninitialized variables; unused variables; variable references; variable snapshot/tracing.*

- Sizing and timing analysis is useful for determining that allocations for hardware and software are made appropriately for the software design architecture. It is performed during incremental code development by obtaining program sizing and execution timing values to determine if the program will satisfy processor size and performance requirements allocated to the software. Significant deviations between actual and predicted values is a possible indication of problems or the need for additional examination. *Issues: algorithm efficiency; bottlenecks; boundary test cases; branch and path identification; branch testing; software integration tests; processing efficiency; program execution characteristics; retest after change; space utilization evaluation; software system tests; timing; unit tests.*

- Slicing is a program decomposition technique used to trace an output variable back through the code to identify all code statements relevant to a computation in the program. This technique may be useful to demonstrate functional diversity. *Issues: allocation of V&V resources; common code; information flow consistency; program decomposition; variable references.*

- Software failure mode, effects and criticality analysis reveals weak or missing software requirements by using inductive reasoning to determine the effect on the system of a unit (includes software instructions) failing in a particular failure mode. A matrix is developed for each unit depicting the effect on the system of each unit's failure in each failure mode. Items in the matrix may include the failure mode and causes, effect on system, criticality, change/action required, and prevention and control safeguards. The criticality factor, that is, the seriousness of the effect of the failure, can be used in determining where to apply other analyses and testing resources. *Issues: hazard analysis; safety; incomplete software requirements specification; threat analysis.*

- Software fault tree analysis identifies and analyzes software safety requirements. It is used to determine possible causes of known hazards. Its purpose is to demonstrate that the software will not cause a system to reach an unsafe state, and to discover what environmental conditions would allow the system to reach an unsafe state. The analyst assumes that an already identified hazard has occurred and then works backward to discover the possible causes of the hazard. This is done by creating a fault tree, whose root is the hazard. The system fault tree is expanded until it contains at its lowest level basic events which cannot be further analyzed. *Issues: hazard analysis; safety; threat analysis.*

- Stress testing tests the response of the system to extreme conditions to identify vulnerable points within the software, and to show that the system can withstand normal workloads. *Issues: design errors; planning for defaults when system over-stressed.*

- Structural testing examines the logic of the units and may be used to support software requirements for test coverage, i.e., how much of the program has been executed. *Issues: bottlenecks; error propagation; evaluation of program paths; parameter checking; program execution characteristics; retest after change.*

- Symbolic execution shows the agreement between the source code and the software requirements specification. This is an evaluation technique in which program execution is simulated using symbols rather than actual values for input data, and program output is expressed as logical or mathematical expressions involving these symbols. *Issues: assertion testing/violations; program execution characteristics; proof of correctness; retest after change.*

- Test certification ensures that reported test results are the actual finding of the tests. Test related tools, media, and documentation are certified to ensure maintainability and repeatability of tests. This technique is also used to show that the delivered software product is identical to the software product that was subjected to V&V. It is used, particularly in critical software systems, to verify that the required tests have been executed and that the delivered software product is identical to the product subjected to software V&V. *Issues: incorrect product version shipped; incorrect test results; reports on test cases that were omitted.*

- Walkthroughs are similar to reviews, but less formal and much more detailed. A walkthrough is an evaluation technique in which a designer or programmer leads one or more other

members of the development team through a segment of software design or code, while the other members ask questions and make comments about technique, style, and identify possible errors, violations of development standards, and other problems. *Issues: checklists; error propagation; effective forerunners to testing; formal specification evaluation; go-no-go decisions; logic errors; manual simulation; parameter checking; retest after change; small, but difficult, or error-prone sections of design or code; status reviews; syntax errors; software system tests; technical reviews.*

Reuse-Specific

Most V&V techniques are applicable to reused software. Guidance in section 3 provides suggestions on issues to be considered for deciding to reuse the software; these issues may require application of V&V techniques. The two techniques identified in this section are crucial.

- Consistency analysis compares the requirements of any existing software with the new software requirements to ensure consistency. *Issues: consistency.*

- Interface analysis (see interface analysis and interface testing above) is especially important to exercise interfaces of reused software to other parts of the system as part of the planning for the reused software, to ensure correct adaption of the reused code to possible differences in the software architecture, operating environment, and application domain from its original usage.

KBS-Specific Techniques

- Alternative model compares the domain model implemented in the KBS to an alternate domain model for completeness and accuracy.

- Control groups can be used during testing to compare performance on executing a given task with or without the KBS available.

- Credibility analysis compares the results of the system to known expert's answers and reasoning to the same cases and judges the credibility of the system.

- Field testing is used only for low risk applications. It places the KBS in actual use and records the results of that use.

- Illegal attribute testing checks rules against constraints for illegal attribute values. This as an effective method for eliminating bugs during the implementation process of KBS development.

- Logical verification is the verification of the expert's knowledge for completeness, consistency, etc., as the domain model for the knowledge base system is being built.

- Meta models compare the knowledge and rules to domain meta models.

- Partition testing selects test cases using partitions of the input and output space as criteria and checks if the specification addresses those cases. This as an effective method for eliminating bugs during the requirements, design, and implementation processes of KBS development.

- Rule verification checks for completeness, subsumed/redundant rules, inconsistent rules, dead end rules, circular rules, unreachable conclusions, etc.

- Statistical validation examines how frequently a KBS uses rules or clusters of rules int he knowledge base. If there are expectations about the frequency of use expected for some rules then statics on rules use can be useful.

- Turing tests compare performance of the system against that of an expert in blind trials.

- Weight analysis compares the statistical information associated with a rule to statics known about the domain.

Table 3-1. Software V&V Techniques

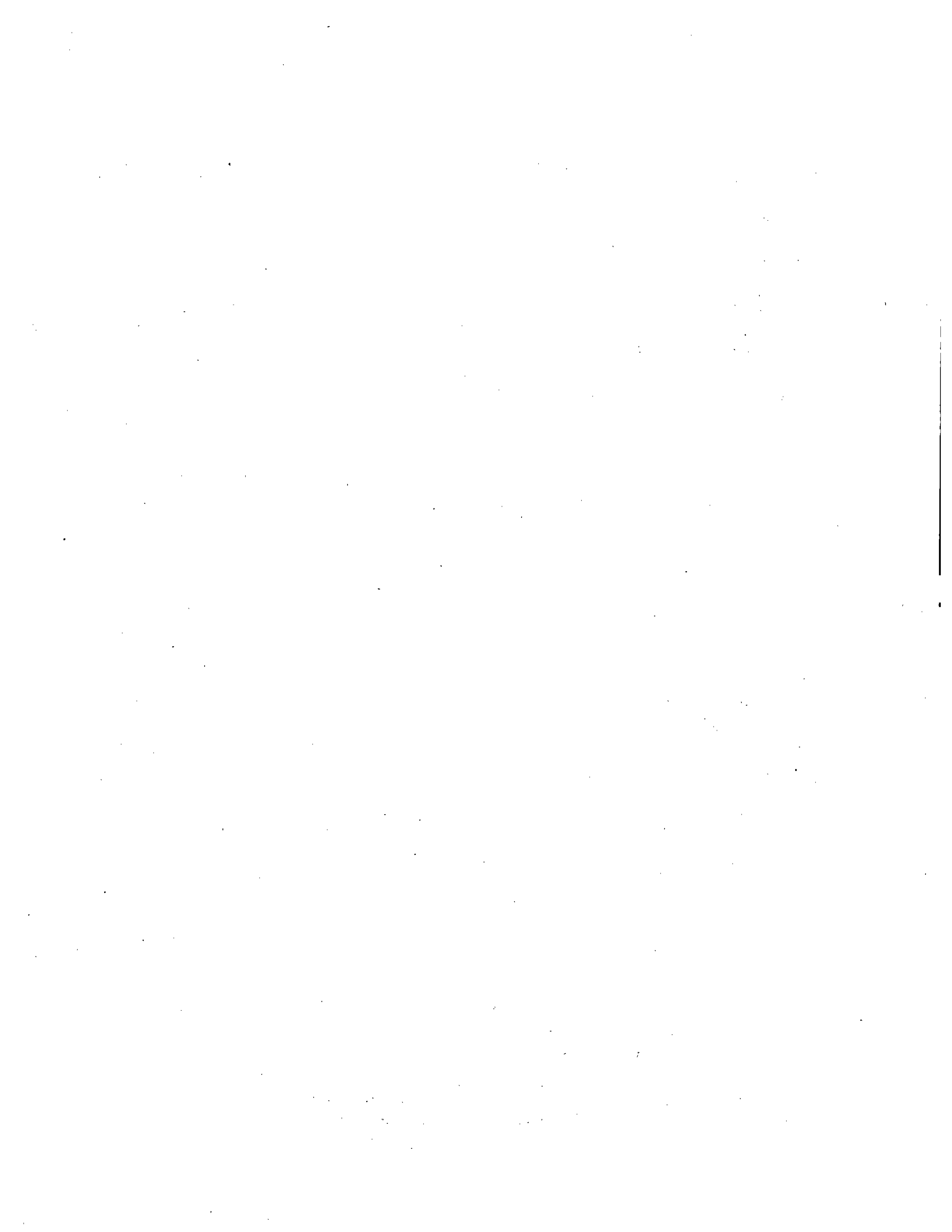
TECHNIQUE	REQ	DESIGN	CODE	UNIT	INTEGR	SYSTEM	INSTALL	OPER	MAINT
Algorithm Analysis	■	■	■	■					■
Analytic Modeling		■	■						
Back-to-Back Testing				■	■	■			
Boundary Value Analysis				■	■	■			■
Code Reading			■						
Control Flow Analysis	■	■	■						
Coverage Analysis				■	■	■			
Critical Timing/Flow Analysis		■							
Database Analysis	■	■	■						■
Data Flow Analysis	■	■	■						■
Decision (Truth) Tables	■		■						
Desk Checking		■	■						
Error Seeding				■	■	■			
Event Tree Analysis	■	■	■						
Finite State Machines	■								
Functional Testing				■	■	■			
Inspections	■	■	■						
Interface Analysis	■	■	■	■	■	■			
Interface Testing			■	■	■	■			
Mutation Analysis				■	■	■			
Performance Testing				■	■	■			
Petri-Nets		■	■						
Proof of Correctness			■						

TECHNIQUE	REQ	DESIGN	CODE	UNIT	INTEGR	SYSTEM	INSTALL	OPER	MAINT
Prototyping	■	■							
Regression Analysis and Testing	■	■	■	■	■	■	■		■
Requirements Parsing	■								
Reviews	■	■	■	■	■	■	■	■	■
Sensitivity Analysis				■	■				
Simulation	■	■	■	■			■		
Sizing and Timing Analysis		■	■	■	■				
Slicing			■						
SFMECA[9]	■	■	■						
Software Fault Tree Analysis	■	■	■						
Stress Testing				■	■	■			
Structural Testing				■					
Symbolic Execution			■						
Test Certification				■	■	■	■		
Walkthroughs	■	■	■	□	□	□[10]	■		
Reuse-Specific									
Consistency Analysis	■								
KBS-Specific									
Alternative Model	■							■	
Control Groups				■					

[9]Software Failure Mode, Effects and Criticality Analysis

[10]□ indicates that it is a walkthrough of the test code.

TECHNIQUE	REQ	DESIGN	CODE	UNIT	INTEGR	SYSTEM	INSTALL	OPER	MAINT
Credibility Analysis				■					
Field Testing								■	
Illegal Attribute Testing			■						
Logical Verification	■								
Meta Models				■					
Partition Testing	■	■	■						
Rule Verification	■								
Statistical Validation				■					
Turing Tests				■					
Weight Analysis				■					



4 REUSE

Computer systems have been used by the health care industry for a long time. As medical devices have increasingly become digital-based systems, software has taken on a larger role. With each upgrade, more software is developed, in addition to the software often reused from the previous device. Similarly, software in diagnostic systems and patient information systems undergoes upgrades or is used in new applications.

Most current literature on software reuse lacks discussion of the use of legacy software in new systems in terms of adapting the software to new considerations. Instead most of the literature discusses new techniques that can be used to make software reusable, or issues for building repositories of reusable software components [JOURNAL, TRACZ, FREEMAN, SSR, FRAKES]. Limited information exists to enable determinations about the "fit" of the reusable component with the new software, and with the relationship of software V&V activities to the reused software as it is integrated into the new system. Dr. Nancy Leveson has raised issues regarding the reuse of software in safety-critical systems [LEVESON95, LEVESON93]. Her work emphasizes the need to consider the entire system, including the software; for example, a fault tree must examine input of the software when performing a fault tree analysis. She also points out the need to understand differences between the operating environment of the original software and that of the new application.

This section on reuse provides general information about software reuse in high integrity systems and provides some suggested assessments to perform prior to accepting the software for reuse.

4.1 Software Reuse Concerns

The reuse of software in high integrity systems requires considerable planning, study, and application of software verification and validation (V&V). Failure to consider the operating environment, the actual machine, and the application of the system which will contain the reusable component can lead to problems. A prominent and tragic example of the seriousness of the need to exercise careful discipline in reusing software is the THERAC-25 radiation device, whose failures resulted in deaths of patients. While many factors contributed to the problems of the THERAC-25, inadequate consideration of V&V issues for the software reused from a previous version of the THERAC radiation device was one factor [LEVESON95].

The term "reusable software" may refer to any of the following types of software:

- software developed commercially, often referred to as COTS for commercial off-the-shelf software; this software may be a word processor, a spreadsheet, a data base program, or any commercial software that performs a recognized function

- software developed for inclusion in applications (e.g., units or segments of existing application software); examples include scientific routines and specialized functions

- software owned by an organization that has already been used in a software system developed by that organization; this software is often referred to as in-house software

Preceding page blank

- software developed by organization for a specific application that is considered for contract to another organization for an application it is developing

Some software considered for reuse may not have been subjected to comprehensive software V&V. Software acquired commercially or existing software that is used on a specific task may have been developed prior to a disciplined set of requirements being placed on the software development effort. Even when it has, it must be examined to ensure it fits the operating environment and application requirements of the new software system.

Acquired and existing software must be evaluated during the process of bringing it into the user organization since any software unit or program can be critical if it is part of a sequence of operations accomplishing a critical function or objective. The basis for the evaluation is the criteria used for development of the software as if that software had been developed using a disciplined approach. The software development process is modified to accommodate existing code. The perspective is from the software installation activity looking back to the beginning of software development.

Criteria from each activity of software development need to be considered as applicable to the software obtained. A specific set of criteria need to be identified and included as a specification within the procurement documentation or requested prior to acquisition from the source. This specification is intended to meet the criteria needed to support new software development to the extent possible. Acceptance verification needs to be performed upon receipt of the software and products requested to determine the foundation of further evaluation of the software. Certain constraints are recognized on availability of some documentation. For example, it is important to trace back to the assumptions and constraints of the original software; if the information doesn't exist, then the code must be examined to identify them.

4.2 Assessing Software for Reuse

The reuse of existing software in a new application requires ascertaining whether the quality of the software is sufficient for the intended application and whether the software can be integrated into the overall system in a way that system quality requirements, such as safety or security, are met. In assessing the suitability of reusable software, the organization must already know the level of integrity required for the new system; the importance of the following determinations declines as integrity requirements decline:

- records and documentation from the product development (including software V&V results)
- history of assessment of software development activities
- history of software V&V performed
- history of operational experience

Several issues should be addressed for all reusable software; the reusing organization should ask the following questions:

- Is the reusable software part of the function thread that directly or indirectly contributes to the accomplishment of a critical function or objective? The degree to which the reusable

software affects a critical function or objective can be determined by performing a criticality analysis (see sec. 3.1).

- Are the limitations and assumptions of the reusable software consistent with the limitations and assumptions of the new system?

- Is the available software documentation sufficient for the software V&V tasks to be performed on the reusable software as part of the system? Decide either to generate the necessary documentation (in whole or part) to support the V&V of reusable software or not to use the software in critical applications.

- Are there any unintended functions (functions built into the reusable item as part of its originally intended features, or as design assumptions that are not desired for the intended application) that will affect the performance and compatibility of the critical system functions of the operational profile to which the product was originally designed with the operational profile expected for the system in which the product will be incorporated?

- Is the configuration control process applicable for the new application?

- What are the mechanisms for error reporting, error correction mechanisms, and upgrade distribution methodology?

- To what extent are the requirements and architecture of the proposed system limited by the reusable software characteristics? These may include items such as concurrency, space utilization, space reclamation [HOOPER].

When the answer to the first question in the above list of issues is affirmative, that is, reusable software is part of the function thread that contributes to accomplishment of a critical function, then the remaining issues in the list take on greater significance. For example, the original software may have been written to accommodate assumptions about the operating environment (e.g., flight paths above sea level instead of below sea level, data transfer rate of 1200 baud instead of 9600 baud, a 16 bit machine instead of 8 bit machine as in the THERAC-25 case). Leveson addresses some of these issues [LEVESON95]. Any misuse of system assumptions in the "new" application can cause serious problems during operation.

When the reusable software is part of the function thread of a critical function, documentation of at least the interfaces is mandatory. Without the documentation, interface analysis during the software V&V of the project cannot be properly accomplished because most software V&V tasks rely heavily on information about software requirements, software design, testing, and other project data found in documentation.

When the reusable software was subjected to a prior software V&V effort, the prior software V&V results may serve as a basis for understanding the performance and limitations of the reusable software relative to the functions of the new system application. If no prior software V&V was performed on the reusable software, then software V&V should be conducted consistent with the software criticality level determined for the entire system application.

A comprehensive review of the reusable software should address issues concerning the reused software relative to the new system through:

- determination and identification of the functions to be performed;
- determination of the software integrity level;
- performance of a risk assessment to determine if the use of the previously developed or purchased software will result in undertaking an acceptable level of risk even if unforeseen hazards result in a failure;
- identification of all interfaces between the new software item and the previously developed or purchased software;
- identification of the capabilities and limitations of the previously developed or acquired software with respect to the project's requirements; and,
- following an approved test plan, testing of the high integrity features of the previously developed or purchased software with the project's software.

The comprehensive review should then assess the quality of the reused software through:

- determination of whether the software item has met quality assurance requirements consistent with the system quality requirements;
- determination of the conformance of the previously developed or acquired software to published specifications;
- assessment of the quality requirements applied to the software during its development;
- determination of the configuration controls applied to the software item;
- assessment of relevant operational experience and historical error reports with the software and maintenance actions applied to the software;
- identification of relevant documents and their status (e.g., product specifications, software design documents, usage documents) that are available to the obtaining organization; and,
- following an approved test plan, testing of the high integrity features of the previously developed or acquired software independent of the project's software.

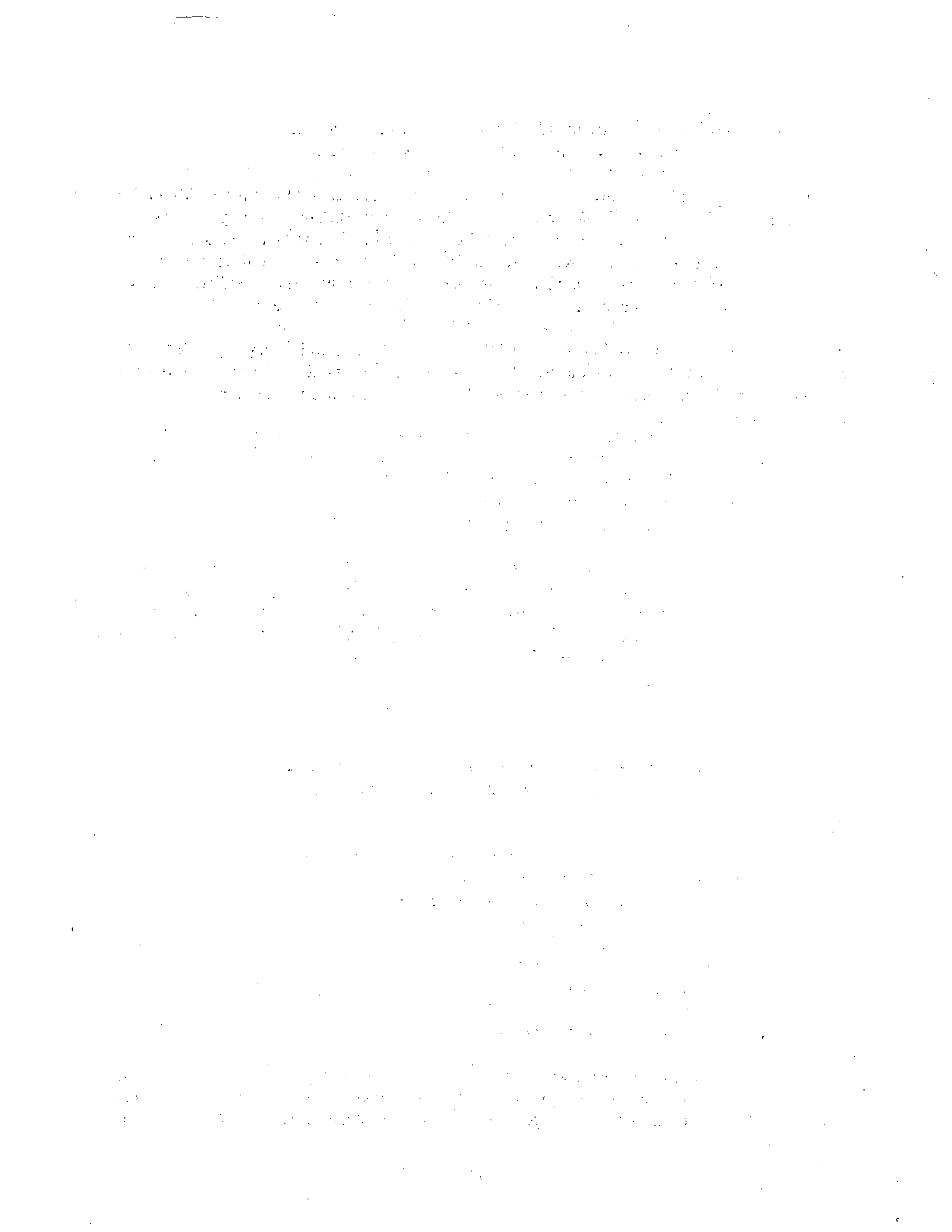
Software should not be reused in high integrity systems if the reused software:

- cannot be adequately tested;
- presents significant risk of hazardous failure;

- becomes unsafe or insecure in the context of its planned use; and/or,
- represents significant adverse consequence in the event of failure.

The inability to determine the level of risk present or the consequence of failure is justification for rejecting the use of the previously developed or acquired software. Equivalent analyses, tests, and demonstrations by the vendor of the adequacy of the vendor-supplied software for use in a high integrity application may be accepted as satisfying the intent of the assessment activities listed above. Previously developed or purchased software that is obtained as source code and modified for use by the project is subject to the same software V&V as are applied to new software.

The specific software V&V tasks performed, and the rigor and extent to which they are applied, will depend on the risks created through the use of the reusable software and the software integrity level of the intended system. Therefore, criticality analyses should be conducted on how the reusable software affects the system.



5 KNOWLEDGE-BASED SYSTEMS (KBS)

There has been a continuing interest from the health care community in the use of artificial intelligence (AI) techniques to manage the knowledge needed to encode expertise in the medical domain and make that expertise more broadly available. The AI community has also been interested in applying AI techniques to the complexity of medical domain (e.g., [KOHUT, ANDREASSEN].) The health care community has continuing problems of quickly and reliably getting expertise to those who need it and of updating that expertise in a rapidly changing field. Health care is a knowledge intensive field with rapidly expanding and evolving knowledge about diagnosis and treatment. The ability to quickly locate those information resources that have the highest probability of applying to a given problem is important to managing this health care information explosion. This knowledge management problem is often a problem of gaining access to needed expertise in a sub-field. One proposed approach is the use of knowledge-based systems (KBS).

The term "knowledge-based system" refers to systems which use or manipulate complex data or knowledge structures using AI techniques. The goal of these systems is to apply specialized expertise to solving problems. KBSs typically incorporate a domain model and apply that model to new problems. The purpose of incorporating a KBS into a larger system is to improve the performance of the overall system for unanticipated situations (e.g., its robustness).

KBS subsumes the older term "expert system" which typically refers to systems that encode an expert's knowledge as rules and apply those rules to solve problems. While many KBSs employ rules, the AI community has developed a variety of reasoning paradigms including case-based reasoning and the use of neural networks. Rules remain a popular, useful, well-understood approach to encoding an important subset of domain expertise. This subset is sometimes called a domain experts decision heuristics or "rules of thumb."

5.1 KBS and Agents

One type of KBS gaining attention is "agents" or "intelligent agents" (IAs). Agents are closely related to expert systems. There are at least two approaches to defining agents; [FONER] provides the following list of attributes of an agent:

- autonomous behavior (e.g., periodic action, spontaneous execution, initiative)
- personalizability to better execute the selected tasks
- discourse or two-way communication with the agent
- risk and trust associated with delegating tasks
- domain should have low criticality
- graceful degradation at the domain boundary
- cooperation between user and agent
- anthropomorphism
- meet expectations enough of the time to be useful

While [FONER] provides counter examples of agents which are missing some of these attributes, [FONER] stresses that agent technology is not well enough understood to be useful for critical domains. [FONER] focuses on the use of agents in game playing and other areas of social interaction.

Preceding page blank

[PAN] discusses the use of IAs as assistants for enterprise integration. For [PAN] "Each IA supports a clearly discernible task or job function, automating what it can and calling the services of other IAs, as well as human agents, when necessary." In this model, agents, both those associated with job functions and personal assistants associated with human agents, help integrate an enterprise through communication and information retrieval and synthesis. This model of IAs evolved from expert systems designed to better manage or integrate tasks on the factory floor. To cope with the changing demands on the factory floor, these expert systems had to be user extensible. In Pan and Tennenbaum's model, this user extensibility evolved into the cooperative interaction associated with IAs. These two points of departure for developing agents, factory floor controllers and social discourse, provide different criteria for evaluating agent usefulness and appropriateness for high risk tasks.

5.2 Differences and Similarities between KBSs and Other Systems

There are critical differences between KBSs and traditional systems cited in the literature affecting verification and validation (V&V) of KBSs:

- A KBS is both a piece of software and a domain model [OKEEFE].
- There may not be a unique correct answer to a problem given to a KBS [DAVENPORT].
- A KBS can adapt by modifying its behavior based on changes in its internal representation of the environment [HOLLNAGEL].

These differences provide the flexibility and special capabilities of a KBS, but these differences make use of traditional V&V methods for KBSs difficult and require the introduction of new techniques [OKEEFE, PREECE, DAVENPORT, NUREG6316].

The key component of a KBS that distinguishes it from other types of software is its encoding of the domain model in a knowledge base. Elicitation, formulation, and encoding of this model are major steps of KBS development. It is the knowledge base component that requires special emphasis during V&V. V&V of components other than the knowledge base (e.g., the inference engine, user interface) can rely on the same techniques as conventional systems [OKEEFE, NUREG6316]. With available expert system shells and tools, a new KBS may be able to reuse existing versions of these system components; however, reuse introduces new concerns for V&V as discussed in section 4 of this report.

V&V of the knowledge base requires understanding how the KBS will use the knowledge base and how the KBS itself will be used. All the uses of the knowledge base and the KBS containing it may not be known at development time. A KBS may operate in a domain with unclear boundaries, without complete information and with no unique correct answer to a given problem [DAVENPORT]. Complete enumeration of possible problems requiring the use of the model is unlikely. It is hard to predict what the range of problems are that any given model might apply to, or the enumeration of those problems might be prohibitively expensive. If the model is simple enough that experts could enumerate all of the possible problems that could be submitted to the KBS and all of the outcomes easily, a KBS would probably not be the best approach. One argument for using a KBS is that it can

improve the "robustness" of a system by supporting problem-solving under conditions that were not specifiable in advance. This can be through adaption of the knowledge base from automatic knowledge acquisition [HOLLNAGEL].

Another area important to KBS usage and related to validation of the model is "establishing credibility" [BAHILL] with KBS users. Even if the domain model is complete and accurate, a KBS user can loose confidence in the KBS if it appears to be using an obviously incorrect chain of reasoning. The user may see this when the KBS is attempting to validate a chain of reasoning by asking for additional information that the user feels is unnecessary or inappropriate. Under these conditions, the user can lose confidence in the KBS. Inappropriate questions can result from a failure to propagate knowledge generated from previous questions and asking redundant questions as the KBS tests new inference chains. Another source of inappropriate questions is incomplete specification of inference rules, resulting in the KBS asking questions that should be "obvious." The classic example for medical diagnosis systems, is the KBS asking if a male patient is pregnant. Obvious lapses of this type cause users to loose confidence in the system. [BAHILL]

5.3 KBS Development

While there is no single "standard life cycle" or development technique for KBSs, typical discussions of KBS development assume some form of rapid prototyping, evolutionary prototyping, or incremental development process. The assumptions underlying this choice are that experts and users cannot articulate expertise systematically and completely in one iteration and that extensive tool support supplies parts of the system other than the knowledge base contents.

There are many expert system shells and products supporting the rapid encoding of knowledge using particular styles of reasoning. While these shells and tools are helpful, they can also be deceptive since prototypes developed using these tools may not scale up or may require substantial further effort to become useable systems. Even with tools, the knowledge engineer must still work with the expert to formulate the expert's knowledge suitable for the expert system. There is no single set of rules for organizing this knowledge.

5.3.1 KBS Analysis and Design Support

KBS Analysis and Design Support (KADS) is one approach to structuring the analysis and design activities of KBS development. The goal of the KADS developers is to extend the ideas of structured programming to KBS development [TANSLEY]. KADS consists of the following:

- requirements analysis and design activities
- definitions of deliverables for those activities
- advice and guidance on the techniques used to perform those activities
- advice and guidance on tool and a library selection
- support for quality assessment and control
- identified opportunities for prototyping [TANSLEY]

KADS emphasizes the development of models through the requirements analysis and design activities. Table 5-1 lists the activities,[11] tasks, and products of the KADS development process.

Table 5-1. KADS Activities, Tasks, and Products

Activity	Task	Models/Products
Requirements Analysis	Process Analysis	Process Model
	Cooperation Analysis	Cooperation Model
	Expertise Analysis	Expertise Model
	Constraints Analysis	Constraints Document
	System Overview	System Overview Document
Design	Global Design	Global System Architecture
	KBS Design	KBS Functional Design KBS Behavioral Design KBS Physical Design

There is some evidence that a modeling approach to eliciting and recording expert knowledge makes the knowledge easier to represent and verify in the KBS. Domain experts develop models of how their domain works within the constraints they are accustomed to working with. For example, an auto mechanic has a model of how an engine works and a model of the physics employed in that engine, but not a general model of physics. Because domain experts find modeling approaches more natural, modeling may allow experts to formally verify the elicited model. [DAVENPORT]

5.3.2 KBS Development Process

The "Framework for the Development and Assurance of High Integrity Software" [NIST223] defines generic software development activities and the tasks related to software V&V that might be part of those activities. Typical KBS development is heavily tool supported using commercial tools and expert system shells and uses a spiral incremental development model. The inability to completely specify expertise in any given domain in one pass makes the waterfall model less useful; however, all development activities are still necessary. Table 5-2[12] maps a generic KBS development process to the framework's activities.

[11] This document adopts the terminology used in [ISO12207]; i.e., a "process" is made up of "activities" which contain "tasks." For example, the software development process includes a software requirements activity and the software V&V process includes, among others, the software requirements V&V activity which contains, among others, a task called software requirements evaluation. ([NIST223] only used the terms "process" and "activity," e.g., software requirements V&V process contains the activity called software requirements evaluation.) ISO12207 terminology is reflected in Table 5-1.

[12] ISO12207 terminology is reflected in Table 5-2 (see footnote 10).

Table 5-2. Development Process Mapping

KBS Generic Development Activity	Corresponding Conventional Activity
-Tool and shell selection -Inference engine selection/development	Early Software Requirements Activity
Iterate the following until KBS reaches acceptable state:	
Knowledge Acquisition Activity	Software Requirements Activity
Domain Modeling/Engineering Activity	Software Requirements/Software Design Activity
Design Activity	Software Design Activity
Knowledge Base Formulation Activity	Code Activity
Integration Activity	Software Integration Activity
[Reimplement in conventional language-especially common for real time systems]	[Software Requirements-Software Design-Coding Activity]
Integration Activity	Software Integration Activity
Installation Activity	Software Installation Activity
Knowledge Maintenance Activity	Software Operation and Maintenance Activity

5.4 Issues for Real Time KBS

Real time constraints introduce another concern for KBS V&V. There has been some interest in using KBSs for extracting information from sensors and other data sources and using that data for solving control and scheduling problems in real time [DAZ]. While there has been some work in extracting information from large data sources, there has been very little work on the V&V of performance constraints associated with real time systems [DAZ]. V&V of performance constraints requires predicting the performance of various KBS reasoning techniques. The ability to make those predictions is at best very difficult. [DAZ] summarizes the problem as "the less procedural the process path, as in typical AI approaches, the more difficult it becomes to predetermine the flow of control and to guarantee a response time." [DAZ]

Another problem for KBS performance is the apparent paradox that the more information available, the longer it can take to obtain a result from the system [DAVENPORT]. Each addition of a special case to a rule, or addition of a fact to a knowledge base, is one more item to be checked. For example, a diagnostic rule that specifies "if a light is not working, replace the light bulb" takes less time to test than one that specifies "if a light is not working and the power is on, replace the light bulb." Of course, additional clauses could be added (e.g., the light switch is on, the fixture is working) to this simple example before deciding that the probability of the light being burned out justified the cost of replacing the bulb. For a real-world, medical diagnosis example, there could be a much larger number of conditions to check before reaching a similar point. Unless the encoded domain model propagates knowledge as it applies inference rules, the KBS may repeatedly ask the user for the same information. This repetition may seriously impede performance; however, the time required to perform the extra inferences needed to propagate the information can also impede performance. [DAVENPORT]

5.5 Reuse and KBS

KBSs often make extensive reuse of components outside the knowledge base. Small to medium size KBSs are frequently built using expert system shells which supply the inference engine and user interface. While these shells permit rapid prototyping, the resulting system may not be appropriate for general use.

In selecting an expert system or in transferring a knowledge base from one KBS to another, it is important to consider the assumptions embedded in the inference engine that affect the construction and execution of the knowledge base. Some examples of assumptions that would affect KBS performance include the following:

- the execution order for the rules
- the ability to attach statistical information
- the representation and capability to manipulate statistical information
- the continued search for acceptable solutions after finding an initial solution

Changes in these assumptions could require reencoding the rules of other parts of the domain model for the new inference engine.

Assumptions about the inference engine are one consideration when reusing the domain model. There are other issues if the developer intends to combine the domain model with other domain models. Domain models incorporate many assumptions about the surrounding context for using the model and the "borders" of the model. It is important to consider these context assumptions when reusing a domain model. The domain model may also incorporate assumptions about capabilities of the inference engine used to execute the encoding of the model.

6 REFERENCES

[AIAA]

R-013-1992, "Recommended Practice for Software Reliability," American Institute of Aeronautics and Astronautics, Space-Based Observation Systems Committee on Standards, Software Reliability Working Group, 1992.

[AIRFORCE]

AFSCP 800-14, Air Force Systems Command, Software Quality Indicators, "Management Quality Insight," U.S. Department of the Air Force, January 20, 1987.

[ALBRECHT]

Albrecht, Allan J. and John E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, November 1983.

[ANDREASSEN]

Andreassen, S., R. Engelbrecht, and J. Wyatt, Artificial Intelligence in Medicine, IOS Press, Washington, 1993.

[BAHILL]

Bahill, A. Terry, Verifying and Validating Personal Computer-Based Expert Systems. Prentice Hall, New Jersey, 1991.

[BASILI]

Basili, V.R., and R.W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, Vol. 13, No. 12, December 1987.

[BEIZER]

Beizer, Boris, Software Testing Techniques, Van Nostrand Reinhold, New York, 1990.

[BEN]

Ben-Ahmeida, M., L.J. Kohout, and W. Bandler, "The Use of Fuzzy Relational Products in Comparison and Verification of Correctness of Knowledge Structures," Knowledge-Based Systems for Multiple Environments, ed. by L.J. Kohout, J. Anderson, and W. Bandler, Ashgate, 1992.

[BOEHM]

Boehm, B.W., and P.N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, The Institute for Electrical and Electronics Engineers, Inc., New York, NY, October 1988.

[BROCKLEHURST]

Brocklehurst, S., P. Y. Chan, Bev Littlewood, and John Snell, "Recalibrating Software Reliability Models," *IEEE Transactions on Software Engineering*, Vol. 16, No. 4, 1990.

[BUTLER]
Butler, R. and G. Finelli, "The Infeasibility of Experimental Quantified Life-Critical Software Reliability," *Proceedings of SIGSOFT'91: Software for Critical Systems*, Association for Computing Machinery, December 1991.

[CAPRIO]
Caprio, William H., "The Tools for Quality," *Total Quality Management Conference,* Ft. Belvoir, Virginia, July 13-15, 1992.

[CONNOLLY]
Connolly, Brian, "A Process for Preventing Software Hazards," *Hewlett-Packard Journal*, June 1993.

[DACS]
"Software Reliability Models," *DACS Newsletter*, Data & Analysis Center for Software, Volume X, Number 2, Summer, 1992.

[DAVENPORT]
Davenport, David, "Intelligent Systems: The Weakest Link?" *Intelligent Systems: Safety, Reliability and Maintainability Issues,* ed. by Okyay Kaynak, Ger Honderd and Edward Grant, New York: Springer-Verlag, 1992.

[DAZ]
D'az-Herrara, Jorge L, "Implications of Artificial Intelligence for Embedded Systems," *Proceedings of the Software Technology Conference,* Software Technology Support Center, 1995.

[DEMMY]
Demmy, W. Steven and Arthur B. Petrini, "Statistical Process Control in Software Quality Assurance," *Proceedings of the 1989 National Aerospace and Electronics Conference, NAECON, May 22-26, 1989, Dayton, OH,* IEEE, Inc.

[DUNN]
Dunn, Robert, Software Defect Removal, McGraw-Hill, Inc., 1984.

[EWICS3]
Bishop, P. G. (ed.), Dependability of Critical Computer Systems 3 - Techniques Directory, The European Workshop on Industrial Computer Systems Technical Committee 7 (EWICS TC7), Elsevier Science Publishers Ltd, 1990.

[FIPS101]
FIPS PUB 101, "Guideline for Lifecycle Validation, Verification, and Testing of Computer Software," U.S. Department of Commerce/National Bureau of Standards (U.S.), 1983 June 6.

[FIPS106]
FIPS PUB 106, "Guideline on Software Maintenance," U.S. Department of Commerce/National Bureau of Standards (U.S.), 1984 June 15.

[FIPS132]
FIPS PUB 132, "Guideline for Software Verification and Validation Plans," U.S. Department of Commerce/National Bureau of Standards (U.S.), 1987 November 19.

[FONER]
Foner, Leonard N., "What's An Agent, Anyway? A Sociological Case Study," Agents Group, MIT Media Lab, Agents Memo 93-01, 1993.

[FRAKES]
Frakes, William B. and Christopher J. Fox, "Sixteen Questions about Software Reuse," *Communications of the ACM*, Vol. 38, No. 6, June 1995.

[FREEMAN]
Freeman, Peter, Tutorial: Software Reusability, IEEE Computer Society Press, Washington, D.C.,1987.

[FREEDMAN]
Freedman, David, Robert Pisani, and Roger Purves, Statistics, W.W. Norton & Company, Inc., New York, 1978.

[HOLLNAGEL]
Hollnagel, E., "The Intelligent Use of Intelligent Systems," Intelligent Systems: Safety, Reliability and Maintainability Issues, ed. by Okyay Kaynak, Ger Honderd and Edward Grant, Springer-Verlag, New York, 1992.

[HOOPER]
Hooper, James W. and Rowena O. Chester, Software Reuse Guidelines and Methods, Plenum Press, 1991.

[IEEE730]
ANSI/IEEE Std 730-1984, "Standard for Software Quality Assurance Plans," The Institute of Electrical and Electronics Engineers, Inc., 1984.

[IEEE982]
ANSI/IEEE Std 982.2-1988, "Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," The Institute of Electrical and Electronics Engineers, Inc., June 1989.

[IEEE1012]
ANSI/IEEE Std 1012-1986, "IEEE Standard for Software Verification and Validation Plans," The Institute of Electrical and Electronics Engineers, Inc., February 10, 1987.

[IEEE1058]
ANSI/ IEEE Std 1058-1987, Standard for Software Project Management," The Institute of Electrical and Electronics Engineers, Inc., 1987.

[IEEE7432]
ANSI/IEEE Std 7432-1993, "Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations," The Institute of Electrical and Electronics Engineers, Inc., 1993.

[IEEEP1059]
IEEE Std P1059-1994, "(DRAFT 7.1) IEEE Guide for Software Verification and Validation Plans," Institute of Electrical and Electronics Engineers, Inc., May 24, 1993.

[ISO12207]
ISO/IEC 12207, "Information Technology-Software Life Cycle Processes," International Standards Organization/International Electrotechnical Commission, 22 February 1995.

[JOURNAL]
Journal of Systems and Software, Volume 30, Number 3, September, 1995.

[JURAN]
Juran, J. M. (ed.), Juran's Quality Control Handbook, 4th ed., McGraw-Hill, Inc., New York, 1988.

[KIRANI]
Kirani, Shekhar, I.A. Zualkernan, and W.T. Tsai, "Comparative Evaluation of Expert System Testing Methods," *Proceedings of the 1992 IEEE Int. Conference on Tools with AI*, IEEE Computer Society, 1992.

[KITCHENHAM]
Kitchenham, B. A. and B. Littlewood, Measurement for Software Control and Assurance, Elsevier Science Publishers Ltd, London and New York, 1989.

[KOHUT]
Kohout, L.J., J. Anderson and W. Bandler, Knowledge-Based Systems for Multiple Environments, Ashgate Publishing Co., Brookfield, VT, 1992.

[LEVESON93]
Leveson, Nancy G and Clark S. Turner, "An investigation of the Therac-25 accidents," *IEEE Computer*, 26(7):18-41, July 1993.

[LEVESON95]
Leveson, Nancy G., Safeware: System Safety and Computers, Addison Wesley Publishing Company, 1995.

[LYLE]
Lyle, Jim, "Program Slicing," to be published in Encyclopedia of Software Engineering, John Wiley Publishing Co., New York, New York.

[LYU]
Lyu, Michael and Allen Nikora, "Applying Reliability Models More Effectively," *IEEE Software*, Vol. 9., No. 4, July 1992.

[MUSA87]
Musa, J.D., A. Iannino, and K. Okumoto, Software Reliability, Measurement, Prediction, Application, McGraw-Hill, New York, 1987.

[MUSA89]
Musa, J.D., and A.F. Ackerman, "Quantifying Software Validation: When to Stop Testing?" *IEEE Software*, May 1989.

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

[NBS93]
NBS Special Publication 500-93, "Software Validation, Verification, and Testing Technique and Tool Reference Guide," U.S. Department of Commerce/National Bureau of Standards (U.S.), September 1982.

[NGUYEN]
Nguyen, T.A., W.A. Perkins, T.J. Laffey, and D. Pecora, "Knowledge Base Verification," *AI Magazine*, Summer 1987.

[NIST165]
NIST Special Publication 500-165, "Software Verification and Validation: Its Role in Computer Assurance and Its Relationship with Software Project Management Standards," U.S. Department of Commerce/National Institute of Standards and Technology, September 1989.

[NIST190]
NIST Special Publication 500-190, "Proceedings of the Workshop on High Integrity Software; Gaithersburg, MD; Jan. 22-23, 1991," U.S. Department of Commerce/National Institute of Standards and Technology, August 1991.

[NIST204]
NIST Special Publication 500-204, "High Integrity Software Standards and Guidelines," U.S. Department of Commerce/National Institute of Standards and Technology, September 1992.

[NIST209]
NIST Special Publication 500-209, "Software Error Analysis," U.S. Department of Commerce/National Institute of Standards and Technology, April 1993.

[NIST223]

NIST Special Publication 500-223, "A Framework for the Development and Assurance of High Integrity Software," U.S. Department of Commerce/National Institute of Standards and Technology, December 1994.

[NIST4909]

NISTIR 4909, "Software Quality Assurance: Documentation and Reviews," U.S. Department of Commerce/National Institute of Standards and Technology, September 1992.

[NIST5459]

NISTIR 5459, "Quality Characteristics and Metrics for Reusable Software (Preliminary Report)," U.S. Department of Commerce/National Institute of Standards and Technology, May 1994.

[NIST5589]

NISTIR 5589, "A Study on Hazard Analysis in High Integrity Software Standards and Guidelines," U.S. Department of Commerce/National Institute of Standards and Technology, January 1995.

[NUREG6316]

NUREG/CR-6316 (Volume 2), "Guidelines for the Verification and Validation of Expert System Software and Conventional Software (Survey and Assessment of Conventional Software Verification and Validation Methods)," U.S. Nuclear Regulatory Commission, March 1995.

[OKEEFE]

O'Keefe, Robert M. and Daniel E. O'Leary, "Expert System Verification and Validation," *Artificial Intelligence Review: An International Survey and Tutorial Journal*, Vol. 7, No. 1, February 1993.

[OLEARY]

O'Leary, D.E. and N. Kandelin, "Validating the Rule Weights in Rule-Based Expert Systems," *International Journal of Expert Systems*, Vol. 1, No. 3, 1988.

[OPMC]

<u>The Organizational Process Management Cycle Programmed Workbook</u>, Interaction Research Institute, Inc., Fairfax, Virginia.

[PAN]

Pan, Jeff Y.C. and Ja M. Tennenbaum, "An Intelligent Agent Framework for Enterprise Integration," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 21, No. 6, Nov./Dec., 1991.

[PREECE]

Preece, Alun D. and Rajjan Shinghal, "Verifying and Testing Expert System Conceptual Models," *IEEE International Conference on Systems, Man and Cybernetics*, Vol. 1.

[RADATZ]
Radatz, J.W., "Analysis of IV&V Data," RADC-TR-81-145, Logicon, Inc., Rome Air Development Center, Griffiss AFB, NY, June 1981.

[ROOK]
Rook, Paul, Software Reliability Handbook, Elsevier Science Publishers Ltd, London and New York, 1990.

[RTCA178B]
RTCA/DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," RTCA, Inc., December 16, 1992.

[SMITH]
Smith, Gary, Statistical Reasoning, Allyn and Bacon, Boston, MA, 1991.

[SQE]
"Software Measurement," Seminar Notebook, Version 1.2, Software Quality Engineering, 1991.

[SSR]
"Proceedings of the Symposium on Software Reusability: SSR '95," Software Engineering Notes, ACM SIGSOFT, August 1995 .

[TANSLEY]
Tansley, D.S.W. and C.C. Hayball, Knowledge Based Systems Analysis and Design: A KADS Developers Handbook, Prentice Hall, New York, 1993.

[TRACZ]
Tracz, Will, Tutorial: Software Reuse: Emerging Technology, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[TURING]
Turing, A.M., "Computing Machinery and Intelligence," *Mind*, Vol.59, 1950.

[VOAS91]
Voas, J., L. Morell, and K. Miller, "Predicting Where Faults Can Hide From Testing," *IEEE Software*, March 1991.

[VOAS92]
Voas, J., "PIE: A Dynamic Failure-Based Technique," *IEEE Transactions on Software Engineering*, August, 1992.

[VOAS95]
Voas, J. and K. Miller, "Software Testability: The New Verification," *IEEE Software*, May 1995.

[WALLACE91]

Wallace, D.R., D.R. Kuhn, and J.C. Cherniavsky, "Report on a Workshop on the Assurance of High Integrity Software," *Proceedings of the Sixth Annual Conference on Computer Assurance (COMPASS '91), NIST, Gaithersburg, MD, June 24-27, 1991*, The Institute of Electrical and Electronics Engineers, Inc., 1991.

[WALLACE94]

Wallace, Dolores R., "Verification and Validation," Encyclopedia of Software Engineering, Volume 2, John Wiley & Sons, Inc., 1994.

[WILEY]

Encyclopedia of Software Engineering, John Wiley & Sons, Inc., 1994.

[WING]

Wing, Jeannette M., "A Specifier's Introduction to Formal Methods," *COMPUTER*, September 1990.

[ZAGE]

Zage, Wayne M., "Code Metrics and Design Metrics; An ACM Professional Development Seminar," November 19, 1991.

APPENDIX A. SOFTWARE V&V MEASUREMENT

This appendix, condensed from [NIST209], identifies metrics related to software error detection, statistical process control (SPC) techniques, and several software reliability estimation models. Metrics are used to assess the product or process. SPC techniques are used to monitor a project by observing trends, and help to locate major problems in the software development process, the assurance processes (e.g., software quality assurance, software verification and validation (V&V)) and the product itself. Software reliability estimation models provide information about the predicted performance of the software.

Error data from the V&V activities can be collected over the entire project and stored in an organizational database, for use with the current project or future projects. An organizational database may also play an important role in software reuse within an organization. In deciding whether or not to reuse a particular software unit, one can examine its error history to determine whether it satisfies the level of assurance required by the intended application. One can evaluate the component by observing its past failure rates and fault densities to ensure that the component is appropriate for reuse. A software component may sometimes be reused to build a system which is of a higher level of assurance than that in which the component was originally used. The database would provide data on the reliability or other quality attributes to help determine how much additional work is needed to increase the quality of the component to the desired level.

A.1 Metrics

In this report, a metric is defined to be the mathematical definition, algorithm, or function used to obtain a quantitative assessment of a product or process. The actual numerical value produced by a metric is a measure. Thus, for example, cyclomatic complexity is a metric, but the value of this metric is the cyclomatic complexity measure.

Two general classes of metrics include the following:

- management metrics, which assist in the management of the software development process
- quality metrics, which are predictors or indicators of the product qualities

Management metrics can be used for controlling any industrial production or manufacturing activity. They are used to assess resources, cost, and task completion. Quality metrics are used to estimate characteristics or qualities of a software product. Some metrics may be both management metrics and quality metrics, i.e., they can be used for both project control and quality assessment.

A disadvantage of some metrics is that they do not have an interpretation scale which allows for consistent interpretation, as with measuring temperature (in degrees Celsius) or length (in meters). This is particularly true of metrics for software quality characteristics (e.g., maintainability, reliability, usability). Measures must be interpreted relatively, through comparison with plans and expectations, comparison with similar past projects, or comparison with similar components within the current project. While some metrics are mathematically-based, most, including reliability models, have not been proven.

Since there is virtually an infinite number of possible metrics, users must have some criteria for choosing which metrics to apply to their particular projects. Ideally, a metric should possess all of the following characteristics:

- simple - definition and use of the metric is simple
- objective - different people will give identical values; allows for consistency, and prevents individual bias
- easily collected - the cost and effort to obtain the measure is reasonable
- robust - metric is insensitive to irrelevant changes; allows for useful comparison
- valid - metric measures what it is supposed to; this promotes trustworthiness of the measure

A.1.1 General Metrics

Primitive metrics such as those listed below can be collected throughout software development. These metrics can be plotted using bar graphs, histograms, and Pareto charts as part of SPC. The plots can be analyzed by management to identify the activities that are most error prone, to suggest steps to prevent the recurrence of similar errors, to suggest procedures for earlier detection of faults, and to make general improvements to the software development process.

<u>Primitive problem metrics</u>
- Number of problem reports per activity, priority, category, or cause
- Number of reported problems per time period
- Number of open real problems per time period
- Number of closed real problems per time period
- Number of unevaluated problem reports
- Age of open real problem reports
- Age of unevaluated problem reports
- Age of real closed problem reports
- Time when errors are discovered
- Rate of error discovery

<u>Primitive cost and effort metrics</u>
- Time spent
- Elapsed time
- Staff hours
- Staff months
- Staff years

<u>Primitive change metrics</u>
- Number of revisions, additions, deletions, or modifications
- Number of requests to change the software requirements specification and/or software design

Primitive fault metrics
- Number of unresolved faults at planned end of activity
- Number of faults that have not been corrected, and number of outstanding change requests
- Number of software requirements and design faults detected during reviews and walkthroughs

A.1.2 Software Requirements Metrics

The main reasons to measure software requirements specifications is to provide early warnings of quality problems, to enable more accurate project predictions, and to help improve the specifications.

Primitive size metrics. These metrics involve a simple count. Large components are assumed to have a larger number of residual errors, and are more difficult to understand than small components; as a result, their reliability and extendibility may be affected.
- Number of pages or words
- Number of requirements
- Number of functions

Requirements traceability (RT). This metric is used to assess the degree of traceability by measuring the percentage of requirements that has been implemented in the software design. It is also used to identify requirements that are either missing from, or in addition to the original requirements. The measure is computed using the equation: $RT = R1/R2 \times 100\%$, where R1 is the number of requirements met by the architecture (software design), and R2 is the number of original requirements. [IEEE982]

Completeness (CM). This metric is used to determine the completeness of the software specification during requirements activity. This metric uses 18 *primitives* (e.g., number of functions not satisfactorily defined, number of functions, number of defined functions, number of defined functions not used, number of referenced functions, and number of decision points). It then uses 10 *derivatives* (e.g., functions satisfactorily defined, data references having an origin, defined functions used, reference functions defined), which are derived from the primitives. The metric is the weighted sum of the 10 derivatives expressed as $CM = \sum w_i D_i$, where the summation is from $i=1$ to $i=10$, each weight w_i has a value between 0 and 1, the sum of the weights is 1, and each D_i is a derivative with a value between 1 and 0. The values of the primitives also can be used to identify problem areas within the software requirements specification. [IEEE982]

Fault-days number (FD). This metric specifies the number of days that faults spend in the software product from its creation to their removal. This measure uses two primitives: the activity, date, or time that the fault was introduced, and the activity, date, or time that the fault was removed. The fault days for the ith fault, (FD_i), is the number of days from the creation of the fault to its removal. The measure is calculated as follows: $FD = \sum FD_i$. This measure is an indicator of the quality of the software design and software development process. A high value may be indicative of untimely removal of faults and/or existence of many faults, due to an ineffective software development process. [IEEE982]

Function points. This measure was originated by Allan Albrecht at IBM in the late 1970's, and was further developed by Charles Symons. It uses a weighted sum of the number of inputs, outputs,

master files and inquiries in a product to predict development size [ALBRECHT]. To count function points, the first step is to classify each component by using standard guides to rate each component as having low, average, or high complexity. The second basic step is to tabulate function component counts. This is done by entering the appropriate counts in the Function Counting Form, multiplying by the weights on the form, and summing up the totals for each component type to obtain the Unadjusted Function Point Count. The third step is to rate each application characteristic from 0 to 5 using a rating guide, and then adding all the ratings together to obtain the Characteristic Influence Rating. Finally, the number of function points is calculated using the equation below. [SQE]

$$FunctionPoints = UnadjustedFunction * (.65 + .01 * CharacterInfluenceRating)$$

A.1.3 Software Design Metrics

The main reasons for computing metrics during software design are the following: gives early indication of project status; enables selection of alternative designs; identifies potential problems early in the software development process; limits complexity; and helps in deciding how to modularize so the resulting modules are both testable and maintainable. In general, good design practices involve high cohesion of modules, low coupling of modules, and effective modularity. [ZAGE]

Primitive size metrics. These metrics are used to estimate the size of the software design or software design documents.
- Number of pages or words
- DLOC (lines of PDL)
- Number of modules
- Number of functions
- Number of inputs and outputs
- Number of interfaces

(Estimated) number of modules (NM). This metric provides measure of product size, against which the completeness of subsequent module based activities can be assessed. The estimate for the number of modules is given by, NM = S/M, where S is the estimated size in LOC, M is the median module size found in similar projects. The estimate NM can be compared to the median number of modules for other projects. [ROOK]

Primitive fault metrics. These metrics identify potentially fault-prone modules. [ROOK]
- Number of faults associated with each module
- Number of requirements faults and structural design faults detected during detailed design

Primitive complexity metrics. These metrics identify modules which are complex or hard to test. [ROOK]
- Number of parameters per module
- Number of states or data partitions per parameter
- Number of branches in each module

Coupling. Coupling is the manner and degree of interdependence between software modules [IEEE982]. Module coupling is rated based on the type of coupling, using a standard rating chart, which can be found in [SQE]. According to the chart, *data coupling* is the best type of coupling, and *content coupling* is the worst. The better the coupling, the lower the rating. [SQE, ZAGE]

Cohesion. Cohesion is the degree to which the tasks performed within a single software module are related to the module's purpose. The module cohesion value for a module is assigned using a standard rating chart, which can be found in [SQE]. According to the chart, the best cohesion level is *functional*, and the worst is *coincidental*, with the better levels having lower values. Case studies have shown that fault rate correlates highly with cohesion strength. [SQE, ZAGE]

(Structural) fan-in / fan-out. Fan-in/fan-out represents the number of modules that *call/are called by* a given module. Identifies whether the system decomposition is adequate (e.g., no modules which cause bottlenecks, no missing levels in the hierarchical decomposition, no unused modules ("dead" code), identification of critical modules). May be useful to compute maximum, average, and total fan-in/fan-out. [ROOK, IEEE982]

Information flow metric (C). This metric represents the total number of combinations of an input source to an output destination, given by, $C = C_i \times (\text{fan-in} \times \text{fan-out})^2$, where C_i is a code metric, which may be omitted. The product inside the parentheses represents the total number of paths through a module. [ZAGE]

Staff hours per major defect detected (M). This metric is used to evaluate the efficiency of the design inspection. The following primitives are used: time expended in preparation for inspection meeting (T1), time expended in conduct of inspection meeting (T2), number of major defects detected during the ith inspection (S_i), and total number of inspections to date (i). The staff hours per major defect detected is given below, with the summations being from i=1 to i=i. This measure is applied to new code, and should fall between three and five. If there is significant deviation from this range, then the matter should be investigated. (May be adapted for code inspections). [IEEE982]

$$M = \frac{\sum (T1 + T2)_i}{\sum S_i}$$

Defect Density (DD). Used after design inspections of new development or large block modifications in order to assess the inspection process. The following primitives are used: total number of unique defects detected during the ith inspection or ith software development activity (D_i), total number of inspections to date (i), and number of source lines of design statements in thousands (KSLOD). The measure is calculated by the ratio below, where the sum is from i=1 to i=i. This measure can also be used in the implementation activity, in which case the number of source lines of executable code in thousands (KSLOC) should be substituted for KSLOD. [IEEE982]

$$DD = \frac{\sum D_i}{KSLOD}$$

Test related primitives. These metrics check that each module will be/has been adequately tested, or assesses the effectiveness of early testing activities. [ROOK]
- Number of software integration test cases planned/executed involving each module
- Number of black box test cases planned/executed per module
- Number of requirements faults detected (and re-assesses quality of requirements specification)

A.1.4 Code Metrics

Lines of Code (LOC). Although lines of code is one of the most popular metrics, it has no standard definition. The predominant definition for LOC is "any line of a program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line." [SQE] It is an indication of size, which allows for estimation of effort, time scale, and total number of faults. For the same application, the length of a program partly depends on the language the code is written in, thus making comparison using LOC difficult. However, LOC can be a useful measure if the projects being compared are consistent in their development methods (e.g., use the same language, coding style). Because of its disadvantages, the use of LOC as a management metric (e.g., for project sizing beginning from the software requirements activity) is controversial, but there are uses for this metric in error analysis, such as to estimate the values of other metrics. The advantages of this metric are that it is conceptually simple, easily automated, and inexpensive. [SQE]

Halstead software science metrics. This set of metrics was developed by Maurice Halstead, who claimed they could be used to evaluate the mental effort and time required to create a program, and how compactly a program is expressed. These metrics are based on four primitives listed below:

n_1 = number of unique operators
n_2 = number of unique operands
N_1 = total occurrences of operators
N_2 = total occurrences of operands

The program length measure, N, is the sum of N_1 and N_2. Other software science metrics are listed below. [SQE]

Vocabulary: $n = n_1 + n_2$
Predicted length: $N\hat{} = (n_1 * \log_2 n_1) + (n_2 * \log_2 n_2)$
Program volume: $V = N * \log_2 n$
Effort: $E = (n_1 N_2 N \log_2 n)/(2n_2)$
Time: $T = E/\beta$; Halstead β=18
Predicted number of bugs: $B = V/3000$

Number of entries/exits per module. Used to assess the complexity of a software architecture, by counting the number of entry and exit points for each module. The equation to determine the measure for the ith module is simply $m_i = e_i + x_i$, where e_i is the number of entry points for the ith module, and x_i is the number of exit points for the ith module. [IEEE982]

Cyclomatic complexity (C). Used to determine the structural complexity of a coded module in order to limit its complexity, thus promoting understandability. In general, high complexity leads to a high

number of defects and maintenance costs. Also used to identify minimum number of test paths to assure test coverage. The primitives for this measure include the number of nodes (N), and the number of edges (E), which can be determined from the graph representing the module. The measure can then be computed with the formula, $C = E - N + 1$. [IEEE982, SQE]

Amount of data. This measure can be determined by primitive metrics such as Halstead's n_2 and N_2, number of inputs/outputs, or the number of variables. These primitive metrics can be obtained from a compiler cross reference. [SQE]

Live variables. For each line in a section of code, determine the number of live variables (i.e., variables whose values could change during execution of that section of code). The average number of live variables per line of code is the sum of the number of live variables for each line, divided by the number of lines of code. [SQE]

Variable scope. The variable scope is the number of source statements between the first and last reference of the variable. For example, if variable A is first referenced on line 10, and last referenced on line 20, then the variable scope for A is 9. To determine the average variable scope for variables in a particular section of code, first determine the variable scope for each variable, sum up these values, and divide by the number of variables [SQE]. With large scopes, the understandability and readability of the code is reduced.

Variable spans. The variable span is the number of source statements between successive references of the variable. For each variable, the average span can be computed. For example, if the variable X is referenced on lines 13, 18, 20, 21, and 23, the average span would be the sum of all the spans divided by the number of spans, i.e., (4+1+0+1)/4 = 1.5. With large spans, it is more likely that a far back reference will be forgotten. [SQE]

A.1.5 Test Metrics

Primitive defect/error/fault metrics. These metrics can be effectively used with SPC techniques, such as bar charts, and Pareto diagrams. These metrics can also be used to form percentages (e.g., percentage of logic errors = number of logic errors ÷ total number of errors).
- Number of faults detected in each module
- Number of requirements, design, and coding faults found during unit and integration testing
- Number of errors by type (e.g., logic, computational, interface, documentation)
- Number of errors by cause or origin
- Number of errors by severity (e.g., critical, major, cosmetic)

Fault density (FD). This measure is computed by dividing the number of faults by the size (usually in KLOC, thousands of lines of code). It may be weighted by severity using the equation

$$FD_w = (W_1\ S/N + W_2\ A/N + W_3\ M/N) / Size$$

where N = total number of faults
S = number of severe faults
A = number of average severity faults

M = number of minor faults
W_i = weighting factors (defaults are 10, 3, and 1)

FD can be used to perform the following: predict remaining faults by comparison with expected fault density; determine if sufficient testing has been completed based on predetermined goals; establish standard fault densities for comparison and prediction. [IEEE982, SQE]

Defect age. Defect age is the time between when a defect is introduced to when it is detected or fixed. Assign the numbers 1 through 6 to each of the software development activities from software requirements to software operation and maintenance. The defect age is computed as shown. [SQE]

$$AverageDefectAge = \frac{\sum (ActivityDetected - ActivityIntroduced)}{NumberDefects}$$

Defect response time. This measure is the time between when a defect is detected to when it is fixed or closed. [SQE]

Defect cost. The cost of a defect may be a sum of the cost to analyze the defect, the cost to fix it, and the cost of failures already incurred due to the defect. [SQE]

Defect removal efficiency (DRE). The DRE is the percentage of defects that have been removed during an activity, computed with the equation below. The DRE can also be computed for each software development activity and plotted on a bar graph to show the relative defect removal efficiencies for each activity. Or, the DRE may be computed for a specific task or technique (e.g., design inspection, code walkthrough, unit test, 6-month operation, etc.). [SQE]

$$DRE = \frac{NumberDefectsRemoved}{NumberDefectsAtStartOfProcess} * 100$$

Primitive test case metrics
- Total number of planned white/black box test cases run to completion
- Number of planned software integration tests run to completion
- Number of unplanned test cases required during test activity

Statement coverage. Measures the percentage of statements executed (to assure that each statement has been tested at least once). [SQE]

Branch coverage. Measures the percentage of branches executed. [SQE]

Path coverage. Measures the percentage of program paths executed. It is generally impractical and inefficient to test all the paths in a program. The count of the number of paths may be reduced by treating all possible loop iterations as one path. [SQE] Path coverage may be used to ensure 100% coverage of critical (safety or security related) paths.

Data flow coverage. Measures the definition and use of variables and data structures. [SQE]

Test coverage. Measures the completeness of the testing activity. Test coverage is the percentage of requirements implemented (in the form of defined test cases or functional capabilities) multiplied by the percentage of the software structure (in units, segments, statements, branches, or path test results) tested. [AIRFORCE]

Mean time to failure (MTTF). Gives an estimate of the mean time to the next failure, by accurately recording failure times t_i, the elapsed time between the ith and the (i-1)st failures, and computing the average of all the failure times. This metric is the basic parameter required by most software reliability models. High values imply good reliability. [IEEE982]

Failure rate. Used to indicate the growth in the software reliability as a function of test time and is usually used with reliability models. This metric requires two primitives: t_i, the observed time between failures for a given severity level *i*, and f_i, the number of failures of a given severity level in the ith time interval. The failure rate $\lambda(t)$ can be estimated from the reliability function R(t), which is obtained from the cumulative probability distribution F(t) of the time until the next failure, using a software reliability estimation model, such as the nonhomogeneous Poisson process (NHPP) or Bayesian type model. The failure rate is as shown below, where R(t) = 1 - F(t). [IEEE982]

$$\lambda(t) = -1/R(t)\frac{[dR(t)]}{dt}$$

Cumulative failure profile. Uses a graphical technique to predict reliability, to estimate additional testing time needed to reach an acceptable reliability level, and to identify modules and subsystems that require additional testing. This metric requires one primitive, f_i, the total number of failures of a given severity level *i* in a given time interval. Cumulative failures are plotted on a time scale. The shape of the curve is used to project when testing will be complete, and to assess reliability. It can provide an indication of clustering of faults in modules, suggesting further testing for these modules. A nonasymptotic curve also indicates the need for continued testing. [IEEE982]

A.1.6 Software Installation Metrics

Most of the test metrics are also applicable during software installation. The specific metrics used will depend on the type of testing performed. If acceptance testing is conducted, a requirements trace may be performed to determine what percentage of the software requirements are satisfied in the product (i.e., number of software requirements fulfilled divided by the total number of software requirements).

A.1.7 Software Operation and Maintenance Metrics

Every metric that can be applied during software development may also be applied during software maintenance. The purposes may differ somewhat. For example, software requirements traceability may be used to ensure that software maintenance requirements are related to predecessor requirements, and that the test activity covers the same test areas as for the development. Metrics

that were used during software development may be used again during software maintenance for comparison purposes (e.g., measuring the complexity of a module before and after modification). Elements of support, such as customer perceptions, training, hotlines, documentation, and user manuals, can also be measured.

Primitive change metrics
- Number of changes
- Cost/effort of changes
- Time required for each change
- LOC added, deleted, or modified
- Number of fixes, or enhancements

Customer ratings. These metrics are based on results of customer surveys, which ask customers to provide a rating or a satisfaction score (e.g., on a scale of one to ten) of a vendor's product or customer services (e.g., hotlines, fixes, user manual). Ratings and scores can be tabulated and plotted in bar graphs.

Customer service metrics
- Number of hotline calls received
- Number of fixes for each type of product
- Number of hours required for fixes
- Number of hours for training (for each type of product)

A.2 Statistical Process Control Techniques

Statistical process control (SPC) is the application of statistical methods to provide the information necessary to continuously control or improve activities throughout the entire development of a product [OPMC]. SPC techniques help to locate trends, cycles, and irregularities within the software development process and provide clues about how well the process meets specifications or requirements. They are tools for measuring and understanding process variation and distinguishing between random inherent variations and significant deviations so that correct decisions can be made about whether to make changes to the process or product.

To fully understand a process, it is necessary to determine how the process changes over time. To do this, one can plot error data (e.g., total number of errors, counts of specific types of errors) over a period of time (e.g., days, weeks) and then interpret the resulting pattern. If, for instance, a large number of errors are found in a particular software development activity, an investigation of the tasks in that activity or preceding ones may reveal that necessary development tasks were omitted (e.g., code reviews were not conducted during the code activity). A plot of the sources of errors may show that a particular group is the most frequent source of errors. Further investigation may confirm that members of the group do not have sufficient experience and training. A plot of the number of specific types of errors may show that many errors are related to incorrect or unclear software requirements specifications (e.g., software requirements are written in a way that consistently causes misinterpretations, or they fail to list enough conditions and restrictions). This would indicate that the software requirements activity needs to be modified.

There are several advantages to using SPC techniques. First, errors may be detected earlier or prevented altogether. By monitoring the software development process, the cause of the error (e.g., inadequate standards, insufficient training, incompatible hardware) may be detected before additional errors are created. Second, using SPC techniques is cost-effective, because less effort may be required to ensure that processes are operating correctly than is required to perform detailed checks on all the outputs of that process. Thus, higher quality may be achieved at a lower development expense. Finally, use of SPC techniques provides quantitative measures of progress and of problems so less guesswork is required [DEMMY].

Despite the advantages, there are also several potential disadvantages. To be successful, SPC requires discipline, planning, continuous commitment to the timely solution of process problems, and frequent access to information from the software development process [DEMMY].

A.2.1 Control Charts

The primary statistical technique used to assess process variation is the control chart. The control chart displays sequential process measurements relative to the overall process average and control limits. The upper and lower control limits establish the boundaries of normal variation for the process being measured. Variation within control limits is attributable to random or chance causes, while variation beyond control limits indicates a process change due to causes other than chance -- a condition that may require investigation. [OPMC] The upper control limit (UCL) and lower control limit (LCL) give the boundaries within which observed fluctuations are typical and acceptable. They are usually set, respectively, at three standard deviations above and below the mean of all observations. There are many different types of control charts, pn, p, c, etc., which are described in Table A-1. This section is based on [OPMC], [SMITH], [CAPRIO], and [JURAN].

Implementation

1. Identify the purpose and the characteristics of the process to be monitored.

2. Select the appropriate type of control chart based on the type of characteristic measured, the data available, and the purpose of the application.

3. Determine the sampling method (e.g., number of samples (n), size of samples, time frame).

4. Collect the data.

5. Calculate the sample statistics: average, standard deviation, upper and lower control limits.

6. Construct the control chart based on sample statistics.

7. Monitor the process by observing pattern of the data points and whether they fall within control limits.

Table A-1. Types of Control Charts

TYPE	DESCRIPTION	IMPLEMENTATION
np	number of nonconforming units (e.g., number of defective units)	The number of units in each sample with the selected characteristic is plotted; sample size is constant.
p	fraction of nonconforming units (e.g., fraction of defective units)	For each sample, the fraction nonconforming, obtained by dividing the number nonconforming by the total number of units observed, is plotted; sample size can change.
c	number of nonconformities (e.g., number of errors)	For each sample, the number of occurrences of the characteristic in a group is plotted; sample size is constant.
u	number of nonconformities per unit (e.g., number of errors per unit)	For each sample, the number of nonconformities per unit, obtained by dividing the number of nonconformities by the number of units observed, is plotted; sample size can change.
X	single observed value	The value for each sample of size 1 is plotted.
XB	X-Bar	For each sample, the mean of 2 to 10 observations (4 or 5 are optimal) is plotted.
R	range	The difference between the largest and smallest values in each sample is plotted.
XM	median	The median of each sample is plotted.
MR	moving range	The difference between adjacent measurements in each sample is plotted.

Interpretation

The existence of *outliers*, or data points beyond control limits, indicates that nontypical circumstances exist. A *run*, or consecutive points on one side of the average line (8 in a row, or 11 of 12, etc.) indicates a shift in process average. A *sawtooth* pattern, which is a successive up and down trend with no data points near the average line, indicates over adjustment or the existence of two processes. A *trend*, or steady inclining or declining progression of data points represents gradual change in the process. A *hug*, in which all data points fall near the average line, may indicate unreliable data. A *cycle*, or a series of data points which is repeated to form a pattern, indicates a cycling process.

Application Examples

Control charts are applicable to almost any measurable activity. Some examples for software include the following: number of defects/errors, training efforts, execution time, and number of problem reports per time period. An example of an np control chart with hypothetical data is shown in Figure A-1. In this example, the number of samples (n) is 100. Each data point represents the number of defects found in the software product in a work week.

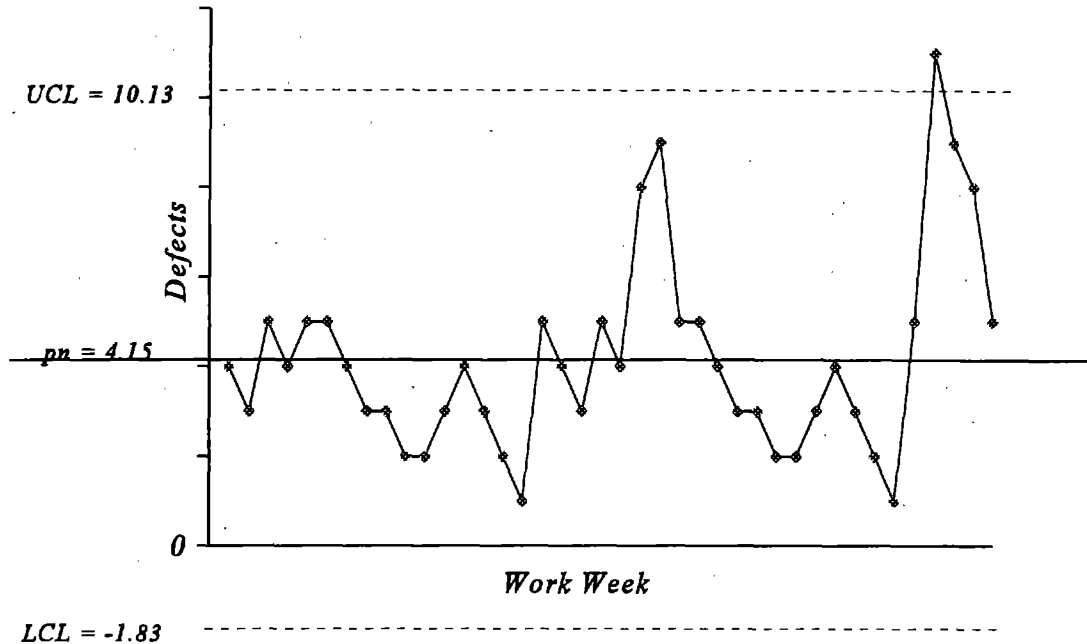


Figure A-1 np Control Chart.

A.2.2 Run Chart

A run chart is a simplified control chart, in which the upper and lower control limits are omitted. The purpose of the run chart is more to determine trends in a process, rather than its variation. Although very simple, run charts can be used effectively to monitor a process, e.g., to detect sudden changes and to assess the effects of corrective actions. Run charts provide the input for establishing control charts after a process has matured or stabilized in time. Limitations of this technique are that it analyzes only one characteristic over time, and it does not indicate if a single data point is an outlier. This section is based on [OPMC] and [CAPRIO].

Implementation

1. Decide which outputs of a process to measure.
2. Collect the data.
3. Compute and draw the average line.
4. Plot the individual measurements chronologically.
5. Connect data points for ease of interpretation.

Interpretation - See Interpretation for Control Charts.

Application Examples

Run charts are applicable to almost any measurable activity. Some examples for software include the following: number of defects/errors, number of failures, execution time, and downtime.

A.2.3 Bar Graph

A bar graph is a frequency distribution diagram in which each bar represents a characteristic, and the height of the bar represents the frequency of that characteristic. The horizontal axis may represent a continuous numerical scale, or a discrete non-numerical scale. Generally, numerical-scale bar charts in which the bars have equal widths are more useful for comparison purposes; numerical-scale bar charts with unequal intervals can be misleading because the characteristics with the largest bars (in terms of area) do not necessarily have the highest frequency. This section is based on [SMITH].

Implementation

1. Define the subject and purpose.

2. Collect the data. Check that the sample size is sufficient.

3. Sort the data by frequency (or other measure) of characteristics.

4. For numerical-scale bar charts, determine the number of bars and the width of the bars (class width), by trying series of class widths, avoiding too fine or too coarse a granularity.

5. Construct the chart and draw the bars. The height of a bar represents the frequency of the corresponding characteristic.

Interpretation

In a simple bar graph in which the characteristics being measured are discrete and non-numerical or if each bar has the same width, the measures for each characteristic can be compared simply by comparing the heights of the bars. For numerical-scale graphs with unequal widths, one should remember not to interpret large bars as necessarily meaning that a large proportion of the entire population falls in that range.

Application Examples

Bar graphs are mostly used to compare the frequencies of different attributes. For example, in Figure A-2, it is used to plot the average customer rating for each evaluation category (e.g., customer service, hotlines, overall satisfaction). The graph shows that Category D has the highest rating.

Other examples of characteristics that may be plotted include: number or percentage of problem reports by software development activity or by type.

A.2.4 Pareto Diagram

A Pareto diagram is a bar graph in which the bars are arranged in descending order of magnitude. The purpose of Pareto analysis is to identify the major problems in a product or process, or to identify the most significant causes for a given effect. This allows a developer to prioritize problems and decide which problem area to work on first. This section is based on [OPMC] and [CAPRIO].

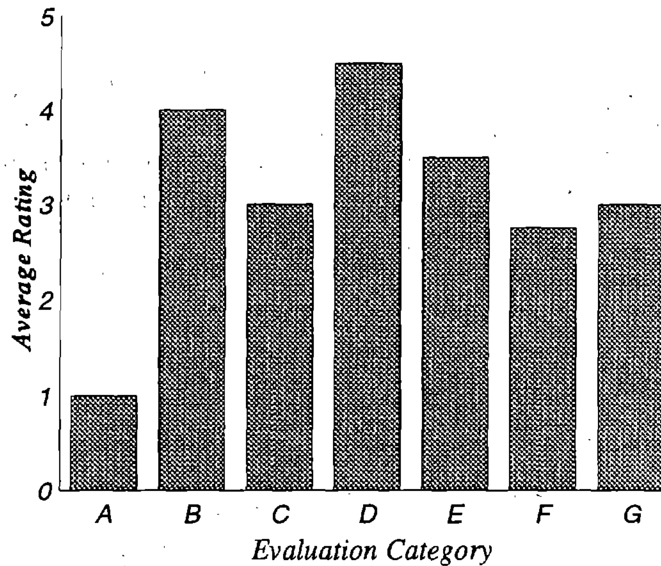


Figure A-2 Bar Chart.

Implementation

1. Construct a bar graph, except the bars should be in descending order of magnitude (height).
2. Determine the "vital few" cause: draw a cumulative percent line and applying the 20/80 rule.
3. Compare/identify the major causes. Repeat until root cause of the problem is revealed.

Interpretation

Pareto analysis is based on the 20/80 rule, which states that approximately 20% of the causes (the "vital few") account for 80% of the effects (problems). The "vital few" can be determined by drawing a cumulative percent line and noting which bars are to the left of the point marking 80% of the total count. In Figure A-3, the vital few are logic, computational, and interface errors since 80% of the errors are found in these modules. By knowing the primary causes of a problem or effect, the developer can decide where efforts should be concentrated.

Application Examples

Most data that can be plotted on a non-numerical scale bar graph can also be plotted on a Pareto diagram. Examples include: number or percentage of errors by type, by cause, or by software development activity, and number or percentage of problem reports by type or by software development activity.

A.2.5 Scatter Diagram

A scatter diagram is a plot of the values of one variable against those of another variable to determine the relationship between them. This technique was popularized by Walter Shewhart at Bell Laboratories. Scatter diagrams are used during analysis to understand the cause and effect

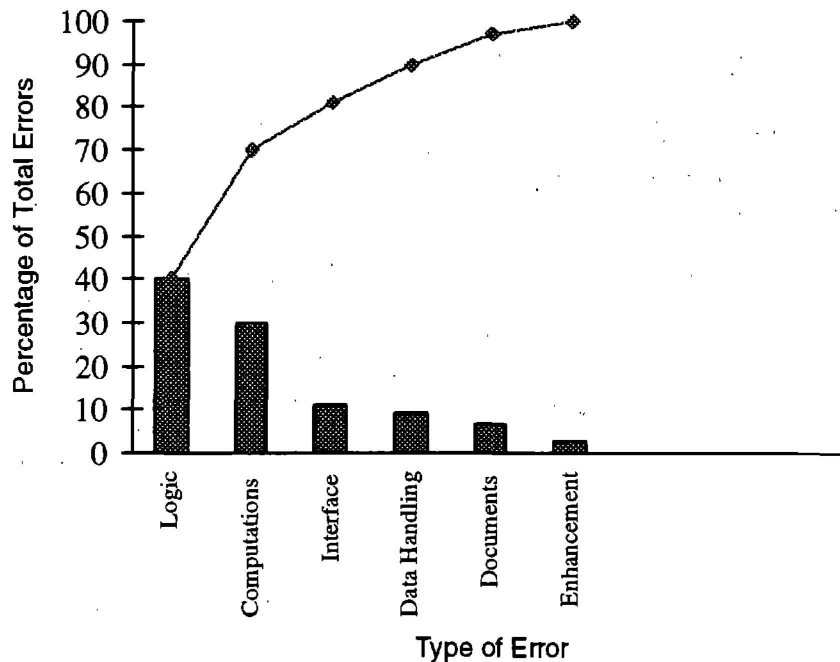


Figure A-3 Pareto Chart.

relationship between two variables. They are also called correlation diagrams. This section is based on [KITCHENHAM], [OPMC], and [CAPRIO].

Implementation

1. Define the subject and select the variables.

2. Collect the data.

3. Plot the data points using an appropriate scale.

4. Examine the pattern to determine whether any correlation exists (e.g., positive, negative). For a more precise specification of the relationship, regression, curve fitting or smoothing techniques can be applied.

Interpretation

If the data points fall approximately in a straight line, this indicates that there is a linear relationship, which is positive or negative, depending on whether the slope of the line is positive or negative. Further analysis using the method of least squares can be performed. If the data points form a curve, then there is a non-linear relationship. If there is no apparent pattern, this may indicate no relationship. However, another sample should be taken before making such a conclusion.

Application Examples

The following are examples of pairs of variables that might be plotted:

- complexity vs. defect density (example shown in fig. A-4)
- effort vs. duration (of an activity)
- failures vs. time
- failures vs. size
- cost vs. time

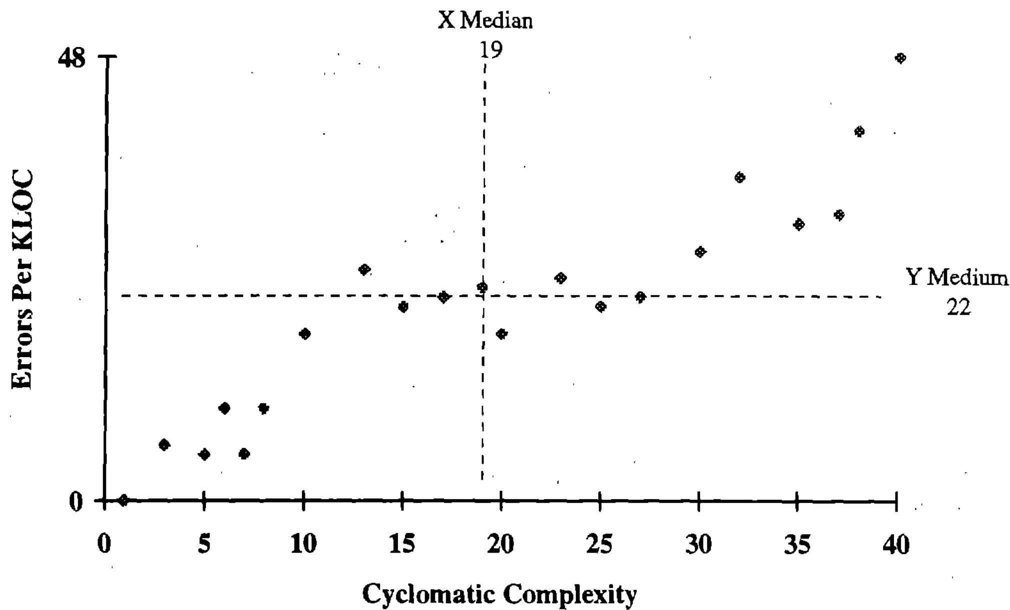


Figure A-4 Scatter Diagram.

A.2.6. Method of Least Squares (Regression Technique)

This technique can be used in conjunction with scatter diagrams to obtain a more precise relationship between variables. It is used to determine the equation of the regression line, i.e., the line that "best fits" the data points. With this equation, one can approximate values of one variable when given values of the other. The equation of the line is $Y = a + bX$, where a and b are constants which minimize S, the sum of squares of the deviations of all data points from the regression line. For any sample value x_i of X, the expected Y value is $a + bx_i$. This section is based on [OPMC], [CAPRIO], and [SMITH].

Implementation

1. Collect n data values for each of the 2 variables, X and Y, denoted by $x_1, x_2, \ldots, x_n$ and $y_1, y_2, \ldots, y_n$.

2. Minimize $S = \Sigma (y_i - a - bx_i)^2$ by first taking the partial derivative of S with respect to a and then with respect to b, setting these derivatives to zero, and then solving for a and b.

3. The results obtained from steps should be the following, where $X_B = \Sigma x_i/n$ and $Y_B = \Sigma y_i/n$:

$$b=\frac{\sum (X_i-X_B)(Y_i-Y_B)}{\sum (X_i-X_B)^2} \qquad a=Y_B-bX_B$$

Interpretation

The constant *a* represents the intercept of the regression line, i.e., the value of Y when X is 0, and *b* represents the slope of the regression line. The idea of this technique is to minimize S, so that all data points will be as close to the regression line as possible. The reason for taking the squares of the deviations, rather than simply the deviations, is so that positive and negative deviations will not cancel each other when they are summed. It would also be possible to sum the absolute values of the deviations, but absolute values are generally harder to work with than squares.

Application Examples

See Application Examples for Scatter Diagrams.

A.3 Software Reliability Estimation Models

"Reliability" is used in a general sense to express a degree of confidence that a part or system will successfully function in a certain environment during a specified time period [JURAN]. Software reliability estimation models can predict the future behavior of a software product, based on its past behavior, usually in terms of failure rates. Since 1972, more than 40 software reliability estimation models have been developed, with each based on a certain set of assumptions characterizing the environment generating the data. However, in spite of much research effort, there is no universally applicable software reliability estimation model which can be trusted to give accurate predictions of reliability in all circumstances [BROCKLEHURST].

It is usually possible to obtain accurate reliability predictions for software, and to have confidence in their accuracy, if appropriate data is used [ROOK]. Also, the use of reliability estimation models is still under active research, so improvements to model capability are likely. Work by Littlewood (1989), for example, involves the use of techniques for improving the accuracy of predictions by learning from the analysis of past errors [ROOK], and recalibration [BROCKLEHURST].

Some problems have been encountered by those who have tried to apply reliability estimation models in practice. The algorithms used to estimate the model parameters may fail to converge. When they do, the estimates can vary widely as more data is entered [DACS]. There is also the difficulty of choosing which reliability model to use, especially since one can not know a priori which of the many models is most suitable in a particular context [BROCKLEHURST]. In general, the use of these models is only suitable for situations in which fairly modest reliability levels are required [ROOK].

There are three general classes of software reliability estimation models: nonhomogeneous Poisson process (NHPP) models, exponential renewal NHPP models, and Bayesian models. Some of the more common reliability estimation models are described below [DUNN], [LYU].

- Jelinski-Moranda (JM). One of the earliest models, it assumes the debugging process is purely deterministic, that is, that each defect in the program is equally likely to produce failure (but at random times), and that each fix is perfect, i.e., introduces no new defects. It also assumes that the failure rate is proportional to the number of remaining defects and remains constant between failures. This model tends to be too optimistic and to underestimate the number of remaining faults; this effect has been observed in several actual data sets.

- Goel-Okumoto (GO). This model is similar to JM, except it assumes the failure rate (number of failure occurrences per unit of time) improves continuously in time.

- Yamada Delayed S-Shape. This model is similar to GO, except it accounts for the learning period that testers go through as they become familiar with the software at the start of testing.

- Musa-Okumoto (MO). This NHPP model is similar to GO, except it assumes that later fixes have a smaller effect on a program's reliability than earlier ones. Failures are assumed to be independent of each other.

- Geometric. This model is a variation of JM, which does not assume a fixed, finite number of program errors, nor does it assume that errors are equally likely to occur.

- Schneidewind. Similar to JM, this model assumes that as testing proceeds with time, the error detection process changes, and that recent error counts are usually more useful than earlier counts in predicting future counts.

- Bayesian Jelinski-Moranda (BJM). This model is similar to JM, except that it uses a Bayesian inference scheme, rather than maximum likelihood. Although BJM does not drastically underestimate the number of remaining errors, it does not offer significant improvement over JM. Actual reliability predictions of the two models are usually very close.

- Littlewood. This model attempts to answer the criticisms of JM and BJM by assuming that different faults have different sizes, i.e., they contribute unequally to the unreliability of the software. This assumption represents the uncertainty about the effect of a fix.

- Littlewood-Verrall (LV). This model takes into account the uncertainty of fault size and efficacy of a fix (i.e., a fix is of uncertain magnitude and may make a program less reliable), by letting the size of the improvement in the failure rate at a fix vary randomly.

- Brooks and Motley (BM). The BM binomial and Poisson models attempt to consider that not all of a program is tested equally during a testing period and that only some portions of the program may be available for testing during its development.

- Duane. This model assumes that the failure rate changes continuously in time, i.e., it follows a nonhomogeneous Poisson process. The cumulative failure rate when plotted against the total testing time on a ln-ln graph follows a straight line. The two parameters for the equation of the line can be derived using the method of least squares.

Implementation

The following is a generic procedure for estimating software reliability [AIAA]. It can be tailored to a specific project or software development activity; thus some steps may not be used in some applications.

1. Identify the application. The description of the application should include, at a minimum, the identification of the application, the characteristics of the application domain that may affect reliability, and details of the intended operation of the application system.

2. Specify the requirement. The reliability requirement should be specific enough to serve as a goal (e.g., failure rate of 10^{-9} per hour).

3. Allocate the requirement. The reliability requirement may be distributed over several components, which should be identified.

4. Define failure. A specific failure definition is usually agreed upon by testers, developers, and users prior to the beginning of testing. The definition should be consistent over the life of the project. Classification of failures (e.g., by severity) is continuously negotiated.

5. Characterize the operational environment. The operational environment should be described in terms of the system configuration (arrangement of the system's components), system evolution and system operational profile (how system will be used).

6. Select tests. The test team selects the most appropriate tests for exposing faults. Two approaches to testing can be taken: testing duplicates actual operational environments as closely as possible; or testing is conducted under more severe conditions than expected in normal operational environments, so that failures can occur in less time.

7. Select the models. The user should compare the models prior to selection based on the following criteria: predictive validity, ease of parameter measurement, quality of the model's assumptions, capability, applicability, simplicity, insensitivity to noise, and sensitivity to parameter variations.

8. Collect data.

9. Determine the parameters. There are three common methods of estimating the parameters from the data: method of moments, least squares, and maximum likelihood. Each of these methods has useful attributes, but maximum likelihood estimation is the most commonly used approach. As stated previously, some data sets may cause the numerical methods not to

converge. There exist automated software reliability engineering tools, which are capable of performing parameter estimation.

10. Validate the model. The model should be continuously checked to verify that it fits the data, by using a predictive validity criteria or a traditional statistical goodness-of-fit test (e.g., Chi-square).

11. Perform analysis. The results of software reliability estimation may be used for several purposes, including, but not limited to, estimating current reliability, forecasting achievement of a reliability goal, establishing conformance with acceptance criteria, managing entry of new software features or new technology into an existing system, or supporting safety certification.

Interpretation

A disadvantage of these models is that they rely on testing and hence are used rather late in the software development process. The models are usually time based, that is, the probability is based on time to failure. Research is needed to identify how to use more valuable parameters with these models. See [ROOK].

Application Examples

Applicability of the models should be examined through various sizes, structures, functions and application domains. An advantage of a reliability model is its usability in different development and operational environments, and in different software development activities. Software reliability models should be used when dealing with the following situations:

- evolving software (i.e., software that is incrementally integrated during testing)
- classification of failure severity
- incomplete failure data
- hardware execution rate differences
- multiple installations of the same software
- project environments departing from model assumptions

INDEX

Page(s)

Accuracy . . . v, 7, 9, 11-17, 23, 26, 29, 30, 33, 76
Actual and formal parameters mismatch . . . 29
Algorithm analysis . . . 23, 25, 26, 35
Algorithm efficiency . . . 26, 31
Allocation of V&V resources . . . 31
Alternative model . . . 33, 36
Anachronistic data . . . 28
Analytic modeling . . . 26, 35
Anomalies or discrepancies between versions . . . 26
Array size . . . 26
Back-to-back testing . . . 26, 35
Behavior . . . 10, 15, 25, 30, 31, 45, 46, 48, 76
Bottlenecks . . . 26, 27, 31, 32, 63
Boundary test cases . . . 27, 28, 30, 31
Boundary value analysis . . . 26, 35
Branch and path identification . . . 27, 28, 30, 31
Branch testing . . . 27, 28, 30, 31
Calls to subprograms that do not exist . . . 28
Cell structure of units . . . 27
Checklists . . . 29, 30, 33
Code reading . . . 24, 26, 35
Code V&V . . . 4, 10, 15, 24
Common code . . . 31
Completeness . . . 11-17, 19, 30, 33, 34, 61, 62, 67
Consistency . . . 11, 12, 14-16, 19, 26, 27, 29-31, 33, 36, 60
Consistency analysis . . . 33, 36
Consistency in computation . . . 26
Control flow analysis . . . 5, 23-26, 35
Control groups . . . 33, 36
Correctness . . . v, 1, 3, 7, 10-16, 23, 24, 26, 27, 30-32, 35, 51
Coverage analysis . . . 27, 35
Credibility analysis . . . 33, 37
Critical timing/flow analysis . . . 27, 35
Criticality analysis . . . vii, 7-9, 25, 32, 36, 41
Data characteristics . . . 27
Data fields unconstrained by data boundaries . . . 28
Data flow analysis . . . 23, 27, 35
Database analysis . . . 23, 24, 27, 35
Decision (truth) tables . . . 27, 35
Design errors . . . 9, 32
Design evaluation . . . 4, 14, 27
Desk checking . . . 27, 35

Preceding page blank

Dynamic analysis 23, 29
Effective forerunners to testing 29, 31, 33
Environment interaction 27, 30, 31
Error propagation 26, 27, 32, 33
Error seeding 28, 35
Evaluation of program paths 27, 32
Execution monitoring, sampling, support 31
Expected vs actual results 27
Failure to implement the design 28
Failure to save or restore registers 28
Feasibility 11, 26, 31, 52
Field testing 33, 37
File sequence error 27, 28, 31
Finite state machines . . . . . vii, 28, 35
Formal specification evaluation 27, 29, 31, 33
Functional testing 24, 28, 35
Global information flow and consistency 27
Go-no-go decisions 29, 33
Hazard analysis 2, 7-9, 28, 30, 32, 56
Hierarchical interrelationship of units 27, 31
Illegal attribute testing 33, 37
Improper nesting of loops and branches 28
Improper program linkages 28
Improper sequencing of processes 28
Inaccessible code 27
Incomplete predicates 28
Incomplete software requirements specification 28, 30, 32
Inconsistencies between limits 26
Inconsistencies between subroutine usage list and called subrout 29
Inconsistency of attributes of global variables 29
Inconsistent interface parameters 29
Inconsistent software requirements 28
Incorrect access of array components 28
Incorrect assumptions about static and dynamic storage of values 29
Incorrect functions used or incorrect subroutine called 29
Incorrect product version shipped 32
Incorrect test results 32
Inefficient data transport 28
Infinite loops 28
Information flow consistency 29, 31
Initialization faults 28
Input-output description errors 29
Input-output faults 28
Inspections 23, 24, 29, 35, 63
Instruction modification 28
Inter-unit structure 27, 31

Interface analysis . . . 4, 12-14, 16, 25, 29, 33, 35, 41
Interface testing . . . 18, 29, 33, 35
Inverted predicates . . . 28
Knowledge-based system (KBS) . . . iii, v-vii, 1, 2, 11, 13, 15-22, 33, 34, 36, 45-50
Logic errors . . . 17, 27, 29, 31, 33, 65
Logical verification . . . 16, 33, 37
Loop invariants . . . 27, 29
Manual simulation . . . 29, 33
Memory allocation . . . 30
Meta models . . . 33, 37
Mismatched parameter lists . . . 28
Missing labels or code . . . 28
Missing validity tests . . . 28
Misuse of variables . . . 26, 28
Modeling . . . 26-28, 30, 48, 49
Mutation analysis . . . 29, 35
Numerical roundoff . . . 26
Numerical stability . . . 26
Omitted functions . . . 26, 30
Parameter checking . . . 26, 32, 33
Partition testing . . . 34, 37
Path testing . . . 27, 28, 31
Performance testing . . . 30, 35
Petri-nets . . . 9, 30, 35
Planning for defaults when system over-stressed . . . 32
Poor programming practices . . . 26
Processing efficiency . . . 26, 27, 31
Prodigal programming . . . 28
Program decomposition . . . 31
Program execution characteristics . . . 28, 31, 32
Proof of correctness . . . 23, 30-32, 35
Proof of critical sections . . . 30
Prototyping . . . 23, 30, 36, 47, 50
Redundancy . . . 26
Regression analysis and testing . . . 16, 24, 30, 36
Reliability . . . v, 1-3, 9, 19, 31, 51-53, 55, 57, 59, 61, 67, 76-79
Reports on test cases that were omitted . . . 32
Requirements parsing . . . 30, 36
Retest after change . . . 27-33
Reuse . . . iii, 2, 9, 11-22, 33, 36, 39, 40, 42, 46, 50, 53, 57, 59
Reviews . . . 2, 6, 7, 23, 24, 29, 31-33, 36, 56, 61, 68
    status reviews . . . 29, 33
    technical reviews . . . 29, 33
Rule verification . . . 34, 37
Safety . . . v, 1, 3, 6-8, 12, 14, 28, 30, 32, 39, 40, 52-54, 66, 79
Security . . . v, 1, 3, 6-9, 12, 40, 66

Sensitivity analysis 31, 36
Simulation 23, 29, 31, 33, 36
Sizing and timing analysis 23-25, 31, 36
Slicing 31, 36, 54
Small, but difficult, or error-prone sections of design or code 33
Software design evaluation 27
Software Design Evaluation 4, 14
Software design V&V 4, 9, 13, 23
Software failure mode, effects, and criticality analysis . . . . . vii, 32
software failure mode, effects, and criticality analysis 9, 36
Software fault tree analysis 9, 32, 36
Software installation test 4, 10, 21
Software integration test 4, 10, 14, 18, 19, 25, 27, 30, 31, 64-66
Software requirements evaluation 4, 11, 31
Software requirements indexing 31
Software requirements to design correlation 31
Software requirements V&V 4, 9-11, 23
Software system test 3, 4, 10, 11, 18-20, 24, 25, 27, 28, 30, 31, 33
Space utilization evaluation 26, 27, 29, 31, 41
Specification error 26
Standards check 31
Statement coverage testing 28, 31
Static analysis 23, 29
Statistical validation 34, 37
Stress testing 32, 36
Structural testing 24, 32, 36
Symbolic execution 32, 36
Synchronization 27, 30
Syntax errors 29, 31, 33
System performance prediction 26-28, 31
Test case adequacy 28, 31
Test case preparation 27, 28, 30
Test certification 32, 36
Test thoroughness 28
Threat analysis 28, 30, 32
Timing 9, 23-28, 30, 31, 35, 36
Turing tests 34, 37
Unauthorized recursion 28
Undeclared variables 28
Uninitialized variables 27-29, 31
Unreachable code 28
Unreferenced labels 28
Unused variables 27-29, 31
User interface 30, 50
Variable references 27, 28, 31
Variable snapshots/tracing 28, 31

Walkthroughs 24, 32, 36
Weight analysis 34, 37