NUREG/CR–6316
SAIC–95/1028
Vol. 2

Guidelines for the Verification and Validation of Expert System Software and Conventional Software

Survey and Assessment of Conventional Software Verification and Validation Methods

Prepared by
L. A. Miller, E. H. Groundwater, J. E. Hayes, S. M. Mirsky

Science Applications International Corporation

Prepared for
U.S. Nuclear Regulatory Commission

and

Electric Power Research Institute

AVAILABILITY NOTICE

Availability of Reference Materials Cited in NRC Publications

Most documents cited in NRC publications will be available from one of the following sources:

1. The NRC Public Document Room, 2120 L Street, NW., Lower Level, Washington, DC 20555-0001

2. The Superintendent of Documents, U.S. Government Printing Office, P. O. Box 37082, Washington, DC 20402-9328

3. The National Technical Information Service, Springfield, VA 22161-0002

Although the listing that follows represents the majority of documents cited in NRC publications, it is not intended to be exhaustive.

Referenced documents available for inspection and copying for a fee from the NRC Public Document Room include NRC correspondence and internal NRC memoranda; NRC bulletins, circulars, information notices, inspection and investigation notices; licensee event reports; vendor reports and correspondence; Commission papers; and applicant and licensee documents and correspondence.

The following documents in the NUREG series are available for purchase from the Government Printing Office: formal NRC staff and contractor reports, NRC-sponsored conference proceedings, international agreement reports, grantee reports, and NRC booklets and brochures. Also available are regulatory guides, NRC regulations in the *Code of Federal Regulations*, and *Nuclear Regulatory Commission Issuances*.

Documents available from the National Technical Information Service include NUREG-series reports and technical reports prepared by other Federal agencies and reports prepared by the Atomic Energy Commission, forerunner agency to the Nuclear Regulatory Commission.

Documents available from public and special technical libraries include all open literature items, such as books, journal articles, and transactions. *Federal Register* notices, Federal and State legislation, and congressional reports can usually be obtained from these libraries.

Documents such as theses, dissertations, foreign reports and translations, and non-NRC conference proceedings are available for purchase from the organization sponsoring the publication cited.

Single copies of NRC draft reports are available free, to the extent of supply, upon written request to the Office of Administration, Distribution and Mail Services Section, U.S. Nuclear Regulatory Commission, Washington, DC 20555-0001.

Copies of industry codes and standards used in a substantive manner in the NRC regulatory process are maintained at the NRC Library, Two White Flint North, 11545 Rockville Pike, Rockville, MD 20852-2738, for use by the public. Codes and standards are usually copyrighted and may be purchased from the originating organization or, if they are American National Standards, from the American National Standards Institute, 1430 Broadway, New York, NY 10018-3308.

DISCLAIMER

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

NUREG/CR–6316
SAIC–95/1028
Vol. 2

Guidelines for the Verification and Validation of Expert System Software and Conventional Software

Survey and Assessment of Conventional Software Verification and Validation Methods

Manuscript Completed: February 1995
Date Published: March 1995

Prepared by
L. A. Miller, E. H. Groundwater, J. E. Hayes, S. M. Mirsky

Science Applications International Corporation
1710 Goodridge Drive
McLean, VA 22102

Prepared for
Division of Systems Technology
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555–0001
NRC Job Code L1530

and

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

ABSTRACT

By means of a literature survey, a comprehensive set of methods was identified for the verification and validation of conventional software. The 153 methods so identified were classified according to their appropriateness for various phases of a developmental life-cycle -- requirements, design, and implementation; the last category was subdivided into two, static testing and dynamic testing methods. The methods were then characterized in terms of eight rating factors, four concerning ease-of-use of the methods and four concerning the methods' power to detect defects. Based on these factors, two measurements were developed to permit quantitative comparisons among methods, a Cost-Benefit Metric and an Effectiveness Metric. The Effectiveness Metric was further refined to provide three different estimates for each method, depending on three classes of needed stringency of V&V (determined by ratings of a system's complexity and required-integrity). Methods were then rank-ordered for each of the three classes in terms of their overall cost-benefits and effectiveness. The applicability was then assessed of each method for the four identified components of knowledge-based and expert systems, as well as the system as a whole.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

EXECUTIVE SUMMARY

In recent years, a large number of expert systems have been developed for use in the nuclear industry. To ensure the reliability and high quality performance of expert systems, the United States Nuclear Regulatory Commission (USNRC) and the Electric Power Research Institute (EPRI) have jointly contracted with Science Applications International Corporation (SAIC) to develop and document guidelines for the Verification and Validation (V&V) of expert systems. This report presents the results of the first of ten activities in this project. The purpose of this first activity is to review software engineering V&V techniques for conventional software systems and to assess their usefulness for expert systems.

This assessment focuses on three technical aspects of conventional V&V techniques: classification, characterization, and assessment. First, conventional methods were classified by a sequential Life-cycle model, i.e., a process of software development and maintenance. Second, the classified techniques were characterized by different factors of power and ease-of-use. Finally, the techniques were assessed according to their applicability to expert systems.

This review resulted in the classification of a total of 153 different conventional software V&V techniques. Based on the sequential life-cycle model, these techniques were divided into two phases: requirements or design, and implementation. The requirements or design phase includes 28 of the techniques while the implementation phase, which includes the categories known as static and dynamic testing, includes the remaining 125 techniques.

Each of the 153 conventional V&V methods was characterized using eight separate factors chosen to permit an assessment of their effectiveness for systems with varying levels of complexity and integrity. These factors are: Broad Power, Hard Power, Formalizability, Human-Computer Interface Testability, Ease of Mastery, Ease of Setup, Ease of Running/Interpretation, and Usage. To determine the extent to which a V&V technique could detect different software defects, a taxonomy of 52 different types of conventional software defects was developed. The techniques were judged as to which of the defects they might detect. Each of the 153 conventional V&V techniques covered anywhere from 2 to 52 of these defects. Each defect was covered by anywhere from 21 to 50 V&V techniques. These findings indicate that conventional V&V techniques, taken together, cover the total space of identified conventional software system defects.

A classification scheme was developed to assign various combinations of complexity and required integrity levels into classes of recommended software V&V. High complexity software systems that may require high integrity are placed in V&V Class 1. Systems with medium levels are placed in Class 2. Systems potentially requiring the least stringent V&V levels are placed in Class 3.

The assessment of the applicability of the conventional V&V techniques to expert systems required the identification of four primary components of expert systems: the inference engine, the knowledge base, external interfaces, and tools and utilities. Each component has several sub-components. These components and sub-components were rated on three factors that relate to V&V: (1) whether conventional programming languages were typically used in their implementation;, (2) whether the components were highly reusable across

applications; and (3) whether the components' potential defects could be identified via formal means. To the extent that conventional languages are used, conventional V&V techniques apply to such components. If the components have high reusability, then certification and bench-marking techniques are recommended as being most appropriate. Formal analysis procedures apply to components whose defects can be fully characterized in terms of specific features. Each of the components was evaluated in terms of the degree to which conventional techniques were applicable for V&V of that component.

With some qualifications, conventional V&V techniques were collectively and individually judged to be highly applicable to the expert systems as a whole and to the following expert systems components: inference engine, the external interfaces, and the tools and utilities. However, current conventional V&V methods were judged to be inadequate in their present form to evaluate the knowledge base component sufficiently, particularly for the more stringent V&V Classes. Nonetheless, conventional methods are judged to be extendable to provide sufficient V&V of the knowledge base.

In conclusion, the immense historical and growing body of conventional V&V techniques and practices are fully usable for expert systems V&V, either directly or by extension. There is no evidence that expert systems are so unique that neither the conventional development and management software processes nor the conventional V&V techniques apply to them. Expert systems are best regarded as special types of software and all conventional practices and software engineering principles fully apply.

1 INTRODUCTION

The United States Nuclear Regulatory Commission (USNRC) and the Electric Power Research Institute (EPRI) are involved in a broad-based evaluation of possible applications of expert systems in the nuclear industry. One of the issues with using expert systems is the lack of an accepted V&V methodology to ensure reliable and high quality performance of this software. The development of appropriate methods for the V&V of expert systems, which is the overall goal of this project, will promote their acceptability in all segments of the nuclear power community.

1.1 Background

Expert systems can be defined as computer software that exhibits human level intelligence and are part of the larger field of artificial intelligence. An expert system can be divided into the following four components: knowledge base, inference engine, interfaces, and tools and utilities. The knowledge base is the component that contains detailed information about the expert system subject matter. This information can be stored in the knowledge base in a variety of forms including "IF-THEN" rules and simple facts. The inference engine uses the knowledge base to make decisions or take actions. Interfaces deal with the connection of the expert system to databases, communication channels, the user, etc. Tools and utilities refers to general application programs that may be used in building the knowledge base or assisting in any other features of the expert system. As compared to expert systems, conventional software typically does not have a knowledge base or inference engine.

V&V refers to two different processes that are used to make sure that computer software reliably performs the functions that it was designed to fulfill. The overall sequence of development and maintenance for computer software is called its Life-cycle. The Life-cycle consists of a number of steps, which may be sequential or iterative, that start with the requirements for the software, V&V of the software, field installation and use, engineering changes, and typically ends years later when the software is retired from use. Verification occurs during software development and checks each stage of development against the version of the previous stage. Validation tests and evaluates the software system to make sure that it correctly performs its intended functions. Verification occurs during the development phases of the software's Life-cycle while validation is performed after software development is completed (see page 9).

Expert systems are also widely used outside the United States and have been extensively applied to non-nuclear industries such as aerospace, geology, telecommunications, and computer manufacturing. Therefore, this project has included information on expert system V&V from foreign and non-nuclear industry sources.

1.2 Objective and Scope

The objective of this project is to develop and document guidelines for the verification and validation of expert systems in the nuclear industry. This project consists of ten activities that will result in a technical report and user manual presenting V&V methods to be used for a wide range of expert systems that may be developed for the nuclear industry.

The first activity, which is the subject of this report, is a detailed survey of currently available V&V methods for conventional software. Existing V&V methods for conventional software are evaluated to find out if they can be used for expert systems. Where necessary, new methods will be developed later to make up for deficiencies in the available software V&V techniques. Applicable current conventional and expert system V&V methods will be tested on two actual nuclear expert systems that were designed for nuclear power applications. Finally, the results will be reviewed and used to develop a set of recommended V&V methods that should be used for different classes of expert systems depending on their importance and complexity. This report documents the results of this first activity.

1.3 Report Organization

This report is divided into nine sections: (1) introduction, (2) context, (3) conventional V&V management, (4) software development Life-cycles, (5) conventional software V&V method classification, (6) conventional software V&V method characterization, (7) applicability of conventional V&V methods to expert systems, (8) summary and conclusions, and (9) references.

This first section introduces the project and activity. The second section lays the groundwork for understanding the subjects of expert systems, V&V, and the overall strategy used for this effort. The third section discusses the importance and benefit of managing software V&V and differentiates V&V from configuration management and quality assurance. The fourth section provides a detailed presentation on the software development Life-cycle that is assumed for this project. Section 5 describes the 153 conventional software V&V methods that were found during this activity and places them into categories that are related to how these techniques are applied to software. Section 6 examines and categorizes software defects and presents a means of measuring how effective each V&V method is in finding these defects. Section 7 analyzes the conventional software V&V methods that were discussed, evaluated and categorized in Sections 5 and 6 and determines which methods can be used for the V&V of expert systems. Section 8 presents a summary of the accomplishments and conclusions from this activity. Section 9 lists the reference documents that were used in this activity.

2 PURPOSE AND CONTEXT

This report concerns the first of ten activities, the overall goal of which is to formulate and document guidelines for verifying and validating (V&V) expert systems[1] for use in the nuclear industry. This work is sponsored jointly by the United States Nuclear Regulatory Commission (USNRC) and the Electric Power Research Institute (EPRI). Both agencies are concerned with the quality and reliability of expert systems used in the nuclear industry. Since 1983, EPRI has sponsored a broad program for applying expert systems to utility needs, tool development support, practical applications, studies on V&V, and recently, a training facility (Naser, 1991).

The USNRC and EPRI are not alone in their concern with the V&V of expert systems applications. The NASA Space Station Freedom project convened a government-industry working group to advise NASA regarding V&V of expert systems associated with the station. Currently, the NASA Johnson Space Center is sponsoring a project to investigate its needs for expert systems V&V. In the Department of Defense (DoD), the Defense Advanced Research Projects Agency (DARPA) has sponsored, through the Air Force's Rome Laboratory, the development of an automated tool to assist in expert systems V&V. The Army Test and Evaluation Command has also sponsored work in this area. The Institute for Electronics and Electrical Engineers (IEEE) and American Institute of Aeronautics and Astronautics (AIAA) have established standards committees for Artificial Intelligence (AI) whose purview includes expert systems V&V.

In the past five years, there has been an increasing number of professional conference sessions and tutorials on expert systems V&V. It should be noted that the leading AI society, the American Association for AI, has sponsored yearly workshops on expert systems V&V since 1987.

These concerns for V&V of expert systems indicate that AI technology is a reliable software-development approach, capable of being integrated with other kinds of software and systems. Expert systems V&V lacks the history of conventional software V&V, but it is further along than V&V of other non-conventional software technologies, including neural network systems, object-oriented systems, and specialized software for parallel processors. This maturity provides a sound basis for the present effort to develop practical and effective V&V guidelines for systems where high standards for quality and safety are paramount.

A general comment on V&V is that it is poorly understood and generally disliked. Adequate V&V is also expensive. Traditionally, it is one of the first activities to be cut when a software project experiences difficulties. The need for V&V, especially for the benefits of reduced system maintenance, is seldom understood by management. Nevertheless, there is considerable agreement among software analysts that V&V will always show a positive cost-avoidance benefit over the life of a system. Careful V&V will do so in some cases even when assessed just for the development stage (see Section 3.3).

---

[1] The terms "expert system" and "knowledge-based systems" are considered to be interchangeable, and the former is used throughout this report for consistency. Expert systems include rule-based implementations, frame-based, and various combinations thereof; the term also extends to hybrid systems which involve rules and frames embedded within an object-oriented approach.

2.1 Purpose of the Survey

The overall objective of the survey is to determine how much of the extensive context and accomplishments of conventional V&V activities can be employed directly for expert systems. More specific objectives are to determine the best conventional techniques to use for the specific components of expert systems as well as for the overall system and how these techniques might need to be modified.

The present survey examines current **conventional software.** These software programs are written in **procedural** programming languages such as FORTRAN, C, COBOL, PL/I, PASCAL, Ada, and ALGOL. In these languages, a single sequential algorithm is specified by the programming language source statements. At any one point in the source program, after execution of the "present" source statement, the next action to be executed by the hardware is found in the next statement in the program unless the next statement is a special transfer-of-control statement (such as an "if", "do", or "case"). In these programs, the data-flow and control-flow operations are also inter-mixed.

There are several alternatives to conventional software. However, software involved in knowledge-based systems or expert systems is of special interest to this report. Systems of this type can be written using an AI-type language, such as LISP (a **functional** AI language) and PROLOG (a **logic-programming** language). More common, and of most direct interest, are those expert systems that have been written utilizing one of the many types of commercial Expert System **shell** products, e.g., EXSYS, NEXPERT OBJECT, ART, KEE, CxPERT, VP-Expert, IQ-2000 (also NASA's CLIPS shell). With these products, the application is mainly written as a set of declarative facts/descriptions and IF-THEN rules that constitute the **knowledge base** of the system. Unlike conventional (procedural) software systems, static inspection of the declarative (non-procedural) knowledge base will not easily or completely reveal the sequence of execution of rules. The actual "execution" of knowledge base elements is fully determinable as a function of the properties of the executing agent that is called the **inference engine**.

Within this context, the purpose of this report is to survey software V&V techniques that have been used to test conventional software systems, and then to assess which of these techniques could be applied to AI systems, specifically expert systems. This survey emphasizes the technical aspects of the techniques rather than the management processes, although both elements are essential. How the survey is influenced by the definition of V&V, by the guidance given concerning standards, and by a number of important aspects concerning the scope of coverage are addressed below.

There is a great advantage in utilizing a wholesale reuse of conventional V&V methods and philosophy for expert systems. Conventional techniques are much more acceptable than novel techniques developed solely for expert systems. Additionally, the acceptability of expert systems software could well be greatly increased. Rather than just being acceptable or unacceptable, there are several possible gradations of applicability of a conventional V&V method to expert systems[2]:

[2] These ratings are applied to techniques in Section 7.

1) The method can be used directly without any modifications;

2) The method largely applies, but some modifications are necessary;

3) The general concept of the method applies, but extensive specific changes are needed; or

4) The method does not really apply at all.

A final objective of this survey is to identify which aspects of expert systems, if any, are poorly addressed by conventional V&V techniques. The Volume 3 follow-on survey of V&V methods will determine whether these needs have been addressed within the expert systems field or whether they must be met by invention in later tasks.

This survey will not provide details of specific methods. It is not intended to be a "how-to" tutorial for actual use. However, it will provide summary information on the relative ease-of-use and effectiveness of the surveyed methods.

2.2 Nature of V&V

A number of slightly different definitions of the terms **Verification** and **Validation** occur in the applicable standards for this study. The following definitions from **IEEE Standard 729-1983** are adopted as being the most widely accepted understanding:

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

The concepts of V&V involve a developmental Life-cycle. These concepts are related to a number of other topics including testing, certification, quality assurance, and configuration management. These topics are briefly defined here to show this relationship.

Testing is performed to demonstrate that the integrated system meets the requirements. Testing involves several activities: test plan development, test execution, and results analysis. The test plan should discuss test requirements, test philosophy, test environments, test specifications, detailed test descriptions, test procedure, and test evaluation approach. Test execution and results analysis includes the performance of validation testing, the recording of test results, and the analysis of results for acceptability (NSAC-39, 1981).

Certification implies a particular kind of formalized testing to demonstrate high capability, usually for particular environments, but it is not a requirement of V&V. Very few software products are actually "certified".

Quality Assurance (QA) is concerned with ensuring that the software product has undergone all of the specified procedures that accompany the design and implementation that are in place to ensure quality. V&V is only one aspect of QA, which includes other elements.

Configuration Management (CM) is an essential procedure for ensuring the quality of a system by controlling the manner in which its components are modified and improved. CM deals with establishing and controlling updates to a baseline version of a system. A characterization of V&V versus QA and CM is given in detail in Section 3.

In summary, the underlying technical concepts of V&V and the related procedures are to ensure that all of the behavioral and performance functions specified in the requirements for the system are fulfilled during development (verification) and in the final implementation (validation).

One objective of V&V is to ensure that the implemented system does not contain unintended functions. An unintended function is described as a function which is not traceable to a specific requirement. Although this principle is not always cited in V&V activity, it should always be considered. The implementation of this additional purpose involves identification and examination of all the residue design or implementation elements after all requirements have been traced.

It is important to probe more closely into the relationship between **testing** and **V&V**. Testing and debugging refer to the analysis, exercise, and repair of software by the developers. V&V, on the other hand, is a process which is independent from normal development and testing activities. It is often called **Independent V&V**, or **IV&V** to indicate that it is accomplished by parties who are not part of the immediate development team. The purpose of V&V is not to directly assist in the development of reliable software but to provide independent evidence of software reliability and that the system performs according to its requirements. In fact, the developers and the V&V agents may use the same discovery techniques, but the V&V agent does so as an independent check on the quality and compliance of the system. Traditionally, V&V agents only discover problems; they do not fix them. Correcting the problem is the province of the development team. However, the V&V agents are more likely to have a deeper understanding of testing methods, test-case construction, and a much better understanding of how to accomplish the complex task of repairing the detected problems completely, generally, and with fewer side-effects[3].

Up to this point, the discussion has been from a technical point of view. However, V&V is an important management process. Management commitment is mandatory to ensure the funding, the staffing, and the cooperation necessary to accomplish the V&V tasks. Detailed management is needed to develop the specific V&V test plans and methods, and the level of effort to be applied. While some of these issues are briefly

---

[3] Re "completely", V&V agents, through experience and training, look for multiple errors near a detected problem. Re "generally", the most in-depth V&V can involve an analysis of the characteristics and probable cause of a problem followed by a search for other instances of that problem-type in other parts of the system. Re "side-effects", the typical V&V agent is acutely aware that problem-fixes often create new problems, and, therefore, adopts a repair strategy aimed at minimizing this effect.

discussed in Section 3, the management aspects, which are critical to actual success of V&V efforts, are not the focus of this survey.[4]

2.3 The Standards Environment

This survey and the whole project recognize the eight key standards and guideline documents, as listed in Table 2.3-1. NSAC-39, NUREG-0653, and NUREG/CR-4640 are currently the primary sources of V&V guidance in the nuclear industry. NSAC-39 provides guidance on how to structure a V&V program for a Safety Parameter Display System (SPDS). Based on a suggested software life-cycle, it outlines V&V activities and documents that should occur at each step. Many expert systems being developed for nuclear power applications are for use by control room operators to here provide an interactive interface for the user(s). Consequently, these systems are very similar in nature to an SPDS, thereby making NSAC-39 especially relevant. Its life-cycle and recommendations form the basis of the description of the conventional software development Life-cycle described in Section 4.

NUREG-0653 discusses software quality assurance requirements for thermal-hydraulic safety analysis software, and its conclusions are considered generally applicable to many types of safety analysis software in the nuclear power industry. As expert systems become incorporated into safety analysis and safety critical software, this standard will eventually become more and more applicable. Therefore, it must be examined for future planning.

NUREG/CR-4640 provides an important mapping of recommended software quality assurance (SQA) practices against the 10 CFR 50 criteria for a complete nuclear quality assurance program. Software quality assurance and V&V go hand-in-hand. System V&V constitute a vital portion of an SQA program. NUREG/CR-4640's suggested Life-cycle diagram is also shown in Section 4. Other sections of this document include descriptions of the documentation required: applicable standards, practices, and conventions; review and audit procedures; software configuration management; V&V; and procurement management. Its recommendations are included in the Section 3 discussion of management aspects of V&V.

Two other standards documents considered to be key are ANSI/IEEE ANS 7-4.3.2-1982 and ANSI/IEEE STD 1012-1986. The first provides criteria for safe practices for design and evaluation of safety performance and reliability. The second defines minimum requirements for the format and content of software V&V plans and for V&V tasks pursuant to those plans. The ASME Code Standard provides guidance for nuclear facility quality assurance including a treatment of Life-cycle, V&V, configuration control, documentation, procurement, and records. The IEC Standard provides guidance for nuclear power plant safety system software.

---

[4] An alternative management approach to development moves V&V into a leading role. It involves a multi-faceted approach. A key element is the use of a developmental Life-cycle involving incremental system-level builds with extensive testing after each build (see Section 4.1.2). The three-part theme of this Life-cycle is "build a little, test a lot, and fix as you go". The keys to this approach are adherence to requirements, design document updating, and continual improvement in the processes of software development by analysis of problems found at each build step -- i.e., a Total Quality Management approach (Miller, 1992).

Table 2.3-1 Key standards and regulations related to V&V of conventional software systems

Document		Document ID	Description
Key Documents	**SPDS**	NSAC/39	Verification and Validation for Safety Parameter Display Systems, December 1981
	QA, Design and Analysis Codes	NUREG-0653	Report on Nuclear Industry Quality Assurance Procedures for Safety Analysis Computer Code Development and Use, August 1980
	QA	NUREG/CR-4640	PNL-5784, Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry, August 1987
		ASME NQA-2a-1990 Part 2.7	Quality Assurance Requirements for Nuclear Facility Computer Software, 1990
	Class 1E Real Time Systems	ANSI/IEEE ANS-7-4.3.2-1982	Application Criteria for Programmable Digital Computer Systems of Nuclear Power Generating Stations, July 6, 1982
		Reg. Guide 1.152	(Task IC 127-5) Criteria for Programmable Digital Computer System Software in Safety-Related System of Nuclear Power Plants, November 1985
	V&V	ANSI/IEEE Std 1012-1986	Software Verification and Validation Plans, 14 November 1986
		IEC 880 1986	Software for computers in the safety systems of nuclear power stations, 1986

The key lessons to be learned from all of these standards are the following:

1) How and why to establish a software development life-cycle,

2) The SQA practices, conventions, and procedures that should be followed at each step,

3) The documentation to be produced/revised at each step,

4) Criteria for testing and validating nuclear power software applications, and

5) Testing techniques.

Detailed discussions on these topics can be found in Section 3 of this report.

In addition to the referenced documents, there are a host of other standards, guidelines, and recommended procedures which relate to the V&V of conventional systems and, thus, to this survey. The total set of related documents are shown in Table 2.3-2. The United States Department of Defense standards DoD-STD-2167 and its replacement DoD-STD-2167A (in the General section in Table 2.3-1) provide the most stringent examples of Life-cycle development and associated reviews and documentation. These standards have been in use for many years. The 1988 modification explicitly acknowledges that iterative prototyping might be required, and that the other aspects might need to be customized for particular types of projects. This standard permits military systems to utilize the iterative or cyclical life-cycle of characteristic of expert systems development.

There are two new draft standards from the United Kingdom, MOD 0055 and 0056 (the QA, CLASS 1E section). These draft standards strongly emphasize the front-end aspects of the system development process, requirements analysis, and design verification. They go much further than previous documents by proposing that **formal proving methods** should be employed for these stages.

2.4 Scope of Survey

The scope of this task is detailed below in the discussion of nine scope factors. In general, the broadest scope was chosen to increase the chances of exhaustive coverage and increase the probability that potentially useful V&V techniques would be gathered.

2.4.1 Management vs. Technical Aspects

This is the exception to the broad scope rule which is stated above. Although good management is essential to achieving the goals of V&V, the technical aspects of V&V are emphasized.

Table 2.3.2 Key standards and regulations related to V&V of conventional software systems

Key Documents	**SPDS**	NSAC/39	Verification and Validation for Safety Parameter Display Systems, December 1981
	QA, Design & Analysis Codes	NUREG-0653	Report on Nuclear Industry Quality Assurance Procedures for Safety Analysis Computer Code Development and Use, August 1980
	QA	NUREG/ CR-4640	PNL-5784, Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry, August 1987
		AMSE NQA-2a-1990 Part 2.7	Quality Assurance Requirements for Nuclear Facility Computer Software
	Class 1E Real Time Systems	ANSI/IEEE ANS-7-4.3.2-1982	Application Criteria for Programmable Digital Computer Systems of Nuclear Power Generating Stations, July 6, 1982
		Reg. Guide 1.152	(Task IC 127-5) Criteria for Programmable Digital Computer System Software in Safety-Related System of Nuclear Power Plants, November 1985
	V&V	ANSI/IEEE Std 1012-1986	Software Verification and Validation Plans, 14 November 1986
		IEC 880 1986	Software for computers in the safety systems of nuclear power stations.
Reference Only	**Design & Analysis Codes**	ANSI/ANS-10.4-1987	Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry, 13 May 1987
		NUREG-0856	Final Technical Position on Documentation of Computer Codes for High-Level Waste Management, June 1983
	QA, Class 1E	ANSI N45.2.11-1974	Quality Assurance Requirements for the Design of Nuclear Power Plants
		NUREG-0493	A Defense-In-Depth and Diversity Assessment of the Resar-414 Integrated Protection System, March 1979
		10 CRF 50	Code of Federal Regulations, 1 January 1984
		UK Draft Defense Standard 0055	UK MOD Interim Standard on Requirement for the Procurement of Safety Critical Software in Defense Equipment
		UK Draft Defense Standard 0056	UK MOD Interim Standard on requirements for the Analysis of Safety Critical Hazards

Table 2.3.2 (Continued)

		EWICS TC7	Critical Computer Systems (Safety Assessment)
	Computer Society ANSI/IEEE Stds.	Std. 729-1983	Glossary of Software Engineering Terminology, 18 February 1982
		Std. 829-1983	Standard for Software Test Documentation, 18 February 1983
		Std. 982.1	Standard for Measures for Reliable Software
		Std. 982.2	Guide for Measures for Reliable Software
		Std. 983-1986	Software Quality Assurance Planning, 13 January 1986
Reference Only (cont.)	**Computer Society ANSI/ IEEE Stds. (cont.)**	Std. 990	IEEE Recommended Practice for Ada as Programming Design Language
		Std. 1008-1987	Code of Federal Regulations, 1 January 1984
		Std. 1002-1987	Software Unit Testing, 29 December 1986
		Std. 1028	Standard for Software Reviews and Audits
		Std. 1042-1987	Guide to Software Configuration Management, 12 September 1988
		Std. 1044	Standard for Classification of Software Errors, Faults and Failures
		Std. 1045	Standard for Software Productivity Metrics
	IEEE Stds. (Gen'l)	Std. 730-1984	Software Quality Assurance Plans, 30 June 1984
		Std. 828-1983	Software Configuration Management Plans, 24 June 1983
		Std. 830-1984	Software Requirements Specifications, 10 February 1984
		Std. 1016-1987	Recommended Practice for Software Design Descriptions, 13 July 1987
		Std. 1058.1-1987	Software Project Management Plans, 31 August 1988
		Std. 1063-1987	Software User Documentation, 22 August 1988
	General	ANSI MC8.1-1975	Hardware Testing of Digital Process Computers, October 1971

Table 2.3.2 (Continued)

		ANSI N413-1974	Guidelines for the Documentation of Digital Computer Programs, 20 June 197
		NSAC/5	Computer Systems Interface Guidelines for Nuclear Plants September 1980
		INPO TS-407	Good Practice Computer Software Administrative Controls (Draft) 1983
		DOD-STD-2167	Defense System Software Development, 4 June 1985
		DOD-STD-2167A	Defense System Software Development, 29 February 1988
		DOD-STD-2168	Software Quality Evaluation (Draft) 26 April 1985
		NUREG/CR-2186	ANL-81-84, Quantitative Software Reliability Analysis of Computer Codes Relevant to Nuclear Safety, December 1981
	SPDS	NSAC/40	Accident Sequences for Design, Validation and Training-SPDS-April 1982
		NUREG-0696	Functional Criteria for Emergency Response Facilities, Final Report, February 1981
		NUREG-0700	Guidelines for Control Room Design Reviews, September 1981
		NUREG-0737	Supplement 1, Clarification of TMI Action Plan Requirements, January 1983
Reference Only (cont.)	**SPDS (cont.)**	NUREG-0800	Standard Review Plan, Rev. 1, (formerly NUREG-75/087)
		NUREG-1342	A Status Report Regarding Industry Implementation of Safety Parameter Display Systems, April 1989
	EOPs	NUREG-0899	Guidelines for the Preparation of Emergency Operating Procedures, Resolution of Comments on NUREG-0799, August 1982
		NUREG/CR-3177	EGG-2243, Vol. 1, Methods for Review and Evaluation of Emergency Procedure Guidelines, Volume 1: Methodologies, March 1983
	FIPS PUB (Gen'l)	FIPS PUB 105	Guidelines for Software Documentation Management, 6 June 1984

Table 2.3.2 (Continued)

		FIPS Pub 30	Software Summary for Describing Computer Programs and Automated Data Systems, 30 June 1974
		FIPS PUB 38	Guidelines for Documentation of Computer Programs and Automated Data Systems, 15 February 1976
		FIPS PUB 106	Guidelines for Software Maintenance, 15 June 1984
	FIPS PUB (V&V)	FIPS PUB 132	Guideline for Software Verification and Validation Plans, 19 November 1987
		FIPS PUB 101	Guideline for Lifecycle Validation, Verification and Testing of Computer Software, 6 June 1983
	V&V	EPRI NP-5236	Approaches to the Verification and Validation of Expert Systems for Nuclear Power Plants (1987)
		EPRI NP-5978	Verification and Validation of Expert Systems for Nuclear Power Plant Applications
		NBS 500-56	Validation, Verification, and Testing for the Individual Programmer, February 1980
		NBS 500-75	Validation, Verification, Testing of Computer Software, February 1981
		NBS 500-93	Software Validation, Verification, and Testing Technique and Tool Reference Guide, September 1982
		NBS 500-98	Planning for Software Validation, Verification and Testing, November 1982
		NBSIR 82-2482	A Survey of Software Validation, Verification, and Testing Standards and Practices at Selected Sites, April 1982
	NBS (Gen'l)	NBS 500-73	Computer Model Documentation Guide, January 1981
		NBS 500-87	Management Guide for Software Documentation
		NBS 500-106	Guidance on Software Maintenance, December 1983 (General)

2.4.2 System Complexity

Of particular concern from the point of view of V&V are the characteristics of software systems that define its **complexity**. These factors make the system harder to develop and analyze. Generally, the higher the complexity, the greater the opportunity for errors and the greater the need for V&V. Six complexity factors for software systems, with three levels each, are identified in Table 2.4.2-1. These are more general than Boehm's focused description of detailed factors of module complexity (Boehm, 1981, p. 391, Table 6). They are also more descriptive of all software systems than Hayes-Roth's description of levels of architectural complexity in expert systems (Hayes-Roth, 1983, p. 22). The first complexity factor, **physical control capability**, concerns whether the system can control aspects of its environment directly; those that can are more difficult to validate. The lowest level of this complexity factor has no relation at all to control actions. The medium level has no direct control function but provides advisory or decision data for control decisions. The high level directly involves control of system elements.

The second factor, **processing**, has six sub-features associated with it, concerning: real-time aspects, number of processors, whether they are sequential or parallel, synchronous or a synchronous, centralized or distributed, and batch or interactive. The low level is the simplest on all of these features. The medium and high levels are much more complicated, with the high level having the extreme values.

Interactivity with other systems is the third complexity factor and has four sub-features: stand-alone vs. attached or embedded, number and type of interfaces, data- vs. user-driven, and whether there is interrupt-handling.

The fourth factor concerns **Knowledge and Data structures**, with three sub-features: whether or not the information is homogeneous in structure and type, whether the information is in one central place or distributed, and whether the information is easily derived from well-structured codified sources, has to be extracted from experts, or has to be invented.

The type of **decision procedure**, factor five, deals with four sub-features: type of chaining, type of search (although this is not a discriminant among levels), whether reasoning is monotonic or non-monotonic, and then a variety of types of specialized types of reasoning. The last factor details the extent to which the system has **uncertainty handling** features. These two factors, decision procedure and uncertainty handling, are admittedly more characteristic of expert systems than conventional ones. However, these factors do significantly influence complexity and they both could be implemented in conventional programming languages.

The average complexity level across the range of all existing conventional systems would probably be low to low-medium. However, new software is tending much more to the higher-medium and even high complexity levels. This is particularly true when the software is in the form of expert systems. A recent survey of almost 300 expert systems in the nuclear industry revealed quite a number that would definitely be of medium complexity. To the extent that program managers can be assured of the quality and reliability of the programs, such as through reliance on the guidelines to be developed in this project, one can expect that more and more expert systems will have higher and higher complexity characteristics. One example of a primary candidate for implementation with major reliance on expert systems is the SPDS (Safety Parameter Display System). This

Table 2.4.2-1 Six factors of software system complexity

COMPLEXITY FACTOR	COMPLEXITY LEVEL		
	LOW	MEDIUM	HIGH
Physical Control Capability	• None • Advisory function only	• No direct, but can provide decision data into control modules	• Can directly manipulate and control system elements
Processing	Not real time • Sequential • Single Processor • Synchronous • Centralized • Batch/Interactive	• Near or full Real-Time • Multiple processors • Central/Distributed • Interactive	• Real-Time • Concurrent/multiple, heterogeneous processors • Highly distributed • Cooperating • Asynchronous • Interactive
Interactivity with Other Systems	• Stand-alone • Single user-interface • No data-interfaces • User-driven • No interrupt handling	• Embedded/Attached • Continuous/Intermittent Data-Input • Usually Data-driven • Possible Interrupt-handling	• Embedded • Continuous Data-Input, Multiple channels • Usually Data Driven • Possible Interrupt-handling
Knowledge/Data Structures and Storage	• Homogeneous • Centralized • Derived from codified sources	• Homogeneous/Heterogenous • Centralized/Distributed • Derived from codified sources and experts	• Heterogeneous • Centralized/Distributed • Derived from codified sources, experts, or invented
Decision Procedure	• Backward (top-down) or • Forward (bottom-up) Chaining • Breadth first or Depth first • Monotonic Reasoning	• Backward, Forward, and mixed chaining • Breadth first or Depth first • Monotonic or Non-Monotonic reasoning • Heuristic Reasoning • Constraint-based reasoning • Belief-revision, truth maintenance	• All types of chaining • Breadth first or Depth first • Monotonic or Non-monotonic reasoning • Model-based inferencing, plus all other types
Uncertainty Handling	• None	• Fuzzy Reasoning, • Reasoning under uncertainty	• Complex Fuzzy and uncertainty reasoning, • Multiple-Hypothesis evaluation (e.g., Bayesian)

high-medium to high complexity system must detect events occurring in real-time data channels as well as respond to process interrupts, run concurrently and asynchronously with other processes, and perform complex decision procedures under sometimes uncertain conditions.

2.4.3 Definition of Systems in Terms of V&V Classes

The **complexity** of systems, as described in Section 2.4.2, is an important factor in determining the amount of V&V needed for a system to ensure its reliability and compliance with requirements. However, there is another factor which needs to be considered in determining the extent of estimated V&V required: **system integrity**. This factor refers to the joint capability of a system to operate for long periods without failures, to fail gracefully with reasonable warnings, to be able to recover rapidly without much difficulty, and to avoid causing expensive damage to property or harm to people or the environment. High integrity systems rarely fail. They do so very safely and economically. Additionally, they are easy to fix and easy to restart. However, low integrity systems are deficient in one or more of these aspects. How much integrity is required of a system will be a function of several factors. These factors tend to be independent of the factors that make up complexity. Thus, a highly complex system with a low degree of required system integrity should probably not need as much V&V as a highly complex system with a very high degree of required system integrity. These two factors of complexity and required system integrity are best thought of as continuous underlying dimensions along which various points can be identified.

Table 2.4.3-1 shows three points on the complexity dimension coordinated with three points of the degree of required system integrity. The factors supporting each of the three complexity points are written in the first column. These factors represent conditions that define complexity as very high, moderately high, and low. Factors underlying the points on the required integrity dimension are not identified, as this is a very complex judgment which will differ greatly from one site and situation to another. The required integrity points are simply entitled as low, medium, and high, however this might be determined (e.g., from the point of view of system operability, safety, mission capability).

The intersection of the three points on the two dimensions creates nine cell combinations. Again, these are selected combinations out of a much larger potential set of combinations of values from the two dimensions. Nevertheless, the combinations do represent practical and significant situations, and names existing expert system application examples which are entered in each of the nine numbered cells of this 3 x 3 table.

A representation like that in Table 2.4.3-1 makes it feasible to envision the two independent variables of complexity and required integrity combining to reasonably determine the level of V&V that should be applied to a particular system. The cell at the highest values of complexity and required integrity, cell number 3, should receive the most stringent application of V&V methods. The most extensive and thorough methods would be used for this situation. Similarly, the lowest complexity and required integrity cell, cell number 7, should receive only the minimum degree of V&V. How the cells are to be assigned is suggested below, but the responsibility ultimately rests with the individual agency utilizing the chart.

Inspection of Table 2.4.3-1 will reveal that the cells are grouped into three classes: the upper right cell, number 3, is the lone member of V&V Class 1, the most stringent class; the bottom left-most two cells, cells 7

Table 2.4.3-1 Three levels of V&V stringency used in this report for expert system software in the nuclear power industry [1]

System Complexity	Degree of Required System Integrity		
	Low	Medium	High
	INTEGRITY DIMENSION		
Quite High Embedded, Real-time, Continuous Data-Input channels, Direct Control Functions, May have Interrupt Processing	1 Steam Generator Blowdown Control system Radioactive waste management	2 Automatic control-rod manipulation Main Feedwater Control System	3 Reactor Protection System ***V&V CLASS 1***
Moderately High Embedded or Attached No Direct Control functions, Control-Decision Support function, At least near real-time Continuous Data Input Channels	4 TPA (Thermal Plant Analyzer) Turbine Generator Diagnositc Monitoring	5 EOPTS (Emergency Operating Procedure Tracking System) RSAS (Reactor Safety Assessment System) REALM (Reactor Emergency Action Level Monitor) ***V&V CLASS 2***	6 ECCS (Emergency Core Cooling System) Real-time Monitoring and Diagnosis
LOW Stand-Alone, User Driven, Non Real-time, Advisory Functions, No continuous Data Input	7 Fuel-Rod Reshuffling Planner Water Chemistry Advisor	8 SARA (Safety Review Advisor) Plant-Layout ***V&V CLASS 3***	9 In-service ECCS (Emergency Core Cooling System) inspection Advisor ESAS (Emergency Safety Actuation System) Testing System

COMPLEXITY DIMENSION

[1] Candidate systems for V&V are to be matched to the closest cell examples to determine suggested V&V class

and 8, constitute V&V Class 3; and the remaining six cells are in V&V Class 2 (cells 1, 2, 4, 5, 6, and 9). The meaning of the classes is as follows: Class 1 receives the most stringent level of V&V; Class 2 receives a substantial degree of V&V, and both it and Class 1 receive considerably more stringent V&V than the Class 3 cells, which receive the minimum. The concept of increasing stringency implies several things: (1) a greater thoroughness in testing system aspects represented by more test cases per function tested; (2) a greater completeness in testing coverage of the system represented by more functions and/or program structures being tested; and (3) a greater effort to discover truly hard or subtle faults. The actual methods to be applied in the three classes are not yet specified here. That will be accomplished near the end of this project. However, the use of the complexity and required integrity dimensions, the selection of the three points on each, the specification of examples, and, most importantly, the assignment of cells to Table 2.4.3-1 is presented.

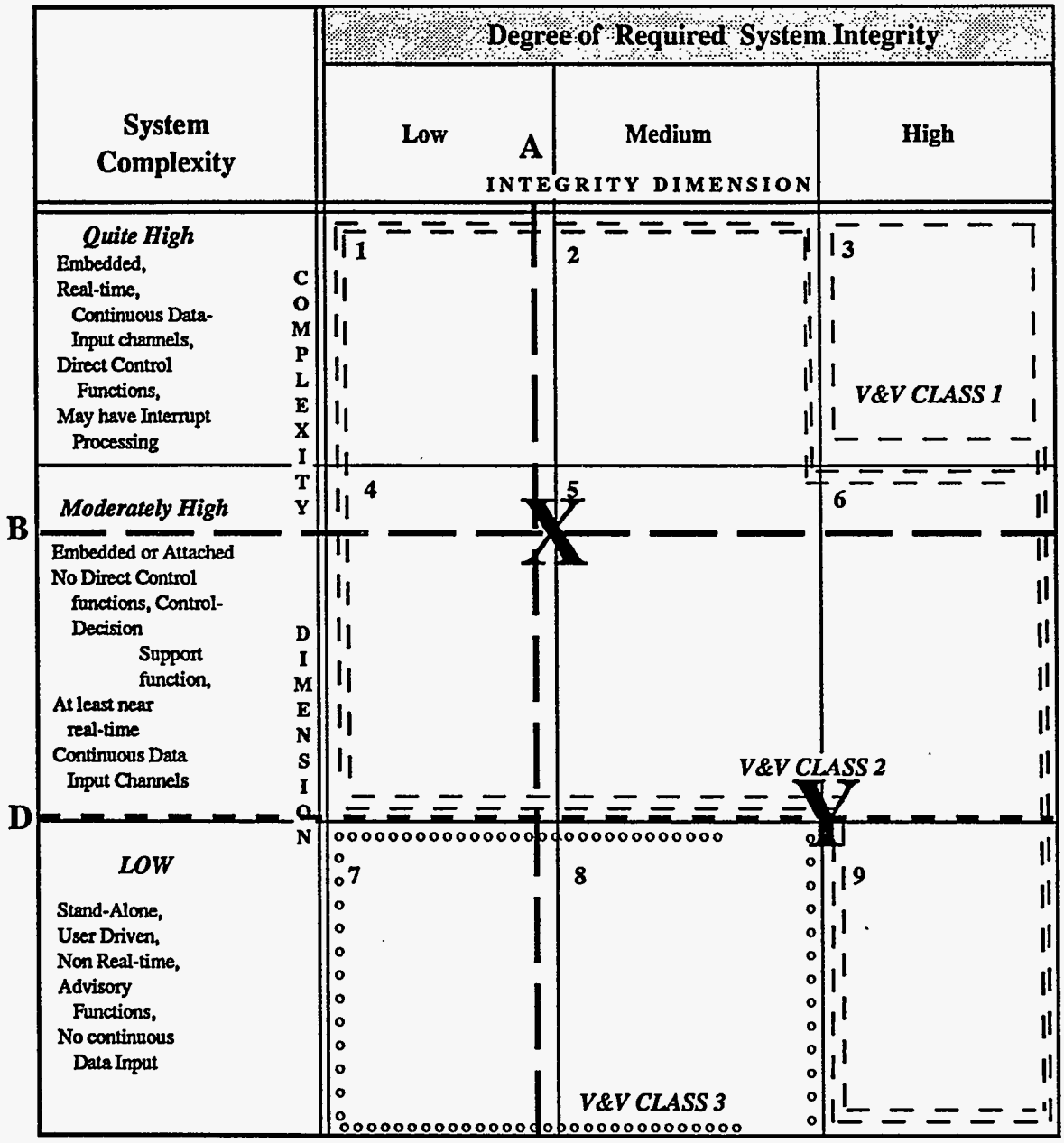
In practice, anyone planning to develop an expert system, or to V&V an existing expert system, should first consider the requirements of the system and decide by appropriate means the required integrity of the system. Required integrity need not be exactly "low", "medium", or "high". It may have intermediate values. The location of this judgment should be marked with a point on the line labeled "required integrity dimension". Additionally, the person should evaluate the complexity of the software system in terms of the complexity factors represented in the table and then locate the judged complexity with a point on the line labeled "complexity dimension".

Two examples of using Table 2.4.3-1 to determine the V&V Class of an expert system as was done in this investigation are shown in Table 2.4.3-2. In the first example, the integrity and complexity values of the system are shown by points **A** and **B**, respectively. The intersection of these values is marked by point **X**, in cell 4. This point is in the low-integrity/moderate-complexity cell, but close to the medium-integrity border, so one needs to look at the adjacent cell, cell 5. But cell 5 is also in V&V Class 2, so even if point A were actually to move towards increased integrity a bit, it would not change the recommended V&V class. The second example, with points **C** and **D**, locates the second system near a corner of cell 6. Here three adjacent cells (5, 8, and 9) may be considered. Cell 5 and cell 9 are in the same V&V Class as cell 6, which contains Y. So, the only consideration is whether Y actually should be located closer to, or in, cell 8, which is in the Class 3 region. If one re-evaluates the required integrity and complexity of the candidate expert system and determines that indeed the candidate system is somewhere between cell 6 (or 5 or 9) and cell 8, then the degree of closeness should be assessed (in terms of closeness of 8 to 5/6/9 on a 10-point scale), and the appropriate supplemental V&V techniques to move between Class 3 and Class 2 may be used as prescribed.

2.4.4 System Components

The scope of concern encompasses the overall software system and its components (Table 2.4.4-1). Highly structured conventional software systems (particularly military ones) are often composed of major subsystems with separate configuration management baselines. These baselines contain **computer software configuration items** (CSCIs) as defined in United States Department of Defense systems developed under DoD-STD 2167 or 2167A. Each separate subsystem may have a number of components defined as **computer software components** (CSCs) and each component may have a number of small sub-components, or **modules**.

Table 2.4.3-2 Illustration of Use of the table for two candidate expert systems with estimated integrity/complexity values given by points (A,B) and (C,D) respectively.



Some V&V techniques, particularly dynamic testing ones, are appropriate for modules but not for overall system or subsystems. All of the techniques for all of the components are of interest and value.

Anticipating the discussion in Section 7 of components of expert systems, the distinctions made in Table 2.4.4-1 among software "components", module, subsystem, and system, are really only the differences among small to very large programs. There is nothing in this characterization which reflects anything about the **function** of the component. If the topic of discussion were well-defined types of program applications which accomplished specific functions (such as compilers, database management systems (DBMS), spread-sheets), then it would be possible to identify specific **functional components**.

For example, a DBMS will have specialized components such as "input-query processor", "database access mechanism", and "transaction roll-back module". It is this second sense of functional components that will be used later to characterize expert systems.

2.4.5 Nuclear vs. Non-nuclear Applications

The whole range of V&V techniques from all application areas have been considered; this report includes nuclear and non-nuclear applications. For example, a number of methods were developed for military or space applications, because of need for high integrity and reliability in software. These could easily be applied to nuclear and other types of applications.

2.4.6 United States vs. Foreign

Although the eventual guidelines will be intended primarily for the United States nuclear industry, it was important to extend consideration of V&V techniques to methods developed elsewhere. The European Community was of particular interest due to its great concern for high-integrity systems. Examples of foreign activities which are considered very important in the review are the draft standards of the UK concerning use of formal methods for systems (UK, 1989), the National Science Foundation-sponsored conference and report on Nuclear Instrumentation and Controls and their associated software tools (e.g., Beltracchi, 1991), and the testing and tool-development activity of the Halden project in Norway (e.g., Dahll, 1990).

2.4.7 Evaluation Criteria

Systems can be evaluated with respect to a wide variety of criteria. Table 2.4.7-1 shows a perspective of 68 criteria grouped into three classes: criteria related to the specific functionality of the system, criteria related to performance, and general non-performance attributes.

An alternative view is given in Figure 2.4.7-1 which groups some of these evaluation criteria into three classes of acquisition concern: performance, design, and adaptability. These three classes are composed of a total of 11 major criteria classes and 21 subfactors; the definitions of the subfactors are given in Table 2.4.7-2. Figure 2.4.7-1 and Table 2.4.7-2 are transformations of Sizemore's 1990 representation, pp. 1-6.

Table 2.4.4-1 Components of larger conventional software systems

Aspects	Level		
	Module	Subsystem	System
Development Environment	Programmer	Multiple Programming Teams	Multiple Contractors/ Vendors
Expected Range of Developers	5	5-40	40-500
Complexity of Integration	Minimal/ Other Modules	Somewhat Complex	Very Complex
Testing Precedence	Early	Post Module	Last
Size	$\leq$ 100 Lines Of Code	Many Modules	2 or More Subsystems

Table 2.4.7-1 Criteria to be tested or evaluated for three major classes of requirements

Class	Criteria	Class	Criteria
Functionality	accuracy consistency completeness coverage correctness explanation feasibility help meta-knowledge operational concept tutoring	**General Attributes (cont.)**	data standardization device independence documentation adequacy error handling error recovery error-tolerance expandability extendibility fault tolerance flexibility human engineering instrumentation integrity interoperability learning machine independence maintainability modifiability modularity operability portability readability recoverability reliability reusability robustness safety security self-containedness self-descriptiveness simplicity structuredness testability traceability understandability usability
Performance	database access time execution efficiency "guaranteed adequate answer" I/O handling memory requirements number of solutions power (re: human-time to solution, quality, success, experience level) storage efficiency time to solution		
General Attributes	access auditability access control accountability auditability augmentability availability communicativeness communications commonality communication standardization conciseness cost data commonality		

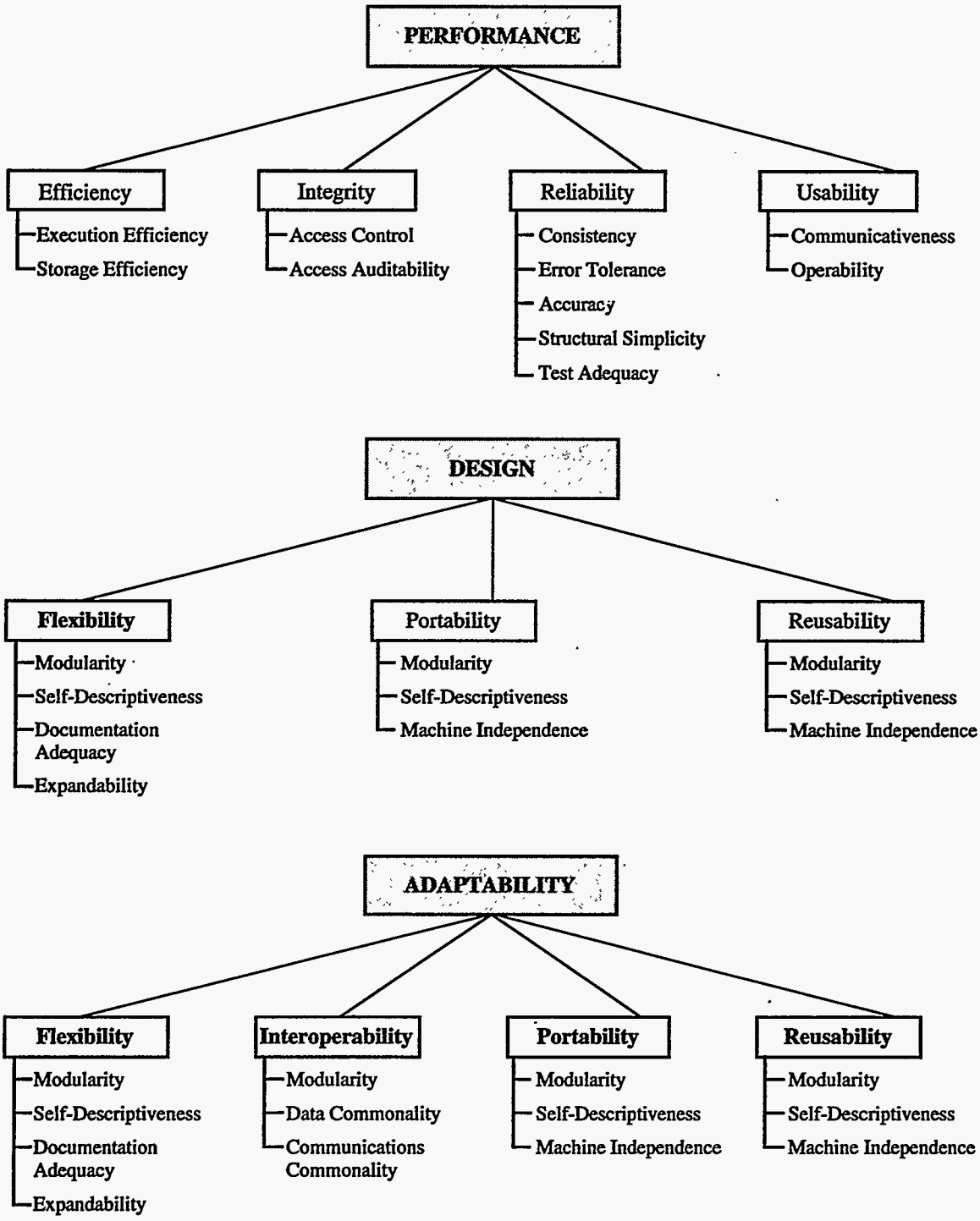


Figure 2.4.7-1 Three Major Acquisition concerns (Performance, Design, andAdaptability) with their 11 Major Performance Factors and 21 Subfactors (based on Sizemore, 1990)

Table 2.4.7-2 Definition of software quality subfactors (adapted from Sizemore, 1990)

Traceability	The extent to which the products of each software development phase implement the products that precede them, have their basis in those products, and provide mechanisms that aid in establishing those ties.
Execution Efficiency	The extent to which a system performs its intended functions with minimum execution time.
Self Descriptiveness	The extent to which program source code is easy to read and understand.
Completeness	The extent to which a system contains all required components and each of those components is fully developed.
Storage Efficiency	The extent to which a system performs its intended functions with minimum consumption of storage resources.
Documentation Adequacy	The extent to which required documentation exists, is in the required formats, and is accurate, clear and complete.
Consistency	The extent to which a system's code and documentation are uniform and free of contradiction.
Access Control	The extent to which a system provides mechanisms to control access to software and data.
Instrumentation	The extent to which a system contains instructions or assertions to facilitate execution monitoring, debugging, and testing.
Error Tolerance	The extent to which a system continues to operate correctly despite input errors or software faults.
Access Auditability	The extent to which a system provides mechanisms to audit the accessing of software and data.
Expandability	The extent to which a system can be easily modified to provide additional functions or data storage capacity.
Accuracy	The extent to which a system is free from error in calculations and output.
Communica-tiveness	The extent to which a system provides useful output and an interface with the user.
Machine Independence	The extent to which a system can be made to execute in more than one hardware or software environment.
Test Adequacy	The extent to which test planning and execution ensure thorough testing of the system.
Operability	The extent to which a system can be loaded, initiated, executed and terminated.
Data Commonality	The extent to which a system uses standard or common data formats, types, representations and structuring.
Structural Simplicity	The extent to which a system is free from complicated data, logical and control structures.

Table 2.4.7-2 (Continued).

Modularity	The extent to which a system is composed of discrete components such that a change to one component has a minimal impact on other components and such that the test performed by single component are functionally related.
Communications Commonality	The extent to which a system uses standard or common communication protocols and interface routines.

All of these factors are of interest. Their relative importance should be specified in the system's Requirements Document. For nuclear industry applications the most critical evaluation criterion should usually be **safety**. The next most important criteria might be the human factors criterion of **operational concept.** This involves the human-computer interface, particularly the style of information display and the allocation of analyses and decisions to user and computer.

2.4.8 Phase in the Life-cycle

The scope of concern encompasses the overall Life-cycle, as well as Life-cycle phases. Some V&V techniques are appropriate for early Life-cycle phases (such as requirements analysis and traceability), whereas other techniques are appropriate only after system implementation (such as dynamic testing).

2.4.9 Other

There are at least four other factors to consider with respect to scope: automated tools and methods, alternative development or testing environments, programming languages, and type of software defects and problems. All aspects of these factors are of interest as they apply to V&V of all software.

2.5 Approach

This survey was conducted in three major stages (corresponding to the three major Sections, 5-7, of this report):

1) Classification of conventional V&V methods (Section 5)
2) Characterization of the methods (Section 6), and
3) Assessment of their applicability to expert systems (Section 7).

The initial activity of classification involved three steps. First, a number of publications which dealt with V&V and software quality assurance topics were reviewed. These included textbooks, journals, proceedings, and
communications containing V&V articles. A literature search was also performed on V&V topics and copies of the most relevant reports were obtained. Over 300 V&V related documents were consulted.

The next step in classification involved sorting the techniques according to the phases of system development they addressed. The phases considered were: Requirements, Design, Implementation, and Maintenance. However, there were no V&V techniques especially designed for this last category although maintenance needs are quite different than those of development (Miller, 1978). Methods used for the Requirements phase were often used for the Design phase, therefore, the final mapping of V&V techniques to Life-cycle involved only two phases: a combined Requirements/Design phase and an Implementation phase.

The third classification step involved grouping the methods within each Life-cycle phase into natural categories, based on similarities among the methods. This classification stage is illustrated in Figure 2.5-1.

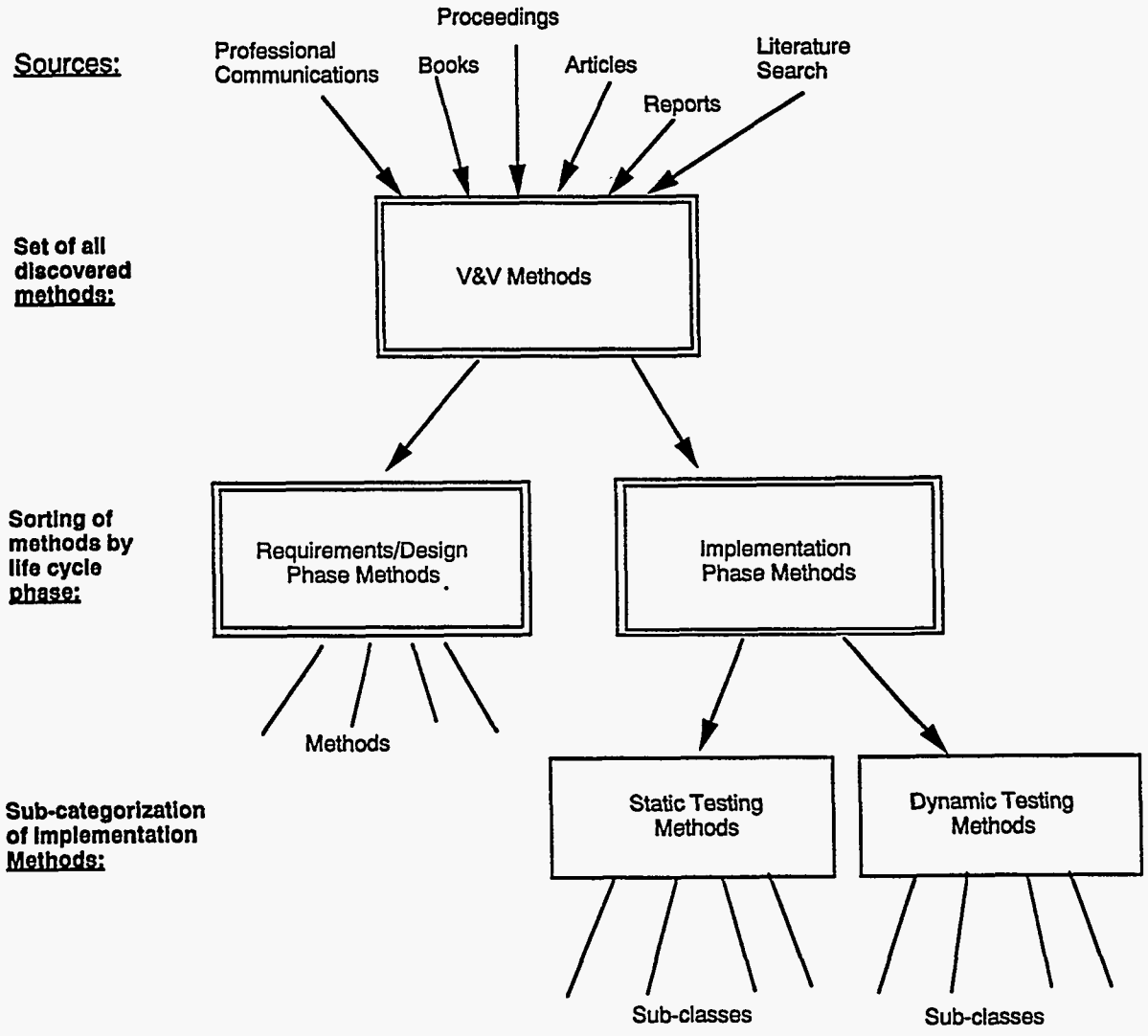


Figure 2.5-1 Survey classification of discovered V&V Methods first by life-cycle phase (Requirements/Design and Implementation) and then by natural categories.

The second major stage of the survey was to characterize the individual V&V methods in terms of the defects they could detect, their effectiveness in doing so, and the various aspects of using these methods. To do this, a taxonomy of software defects was first developed, and then an assessment was made concerning which defects could be detected by which methods. A number of additional characterization features were developed and each method was ranked on all features. Schemes were developed for computing a cost-benefit measure as well as an "effectiveness metric" which provided three scores for each technique, one for each of the three V&V classes identified in Section 2.4.4.

The third and last survey stage was to assess the applicability of these conventional V&V methods for expert systems. To do so, the four major components of expert systems were identified. These components were then characterized in terms of potential defects and similarity to conventional software. Finally, conventional methods were assessed as to their applicability to overall expert systems and their components. The most effective methods were identified.

3 MANAGEMENT ASPECTS OF CONVENTIONAL V&V

Expert systems are primarily software systems; software is software, and software must be managed. Management of software systems is the focus of this section, which details the importance of the software development Life-cycle. According to Boehm's 1976 model, the software development Life-cycle consists of system requirements, software requirements, preliminary design, detailed design, code & debug, test and preparations, and operations and maintenance. Management of the software development Life-cycle is a key process. The Life-cycle drives all components: reviews, quality assurance (QA), testing, documentation, etc.

While ideal Life-cycles for conventional systems and expert systems will be different, the appropriate management techniques will not differ greatly. No system can be developed to the exacting standards of the nuclear industry (or any other industry with safety and reliability concerns) without adequate and thorough management.

Although this effort focuses on the technical aspects of V&V, this section expands on the three major aspects of V&V management. This expansion focuses on the reviews and documentation required by the V&V process throughout the Life-cycle, V&V's association with code-control and quality assurance, and the cost-savings recouped by finding defects early in the life-cycle.

3.1 V&V Documents, Procedures, and Reviews

The minimum set of required documents for a V&V program of any size or type is a Software Requirements Specification and a Software V&V Plan. The Software Requirements Specification is the basis for assessing the correct functioning of the software. The Software V&V Plan details the methodology of how the V&V process will be managed. The project documentation will be used to inform the reader of the project's progress and to trace requirements to specifications, design elements, code modules and tests. Documentation must be clear, concise, and of high quality. It also must be maintained and updated in conjunction with the progress of development on the software system itself.

The most important V&V review is the Software Verification Review which evaluates the Software V&V Plan. This review ensures that V&V is considered and planned at the beginning of the project. It will cover the software development Life-cycle and identify how V&V activities are incorporated. Special constraints or concerns are readily identified during the review process. The plan should be developed incrementally and include revisions and expansions at each step of the Life-cycle. It is possible that multiple reviews may be necessary. Instituting corrective procedures as a result of the reviews should be done carefully. Milestone reviews such as the Preliminary Design Review (PDR) and the Critical Design Review (CDR) are discussed in Section 4.

NUREG/CR-4640 provides a table mapping Software Quality Assurance requirements to 10 CFR 50 Appendix B criteria. This represents a complete nuclear quality assurance program and is reproduced here as Table 3.1-1. The table shows which Appendix B criteria apply to which chapters of NUREG/CR-4640. Nuclear power applications software can be used in the design, analysis or operation of safety-related structures, systems, or components, and it must be included under the regulations of 10 CFR 50.

Table 3.1-1 Correspondence between software quality a assurance requirements (SQA) and appendix B criteria from 10 CFR 50

Report Chapter	Appendix B Criteria
3.0 Software Life Cycle	II. Quality Assurance Program III. Design Control X. Inspection
4.0 Management	I. Organization II. Quality Assurance Program
5.0 Documentation	II. Quality Assurance Program III. Design Control IV. Procurement Document Control V. Instructions, Procedures, and Drawings VI. Document Control XVII. Quality Assurance Records
6.0 Standards, Practices, and Conventions	II. Quality Assurance Program III. Design Control
7.0 Review, Audits, and Controls	I. Organization II. Quality Assurance Program III. Design Control V. Instructions, Procedures, and Drawings VI. Document Control VIII. Identification and Control of Materials, Parts, and Components X. Inspection XVIII. Audits
8.0 Tools and Techniques	III. Design Control IX. Control of Special Processes
9.0 Software Configuration Management and Code Control	I. Organization II. Quality Assurance Program V. Instructions, Procedures, and Drawings VI. Document Control VII. Control of Purchased Material, Equipment, and Services VIII. Identification and Control of Materials, Parts, and components VIII. Handling, Storage and Shipping XIV. Inspection, Test, and Operating Status XV. Nonconforming Materials, Parts, and Components XVI. Corrective Action XVII. Quality Assurance Records
10.0 Verification and Training	II. Quality Assurance Program III. Design Control X. Inspection XI. Test Control
11.0 Control of Software Procurement	I. Organization II. Quality Assurance Program III. Design Control IV. Procurement Document Control VIII. Control of Purchase Material, Equipment, and Services

3.2 Contrast of V&V with QA & CM

The success of V&V activities is greatly dependent on the availability and quality of the software documentation. Therefore, a well defined and effective software quality assurance program is necessary to obtain the maximum benefit from a V&V program. The Software Quality Assurance Plan addresses the following: (1) software Life-cycle definition, (2) software documentation requirements, (3) software development standards, practices, and conventions, (4) V&V requirements, (5) configuration management requirements, (6) quality assurance (QA) reviews and audits, and (7) testing requirements.

A key requirement of both successful QA and V&V is that these activities be accomplished by independent agents. Independent agents are defined as individuals who are not part of the system's development team. They may be from a separate department with the same development organization or from an outside development organization altogether. This is to separate the QA and V&V agents from the day-to-day, mundane details of the development effort. This separation gives them the ability to see the overall picture clearly and not be influenced by the undocumented assumptions of the development personnel, providing an unbiased assessment of the development team's activities. This is not to suggest that QA and V&V as processes should be independent of the development activity. On the contrary, the more closely they are integrated into every aspect of development, the better.

In addition to QA, software configuration management (CM) is required for the successful application of V&V techniques. CM formally documents all changes to development documentation and code. Changes are controlled through a change request and approval process to ensure that only appropriate and approved changes are made. This process allows the verification of documentation and software changes with some assurance that other unauthorized changes were not made. Without the implementation of change control, the total product would require re-verification to detect inappropriate modifications or the introduction of new errors.
CM activities consist of configuration identification, configuration change control, configuration status accounting and reporting, and configuration audits and reviews. ANSI/IEEE Standard 1042-1987 provides a guide to software configuration management. Table 3.2-1 outlines the activities and responsibilities of the V&V team at each phase of the Life-cycle as compared to the QA and CM activities. Note that the items in each column are independent of each other. However, they are coordinated by Life-cycle phase.

3.3 The Value of Detecting Defects Early in the Life-Cycle

V&V is an activity that should take place throughout the Life-cycle, not just at the end of it. Detecting system defects during the requirements or design phase makes recovery relatively inexpensive. However, when the system has been fully implemented, fixes usually involve complete code rewrites which are labor intensive and expensive. The Boehm 1981 study shows that the relative cost of fixing an error increases as a function of the Life-cycle phase in which it was detected; the later the defect is discovered, the more expensive the corrective action. This is evident in Figure 3.3-1.

If a requirements error is discovered in the requirements phase, only the requirements document needs revision. However, if the defect is discovered after code implementation, the requirements document, all subsequent

Table 3.2-1 Lifecycle comparison of activities associated with V&V, Quality Assurance (QA), and Configuration Management (CM)

V&V	QA	CM
REQUIREMENTS	REQUIREMENTS	REQUIREMENTS
o Evaluates, reviews and comments on Requirement Specification	o Reviews specifications for obvious errors and signs off	o Maintains originals of all documents
o Traces requirements to their sources	o Inspects all items received from vendors	o Maintains all source code master files
o Initiates requirement tracing method to trace and verify requirements and links them to detailed design	o Performs on-site factory acceptance of major vendor items	o Receives and logs all documentation received from outside sources
o Ensures accurate translation between Customer's Specification and vendor's Functional Specification	o Participates in reviews	o Maintains all software and documentation libraries
o Inspects hardware configuration for compliance to specifications and contract	o Checks for adherence to standards	o Maintains archives
o Attends reviews	o Monitors vendor performance and quality of documentation	o Maintains total hardware configuration tracking and accounting
o Writes Discrepancy Reports (DRs)		
o Develops V&V Plan	o Reviews design specifications for obvious errors and signs off	o Maintains control of originals on all documentation
DESIGN	DESIGN	DESIGN
o Traces each requirement into design and adds references to requirement tracing method to show linkages	o Attends design reviews	o Manages transfers in design aspects between final CM master files and user directories
o Analyzes detailed design	o Performs analysis of design changes vs. test procedures	o Builds system as required
o Analyzes design document for correctness, feasibility, consistency, testability, operational integrity, etc.	o Checks for adherence to standards	o Backs up system as required
o Performs/analyzes timing and sizing to ensure adequate hardware resources	o Monitors CM actions for completeness and adherence to procedure	o Runs difference program upon request from QA or V&V to verify changes to design
o Analyzes operating sequences, data flow, task interaction, and mode switching		o Maintains informal change control

Table 3.2-1 (Continued)

V&V	QA	CM
DESIGN (Cont.)	**DESIGN (Cont.)**	**DESIGN (Cont.)**
o Looks for unnecessary redundancy, unidentified design element, etc.		o Maintains master file on all Trouble and Discrepancy Reports
o Emphasizes analysis of hi-risk, hi-priority items		
o Analyzes failover and device switching		
o Audits documentation for completeness against the delivered CM library listings - ensures 100% match between requirements document and design		
o Writes Discrepancy Reports		
IMPLEMENTATION, TESTING AND INTEGRATION	**IMPLEMENTATION, TESTING AND INTEGRATION**	**IMPLEMENTATION, TESTING AND INTEGRATION**
o Reviews test procedures for adequacy	o Writes majority of DRs on test failures	o Maintains test result records and test data
o Add test references to requirement tracing method	o Follows test procedure to the letter	o Inspects and inventories systems prior to shipping
o Monitors developer's test program	o Actually performs File Allocation Table	o Monitors all cleaning, packing and loading of system for shipment
o Analyzes test results independently	o Participates in dry run and runs for record	o Participates in Site Acceptance Tests
o Writes DRs on any deficiencies not written up by QA		o Maintains all source code master files
o Performs independent testing if needed		o Certifies final CM build
o Verifies operator/user manual during testing		o Maintains all as-built documentation
o Performs audit of code included in final build		o Initializes and manages all changes to documents and/or code
o Looks for unnecessary, unidentified code, etc.		o Runs difference programs on request from QA and V&V to verify changes to code
o Audits design documentation for completeness against delivered code listings, ensuring 100% match		

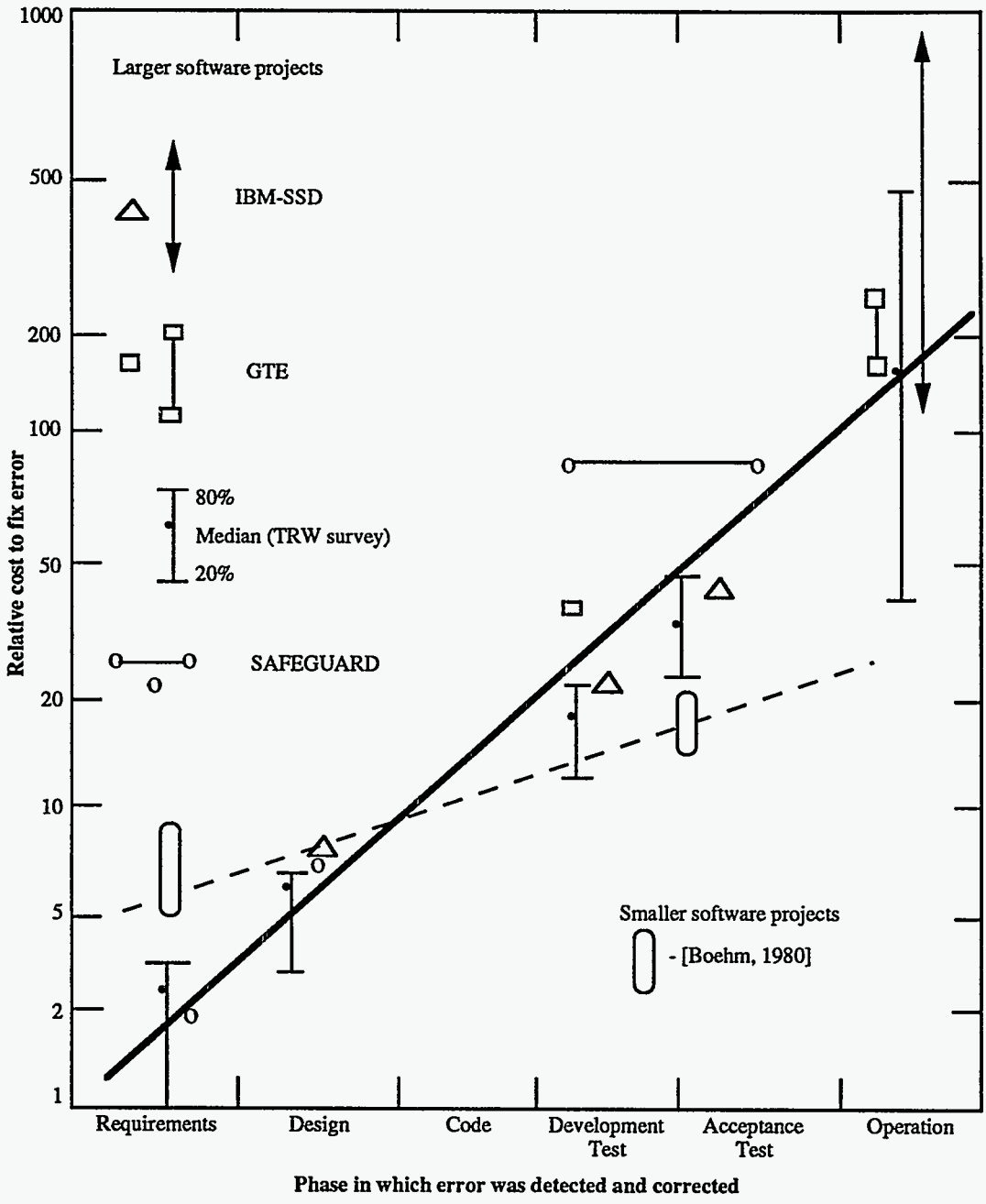


FIGURE 3.3-1. INCREASE IN COST-TO-FIX OR CHANGE SOFTWARE THROUGHOUT LIFE-CYCLE (Figure adapted from Boehm, 1981, his Figure 4-2, p.40)

documentation, and the code itself must be changed. On a large project, an error can be 100 times more costly to fix in the maintenance phase than in the requirements phase. The effect is less extreme in smaller projects, due to less formal procedures for configuration control, but the impact is still a factor. A Rolls-Royce study has even shown that the cost of using labor-intensive formal methods to represent and simulate the requirement specification (see Section 5.2.1) was recovered within the development phase. It was not amortized over the maintenance life of the system.

The total developmental costs were within the budgeted project amounts even though the project remained longer in the requirements/design phases than traditional projects. Nevertheless, the system was completed on schedule and within budget. In contrast, a comparable project with less stringent requirements following the conventional Life-cycle patterns ended up being delivered six months late and at twice the budgeted cost (Hill, 1990).

These results stress the importance of defining and documenting requirements, specifications, and designs prior to implementation. It is important to perform V&V on those documents as well. Good V&V, performed early in the life-cycle can provide tremendous cost savings to the project. The cost savings more than justifies the expenditure for the V&V itself.

4 SOFTWARE DEVELOPMENT LIFE-CYCLE

4.1 Alternatives

The term life-cycle refers to the "start-to-finish" phases of system development. The software development life-cycle encompasses the following: requirements specification, design, implementation, integration, field installation, and maintenance. A software Life-cycle provides a systematic approach to the development and maintenance of a software system. A well-defined and well-implemented life-cycle is imperative for the successful application of V&V techniques. There are two types of life-cycle models: the sequential model and the iterative model. The sequential model is a once-through sequence of steps without providing formal feedback from later phases to prior phases. The iterative model, on the other hand, involves repeated cycling through life-cycle phases.

4.1.1 Sequential Life-cycles

The sequential life-cycle is appropriate when the requirements are well known, when they can be precisely stated, and when the course of implementation is clear. Conventional software systems have traditionally been developed according to the sequential life-cycle model. It has been often called the "waterfall" model; Boehm (1988) and the Department of Defense's MIL-STD-2167 (DoD, 1988) provide examples of the sequential life-cycle model. For the nuclear power industry, the life-cycles described in NSAC-39 and NUREG/CR-4640 are the most referenced. Figure 4.1-1 shows the NSAC-39 life-cycle and Figure 4.1-2 shows the very similar but more detailed NUREG/CR-4640 life-cycle. The NSAC-39 model is the more well-known model, and shows the integration of software activities with hardware development activities. The NUREG/CR-4640 model, which is very similar to that proposed in DoD-STD-2167 and DoD-STD-2167A, is more detailed and shows the documentation products and the formal reviews associated with each phase.

4.1.2 Iterative Life-Cycles

The iterative life-cycle is appropriate when the requirements are not well-known, or are undergoing change, and/or there are significant technical issues/questions about how the software can be implemented to meet those requirements. An iterative model provides successive refinement of requirements and improvement of implementations via a series of prototypes. Uncertain requirements exist in both high technology, state-of-the-art conventional software systems, and in the vast majority of expert system development projects.

The iterative life-cycle exemplified by the spiral model, shown in Figure 4.1-3 (Boehm, 1988), is one of the most well-known of this type. This model defines a succession of four prototypes. It includes a risk analysis to identify the major issues and risks and to select the appropriate alternative solutions to be implemented and tested in each prototype. A precursor to this model was the eternal development cycle model of Deming (Deming, 1985).

The Department of Defense (DoD) has recognized that not all software cannot be developed successfully using the waterfall life-cycle. Therefore, DoD has incorporated revisions in its software development life-cycle standard, MIL-STD-2167A (1988), to allow the flexibility of structuring **an iterative**

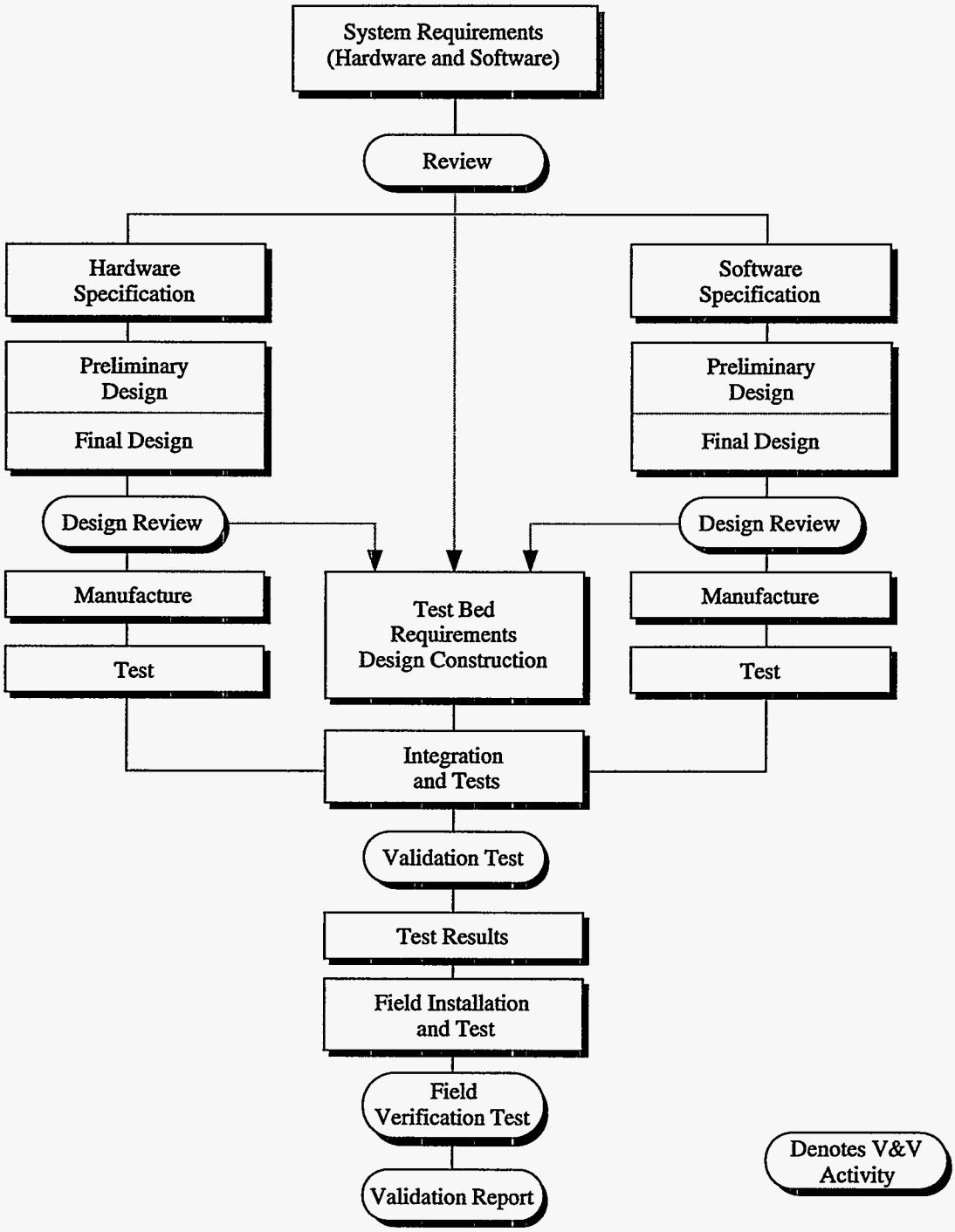


Figure 4.1-1 Relationship of V&V Activities to Generic Project Activities, From NSAC-39 (1981)

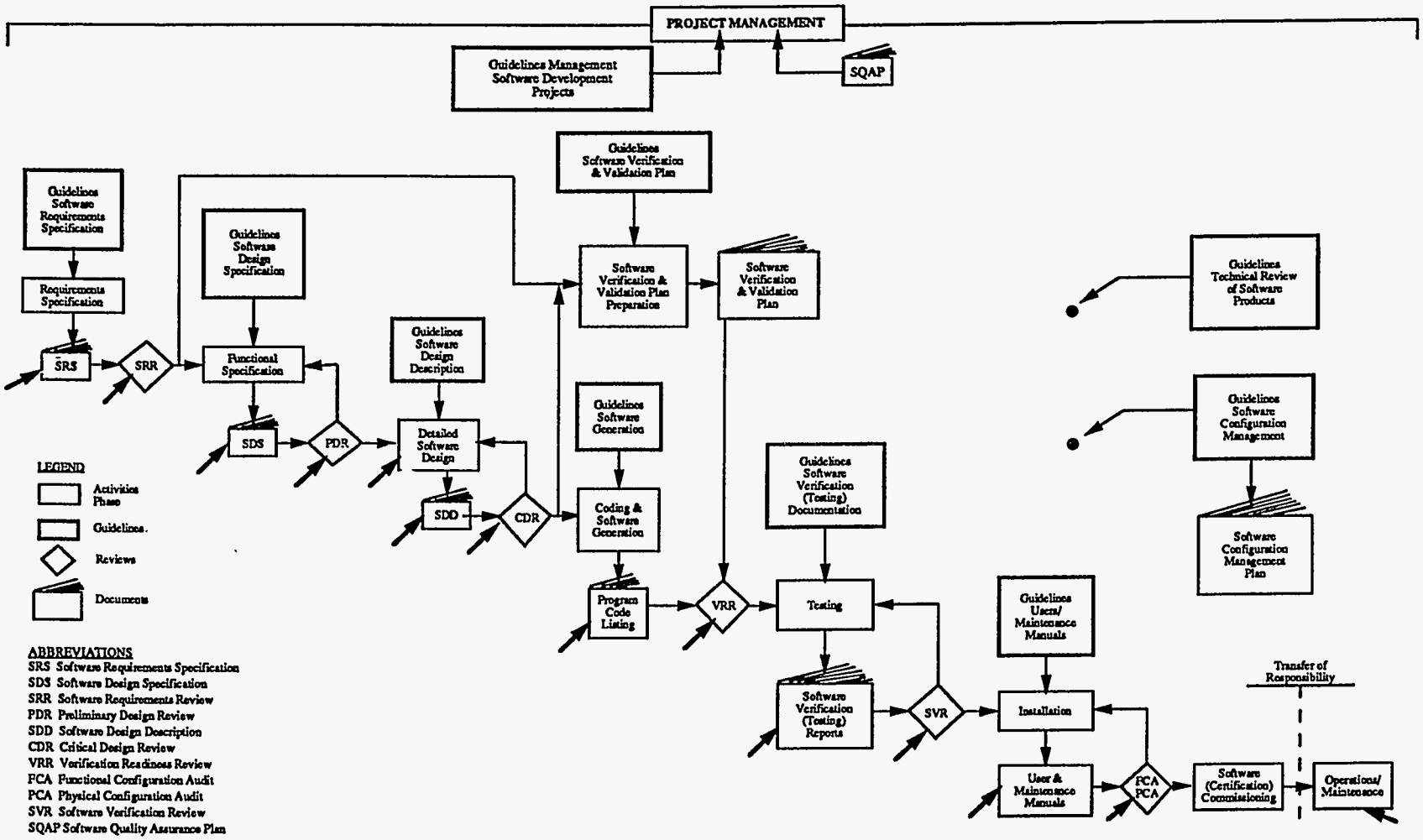


Figure 4.1-2. Software Life cycle, from NUREG/CR-4640 (1987) (redone, for legibility)

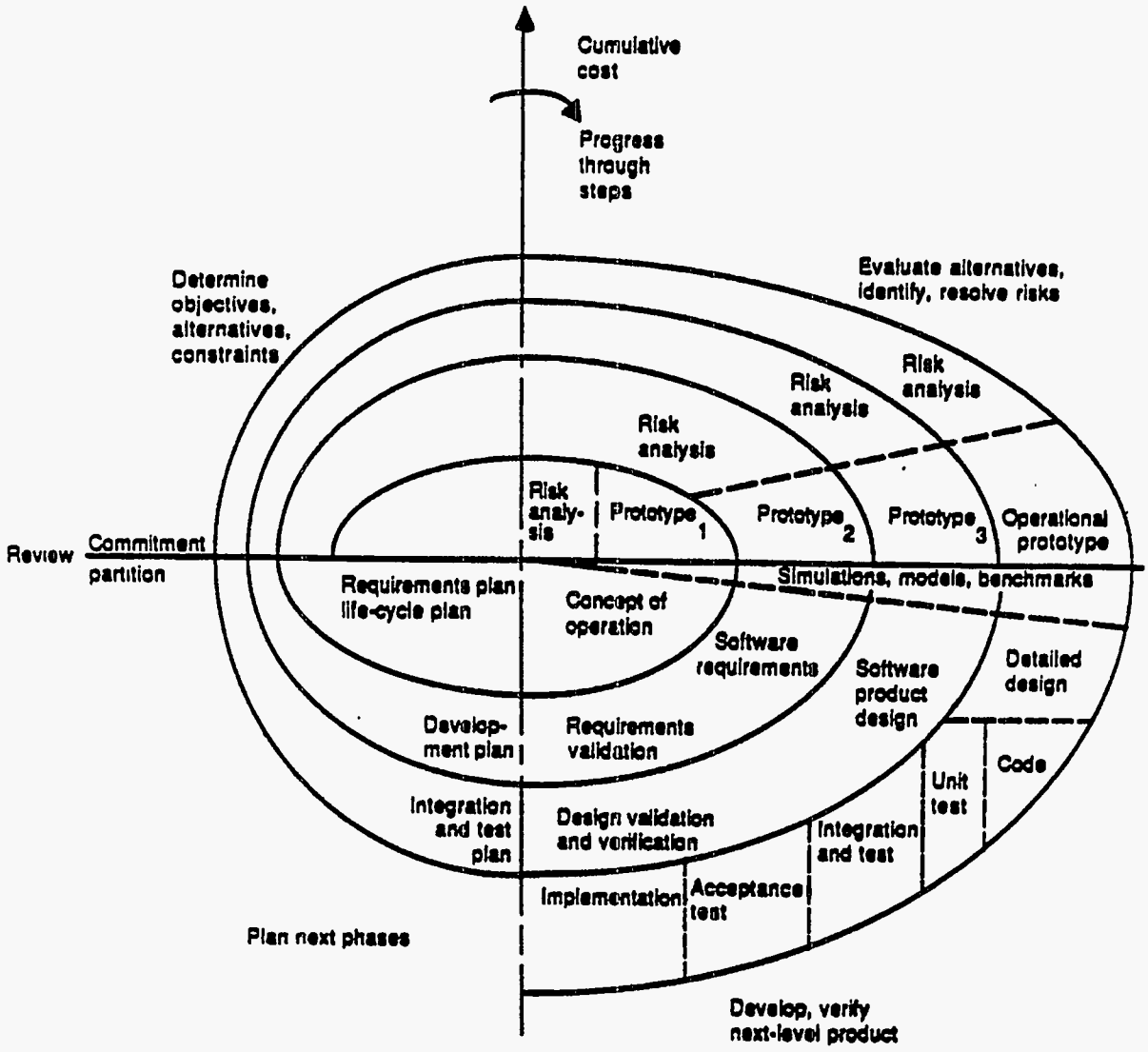


Figure 4.1-3. Spiral Model of the Software Process (from Boehm, 1988)

life-cycle development. An iterative model for expert systems is consistent with MIL-STD-2167A, as shown in Figure 4.1-4 (Miller, July 1990). This life-cycle recognizes the need to liberate initial prototypes from configuration management constraints during exploratory development. Once the requirements are well defined, baseline prototypes are developed using configuration management and V&V constraints until a final system is ready for integration with its delivery environment.

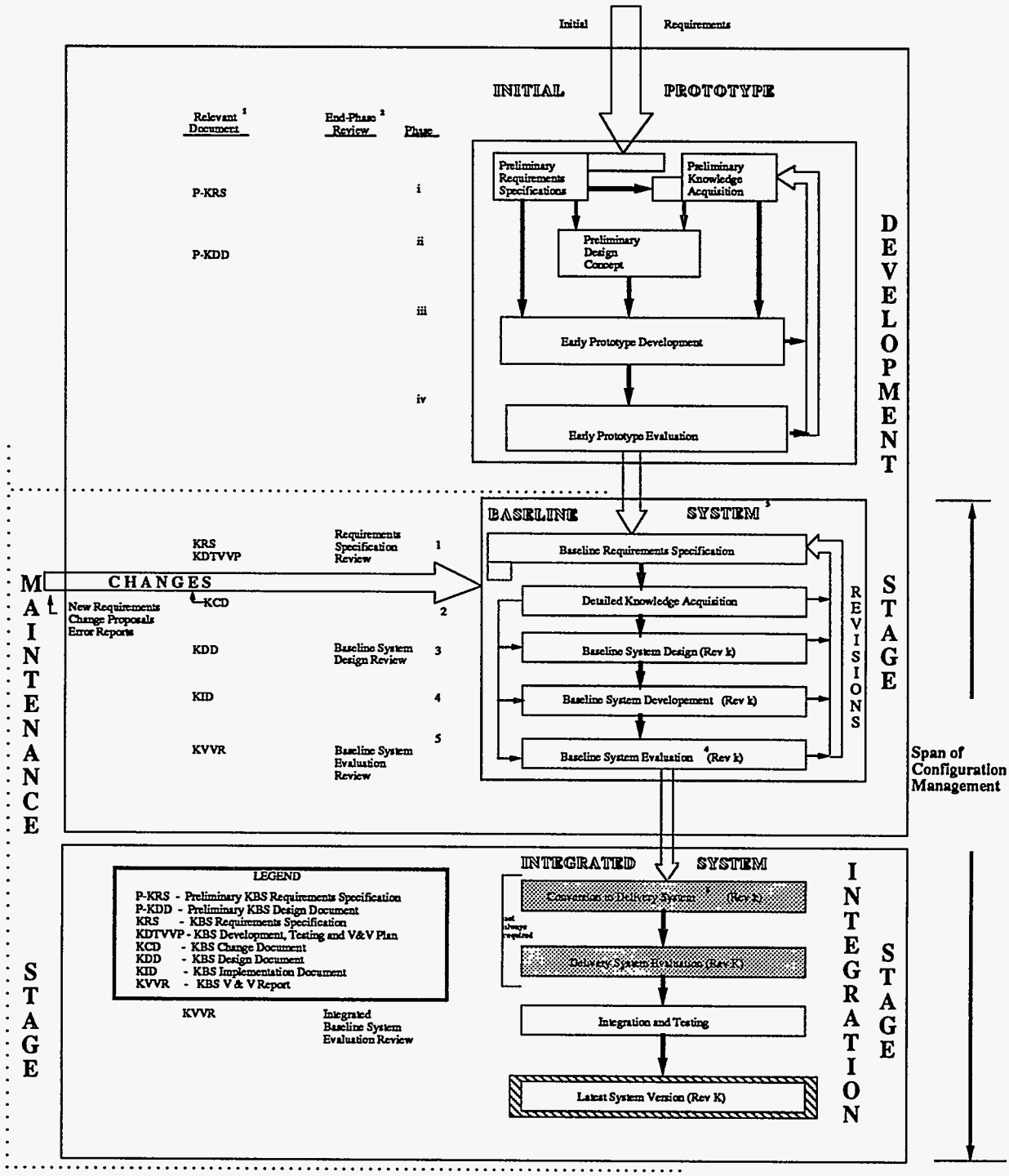
The Incremental Systems Builds (L. Miller, 1990) life-cycle is suggested for conventional software systems, as well as expert systems where high reliability of complex systems is important.[5] This approach calls for breaking the development process down into a series of small construction efforts, each followed by a test and repair activity. The first incremental "build" leads to some significant function that takes the user from the beginning of the application process through to some limited but useful result or output. Then, this miniature of the overall system is thoroughly tested before beginning the next build. Each successive build adds more function and capability until, eventually, the overall system is complete. This Life-cycle has the major advantage that one can always test at the overall system level to determine the quality and performance of the various component interfaces and also the overall operational concept. With each build being a small increment, problems in design or errors in implementation can quickly be detected and corrected. The approach differs from iterative prototyping in that successive prototypes are often are completely different from each other; the incremental build process always layers the new addition onto the base of all previous builds. Although some adjustment to changes in requirements and design can easily be made with this approach, it should not be used if one expects the requirements and design to change radically during development as a result of some kind of discovery process. A major benefit of this Life-cycle is that it is amenable to continuous process improvements. Analysis of the causes of problems found in the testing of each build can lead to improvements in the developmental processes of the next build.

The incremental build approach cycles through three main phases for each build: revision, build, and test/analyze/fix. The revision phase is a combined requirements analysis and design phase. It occurs after the first build (original requirements and design documents) is modified and updated. The build phase occurs when slight to moderate functionality is added to the initial baseline. The build is based on the revised requirements/design obtained from the revision phase. The test/analyze/fix phase consists of those named actions. The build is tested to assure proper function, analyzed for errors, and fixed or corrected if necessary. This Life-cycle accommodates situations where the system development has begun even though requirements are not completely defined. It is especially appropriate for systems requiring very high levels of safety and integrity assurance. The high levels of assurance are achieved by the incremental process of the Life-cycle and the extensive testing during the test/analyze/fix phase.

Figure 4.1-5 shows that there are three types of testing distinguished by temporal direction: past, present, and future. The past direction type tests with regression tests to assure no change in previous function or performance since the last build. The present direction type tests with a number of test types. The future direction type tests with analyses intended to determine how the development process could be improved for future builds and to understand how the present build could enable future integrity-critical failures.

---

[5] This approach embodies features of previous similar models, particularly the iterative enhancement (Basil & Turner, 1975) and the evolutionary Life-cycle models (Gilb, 1973).



Notes

1 Document abbreviations given in Legend

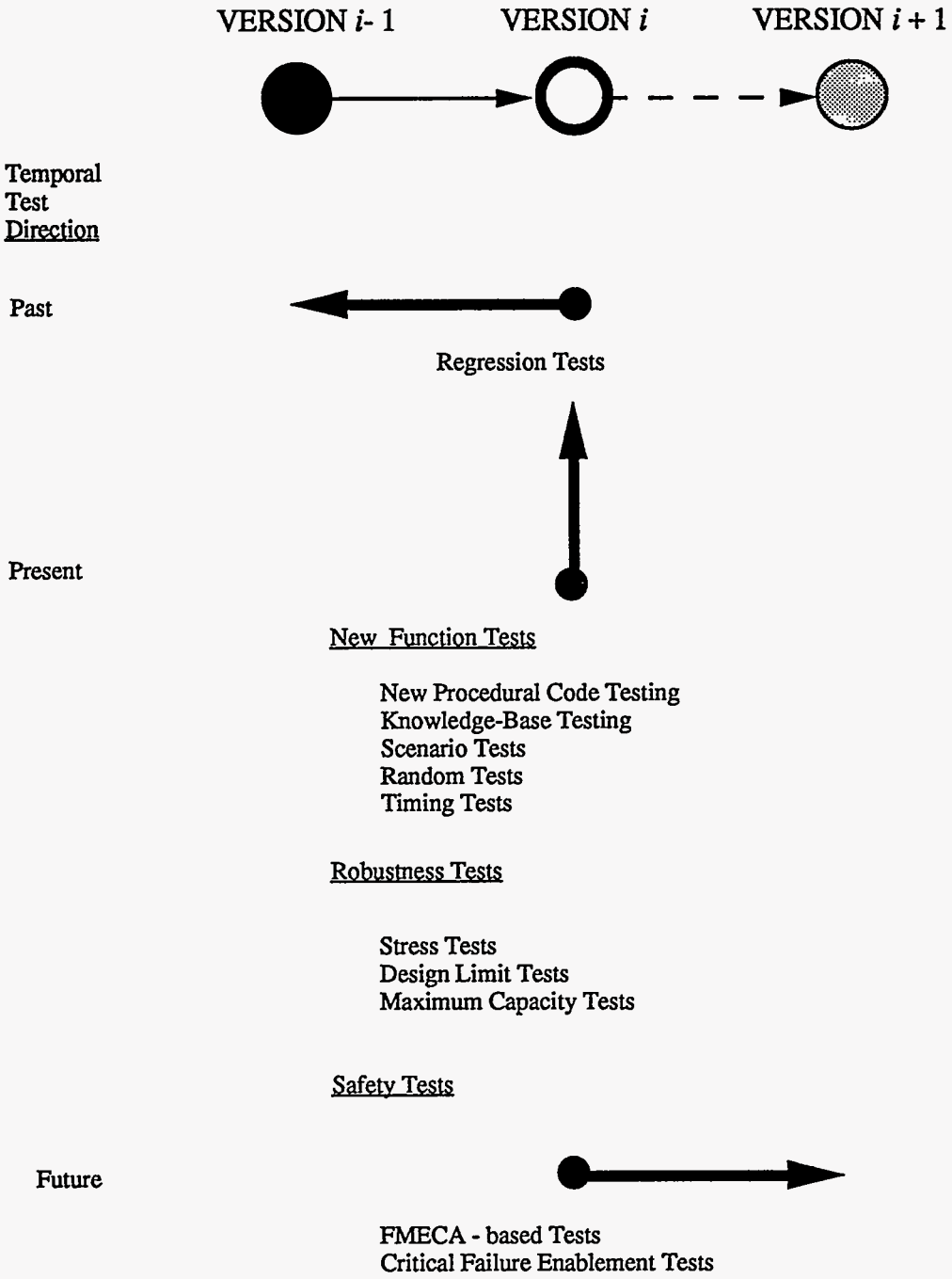
2 Reviews conducted on initial pass through Baseline System Development and on major and/or final revisions

3 Development revisions to occur by incremental knowledge/system build strategy

4 Evaluation includes new-function testing, regression testing, failure-mode analysis and testing, and robustnesstesting. A "Fix-as-you-go" repair strategy is essential

5 Delivery System conversion steps may not be necessary

Figure 4.1-4: An Expert System Life Cycle Consistent with Conventional Software Life Cycle



* Failure-mode, effects, causality analyses

Figure 4.1-5. Testing for Incremental System Builds

In the iterative life-cycle, the general phases of requirements, design, implementation, and maintenance still occur in order, though the first three are dynamic. While the final guidelines will necessarily involve specification of recommended Life-cycles, further consideration of these is not appropriate for the present discussion.

4.2 Reference Life-cycle

A Software Requirements Specification document is prepared in this step of the life-cycle and examined during the Software Requirements Review. Requirements Specifications should include functional requirements, software performance requirements, user performance requirements, and acceptance criteria. Evaluations are performed from both the computer hardware system and software application perspectives. Software interface requirements with hardware, operators, users, and other software are evaluated. IEEE Standard 830-1984 provides a complete description of desired contents and approach to specifying requirements for conventional software.

For the purpose of this survey a traditional waterfall life-cycle based on NSAC-39 (Figure 4.1-1) has been assumed. It consists of the following activities:

1) Requirements definition,
2) Functional specification,
3) Design,
4) Coding and implementation,
5) Integration and testing,
6) Field installation and testing, and
7) Operation and maintenance.

The sections below describe each of these seven steps of the life-cycle model with respect to V&V in more detail. The NSAC-39 life-cycle was chosen as the reference model because it is the simplest and most well-known within the nuclear industry. However, this life-cycle is not the most appropriate for expert systems development. An iterative model is favored for that purpose. However, the following situation does not presume a particular life-cycle, and NSAC-39 provides the most useful model for practical purposes.

4.2.1 Requirements Verification

The purpose of requirements verification is to determine if the requirements specified will correctly and completely describe a system which satisfies its intended purpose. A quality requirements specification is critical to the overall success of the development effort. During requirements verification, each software requirement is uniquely identified and evaluated for software quality attributes including correctness, cons

Requirements tracing is an important V&V technique which begins during the requirements specification stage of the development life-cycle and continues throughout the development process. Traceability of software requirements is a critical attribute for the success of V&V. A software requirement is traceable if its origin is clear, testable (quantifiable), and facilitates referencing to future development steps. Backward traceability is established by correlating software requirements to applicable regulatory requirements, guidelines, and operational concept or any

other preliminary system concept documentation. Forward traceability to design elements and code modules is established by identifying each requirement with a unique name or number.

Requirements verification is accomplished by providing the customer with a requirements document which specifies the environment of concerns, source documents used to develop requirements, needs, goals, assumptions, and constraints of the developmental system. It details the individual requirements assigning one requirement per sentence and numbering unique requirements. The requirements are verified when the customer accepts the requirements document. It should be noted that the customer plays an important role in detailing the requirements document prior to requirements verification.

During the requirements step, critical software functions and their impact to system integrity are also evaluated. The identification of critical software functions in a system with high integrity requirements involves the determination of those specific software modules that would lead to various types of loss of system integrity if they "failed". Such identification provides essential input to the test plan development. This permits special emphasis to be placed on testing these critical components.

A Software V&V Plan is written as part of the project management effort prior to or during the requirements step. This document describes the V&V activities to be performed throughout the development effort. In addition, it defines how the V&V effort will be managed and coordinated with other aspects of the project. It specifies V&V tasks, reports, schedules, and procedures. See IEEE 1012-1986 for further details.

4.2.2 Specification Verification

The second step of the waterfall life-cycle producing a Functional Specification document and conducting a Preliminary Design Review (PDR). The purpose of the PDR is to determine if all the requirements have been mapped to detailed specifications and these, in turn, have all been allocated to the software functional components of the software system architecture. The Functional Specification (sometimes called a Functional Architecture) contains a high level diagram specification of the logical software system architecture. It includes interfaces to other systems. The system is divided into logical functional components which are further described in the Software Design Description in the next step. The Functional Specification describes how the requirements will be met. This is done by mapping each requirement to a functional component within the architecture.

It is at this stage that sophisticated specification verification techniques can be applied to determine the sufficiency of the specification. The creation of an executable model of the specification permits simulation or "animation" of real-time activities. Such animation is highly recommended for Class 1 V&V systems as described in Section 2.4.3, as well as the high-complexity control systems of Class 2 (cells 1 and 2 in Table 2.4.3-1).

4.2.3 Design Verification

In the Design step, a Software Design Description is produced. It is then examined in a Critical Design Review (CDR). The Software Design Description provides a description of the overall system architecture and contains a definition of the control structures, algorithms, equations, and data inputs and outputs for each software module. The complexity of each module is estimated and an effort is made to reduce complexity by breaking up large,

complex modules into smaller ones. IEEE Standard 1016-1987 provides a recommended practice for detailed software design descriptions.

The purpose of the CDR is to evaluate the software design to determine if it correctly represents the requirements and to identify extraneous functions. The Software Design Document is evaluated for software quality attributes such as correctness, completeness, consistency, accuracy, and testability. Also, it verifies compliance with any applicable standards. All interfaces between the software being developed and other software, hardware, and the user environment are evaluated.

Requirements tracing continues during design verification by mapping documented design items to system requirements. This ensures that the design meets all specified requirements. Additionally, non-traceable design elements are identified and evaluated for interference with required design functions. Design analysis is performed to trace requirement correctness, completeness, consistency, and accuracy.

One important aspect of design analysis is an evaluation of data flow, data structures, and the appropriateness of the data attributes. This analysis verifies the correct and efficient handling of data items specified in the requirements and necessary to implement the requirements. Data flow diagrams produced by the developers are analyzed by the V&V evaluator. When data flow information is not included in the developers design documentation, it is often necessary for the V&V group to produce this documentation to facilitate its review. Also, data base structures and attributes are evaluated for correct and complete representations of the data requirements.

During the design phase of development, planning and designing begins for software component testing, integration testing, and system testing. The Software V&V Plan is used as a model for the Software Validation Test Plan. This Test Plan is completed in the Implementation Verification step.

4.2.4 Implementation Verification

During the coding and implementation phase of development, the software detailed design is translated into source code. This activity also creates the supporting data files and data bases. The source code is compiled, assembly errors removed, and individual modules are executed to detect obvious errors.

The purpose of implementation verification is to provide assurance that the source code correctly represents the design. Source code is analyzed to obtain equations, algorithms, and logic for comparison with the design. This process will detect errors made in the translation of the detailed design to code. Information gained during analysis of the code, such as frequently occurring errors and risky coding structures and techniques, is used in finalizing test cases and test data (e.g., Beizer, 1990).

Data flow analysis is a useful technique during implementation verification. Data can be traced through code modules to assure that input values are used but not modified. All output values are assigned as required by the data flow analysis performed during design verification. In some cases, data flow analysis can be automated or performed manually (Beizer, 1990).

Code instrumentation can be used to provide a means of measuring program characteristics. This process inserts checks or print-out statements into the code to audit the behavior of the code while it is executing commands. Instrumentation can be used to check data structure boundaries, data values within allowable ranges, loop control checking, and tracing of program execution.

Unit or module testing is conducted to assure each software module is operating correctly before it is integrated with the rest of the system. ANSI/IEEE Standard 1008-1987 provides a description of software unit testing activities.

During V&V evaluations in this phase of the life-cycle, the source code is traced to design items and evaluated for completeness, consistency, correctness, and accuracy. Interfaces between source code modules are analyzed for compatible data elements and types. Source code documentation and programmer and user's manuals are all reviewed for completeness, correctness, consistency, and accuracy. A Verification Readiness Review is held to determine if the system is ready for integrated system testing.

The Software Validation Test Plan (or alternately, Customer Acceptance Test Plan) is completed in preparation for the next step. It identifies the testing approach, schedule, and activities. Detailed test cases and test procedures are generated and documented using the knowledge gained about the program through its structure and detected deficiencies.

4.2.5 System Validation

During the System Validation step, the system as a whole is evaluated against the original Requirements Specification. Validation consists of planned testing and evaluation to ensure that the final system complies with the system requirements. The Software Validation Test Plan (or Customer Acceptance Test Plan) is utilized during this step, and validation may be performed by an independent third party.

Validation is more than just testing; it involves analysis. A test is a tool used by the validation team to uncover previously undiscovered specification, design, or coding errors throughout the development process. Validation uses testing plus analysis to reach the objectives stated above. The analysis is the design of test strategies, procedures, and evaluation criteria, based on knowledge of the system requirements and design, which proves system acceptability in an efficient fashion.

The purpose of system validation is to demonstrate that the final system meets the intent of the requirements. Validation may consist of independent tests performed by a third-party V&V group, a combination of independent tests and developers' tests, or an independent review of the developers tests by the V&V group. The amount of independence of the V&V testing activity is determined by the criticality of the software being tested. For example, USNRC requires that the V&V group be independent from the developers for Class 1E systems.

A Software Validation Test Plan is a critical component in the success of the validation effort. Tests must be defined to demonstrate that all testable requirements have been met. The test plan includes a description of the purpose, scope, and level of detail for each testing activity. The test organization and responsibilities are fully described. Documentation of testing activities and results should be specified to ensure consistent documentation for

all tests. ANSI/IEEE Standard 829-1983 provides a complete description of basic test documentation. It specifies that test methods be described and justification for selected methods be provided. The standard requires the identification of support software and hardware to be included in the testing environment. The ANSI standard suggests that test standards and criteria for test results and product acceptance are specified so an informed acceptance judgment can be made. In addition, it requires procedures developed for actions taken when tests fail and a determination if testing can proceed.

Test cases and test procedures are evaluated for completeness, correctness, clarity, and repeatability. Requirements tracing continues during validation by tracing test cases to requirements. This ensures that all testable requirements are covered. Expected results specified in the test cases are verified for correctness against the requirements and design documentation.

Tests are performed in accordance with the previously developed test plans and procedures. Test results are evaluated against the criteria specified in the test procedures. Test results are verified to ensure that the correct test inputs are used, outputs are correctly reported, and all test cases were correctly executed in the appropriate environment.

In rapid-prototyping development efforts utilizing an iterative life-cycle, maintaining a regression test case set is especially important. Each test case in the set must be indexed to the requirement(s), design element(s), and code module(s) it tests. Therefore, if the requirements, design elements, or code module(s) change in a given iteration, the test case can be marked for update, deletion, or replacement. Additionally, new test cases must be designed for new requirements, design elements, and code modules which were added during the latest prototype development. Thus, the regression test set is changed at the end of each prototype implementation to reflect the changes in the system requirements, design and implementation.

4.2.6 Field Installation Verification

The purpose of field installation verification is to assure that the software installed in its target environment has not degraded since validation testing. This is typically accomplished by executing a subset of the functional tests performed during validation testing with the software in its final configuration. During this time, all field inputs are carefully checked to ensure that they are properly connected to the system in its operational environment.

During the field installation phase of the development, developers make final modifications to the software documentation. This final documentation represents the primary source of information about the software during operation and maintenance. This final documentation should be verified to assure that it accurately and completely represents the software being placed in operation.

The final V&V step in the development life-cycle is the preparation of a report which summarizes the V&V activities performed, describes results, and presents any recommendations and final conclusions resulting from the V&V effort. This final report provides the status of all discrepancies reported during the V&V effort.

4.2.7 Operation and Maintenance Phase V&V

During the operation and maintenance phase of the life-cycle, modifications may be made to the software and its operational environment. To maintain the verified and validated status of the operational software, an ongoing V&V program is established. The V&V plan used during the development effort is revised to reflect the operational environment constraints and procedures.

A critical factor in the success of V&V during operation and maintenance is the existence of a configuration management program to control modifications to the code, documentation, and the operational environment. There is no way to maintain the verified and validated status of the software system without adequate control of changes. One of the first V&V tasks during operation and maintenance should be to evaluate the configuration management program for adequate change control.

A configuration management program provides a formalized change request and approval process. Change requests should be submitted for any proposed change to the software, documentation, or operating environment. All change requests should be reviewed by the V&V group to determine the impact on the total operational system and documentation. Appropriate V&V tasks are determined by evaluating which development phase products are affected. For example, if a software modification impacts the requirements specification, then appropriate V&V activities should be selected from requirements verification and each subsequent V&V phase.

Changes to the software operating environment include modifications of operating system software or hardware. An impact evaluation on software performance is done when these types of changes are made to the operating environment. This evaluation consists of appropriate field verification activities. The software is verified in its new environment by performing a subset of field verification tests, including previously used regression test-suites, and demonstrating no significant difference in the test results (Beizer, 1990).

5 CLASSIFICATION OF V&V METHODS FOR CONVENTIONAL SOFTWARE

The survey approach consists of three major stages: classification of conventional V&V methods, characterization of these methods, and assessment of their applicability to expert systems. The classification of conventional V&V methods is the subject of Section 5. The other stages (characterization and assessment) will be described in detail in Sections 6 and 7, respectively.

5.1 General Observations and Approach

The classification stage of the survey approach consists of three distinct activities, as illustrated in Figure 2.5.1-1. First, a wide variety of technical sources were reviewed for descriptions and references to conventional V&V methods. Second, these methods were sorted by their relevance to the main life-cycle phase. Third, the life-cycle groups were partitioned into natural sets.

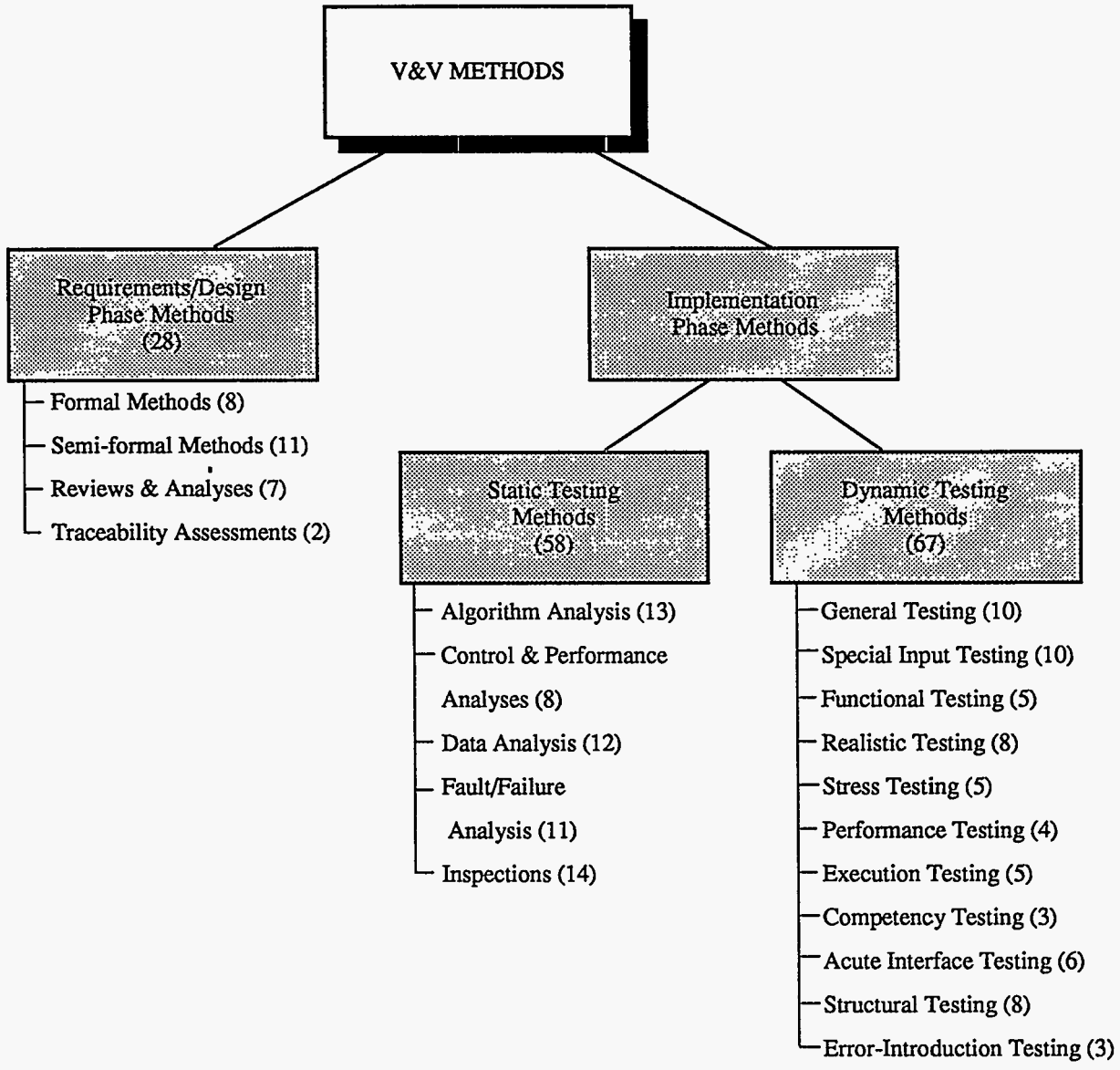
A comprehensive survey was conducted. The authors gathered and reviewed over 300 technical sources. Source material included journal articles, institutional reports, proceedings of software testing conferences, professional communications, and standards or guidelines. The most informative sources were the more recent texts and publications on software testing and methodology. Many of these sources were obtained internally, through local libraries, or through a DIALOG search.

This search yielded the desired results; however, a few difficulties are noted. First, no single source, even the best texts on testing, covered the whole body of assembled techniques. Second, different authors used different phrases to refer to the same technique. Occasionally, the same author used synonyms for the same technique, without explanation. Lastly, authors often did not give detailed definitions or descriptions of their methods, making it difficult to determine how these methods were related or stood up to apparently similar methods.

The criterion for reporting similar methods as separate items or grouping them under a single name was determined by the authors' contrasting (not detailing) methods within their article. Therefore, when similar methods were grouped together, the most accepted name or description for that method was used. A large number of methods are identified here, and it is important to emphasize that the V&V of any single software system would involve only a small subset of these techniques.

5.2 The Three Major Categories and Their Classes

A total of 153 different techniques were discovered. Although these techniques fall into two distinct life-cycle phases, they are clustered into three major categories, as shown in Figure 5.2-1. One category is for the Requirements/Design phase of a Life-cycle, and two are for the implementation phase of a Life-cycle, corresponding to the major distinction of static vs. dynamic testing. Static testing involves analysis and inspection of the system's source code without actually executing the code. Dynamic testing involves the actual execution of the system's code. All the appropriate environments, drivers, and interfaces are installed and operated on a platform, and the system's outputs are obtained for a set of inputs. Each of the entries in Figure 5.2-1 represents a major class of techniques for that category. Each of these classes may be divided into subclasses, but the total number of individually identified techniques in that class is given in parentheses after each entry. The number of major classes, the total number of individual techniques, and the relative percentages of the total by category are shown in Table 5.2-1.



* Number in parentheses indicate the number of individual number of V&V Methods of that type.

Figure 5.2-1. Classes of Conventional V&V Methods Organized by Life-Cycle Phase

Table 5.2-1 Statistics concerning the three major categories of conventional V&V techniques

Major Category	Major Classes	Techniques	Percentage of Total Techniques
Requirements/Design	4	28	18%
Static Testing	5	58	38%
Dynamic Testing	11	67	44%
TOTALS	20	153	100%

The number of Requirements/Design methods reflects the relatively small number of system products for examination at those early phases. The greater number of Dynamic over Static methods reflects the greater complexities of testing an operational system in a specific operating environment. The greater number of Dynamic techniques, 44% of the total, may also reflect programmers' preferences for the most direct approach of executing a program to see how well it runs. Full descriptions of the three technique categories are provided in Table 5.2-2.

5.2.1 Requirements/Design Methods

The Requirements/Design techniques consist of four major classes and their various subclasses. These major classes of techniques are: formal methods, semi-formal methods, reviews and analyses, and traceability assessments. The methods for the first two Life-cycle phases are grouped together because three of the four classes are very much the same whether applied to a requirement specification or a design description. However, the fourth class, Traceability Assessments, involves comparisons between the products of the two phases. The descriptions of the 28 individual conventional Requirements/Design V&V techniques are provided in Table 5.2.1-1; however, brief comparisons are given below.

The Formal Methods involve mathematical and logical calculations for expressing relationships among data and other objects and the processes which interact with them. Using these methods, one can prove various important properties about the system represented, such as the absence of contradiction. The Semi-Formal Methods often involve rigorous constraints on notations, sequencing, and selection of operators/objects to achieve their goal of guiding the analysis or specification within well-defined limits. They are less difficult to apply than the Formal methods. Both the Formal and Semi-Formal methods provide for language representations of systems of varying complexity; the former in more mathematical notation and the latter in more graphical network styles. Both have had variants for the last 15 years, but they have been employed by very few software engineers. With the advent of powerful desk-top computers, graphic interfaces, and local database management systems, these methods are increasingly promoted and are on the rise in usage as they are implemented in computer-based tools.

A major advantage of the semi-formal methods is that their philosophy of supporting system-engineering descriptions in a graphical mode greatly facilitates simulating or animating requirements specifications and designs. Such capability is believed by many to be an essential aspect of developing and assuring the quality of highly complex systems requiring high integrity.

Both the Reviews and Analyses and the Traceability Assessments are absolutely essential to effective quality assurance of any system. Reviews and Analyses are well-worked out procedures for various parties having an interest in the final system to hear presentations on the work in progress and express their concerns. The Traceability Assessments establish the relations between (in the present case) the requirements specification and the design, matching elements of one to the other. After matching, all that remains is either a set of unmapped requirements elements, unfulfilled requirements, or a set of unmotivated additional design elements, unintended design functions. The first clearly signals design inadequacies, and the second raises strong concerns that the non-specified additional functions might lead to unexpected errors and performance and/or safety problems.

Table 5.2-2 Description of major classes of techniques

V&V CLASSES/SUBCLASSES	DESCRIPTION
1.0 REQUIREMENTS AND DESIGN EVALUATION	**EVALUATION OF THE ADEQUACY OF THE REQUIREMENTS AND DESIGN**
1.1 Formal Methods	Use of mathematical and logic formalisms for rigorous and unambiguous representation of initial system documents, including the requirements document, the requirements specification, and the design document. These representations may then be subjected to formal (sometimes automated) deductive reasoning to detect anomalies or defects such as "correctness", "contradiction", "completeness", "deadlock", and "consistency".
1.2 Semi-Formal Methods	Techniques whose normal, forced, or prescribed method of use effectively constrain users in their specification of requirements or designs, such that various problems of expression and elaboration can be avoided or reduced. Such problems include aspects of ambiguity, incompleteness, inconsistency, contradiction, and "ill-formedness." These techniques, while often based on mathematical and logic formalisms, do not explicitly require the user to specify or use such formalisms. The techniques are typically embedded in function-rich, computer-based environments which provide sophisticated graphical representations of user input and often permit the user specifications to be simulated or animated to permit assessment of time and performance characteristics.

Table 5.2-2 (Continued)

V&V CLASSES/SUBCLASSES	DESCRIPTION
1.0 REQUIREMENTS AND DESIGN EVALUATION (cont.)	**EVALUATION OF THE ADEQUACY OF THE REQUIREMENTS AND DESIGN**
1.3 Formalized Reviews and Analyses	Reviews and specialized analyses by various specified personnel of requirements or design products. The reviews follow a detailed checklist or set of procedures.
1.4 Traceability Assessments	Determination of correspondence between individual requirements and design elements, between individual requirements and implemented system features, or between design elements and implemented system features. The two types of problems identified by these analyses are (1) unfulfilled requirements or design elements, and (2) unintended (unmotivated) design or implementation elements.
2.0 STATIC TESTING	**EXAMINATION OF THE PROGRAM SOURCE CODE OR SOME TRANSFORMATION OR MAPPING TO SUPPORT VARIOUS KINDS OF ANALYSES (e.g., UNUSED CODE, INCONSISTENCIES, ANOMALIES).**
2.1 Algorithm Analysis	Analysis of the overall algorithm(s) for achieving required function.
2.2 Control Analysis	Analysis of the control characteristics of the program.
2.3 Data Analysis	Analysis of the data specifications and flow of the program.
2.4 Fault/Failure Analysis	Analysis for particular or any kind of fault or failure, and/or an analysis to determine how particular faults and failures could occur.
2.5 Inspections	Examination of various aspects of the program by various personnel.

Table 5.2-2 (Continued)

V&V CLASSES/SUBCLASSES	DESCRIPTION
3.0 DYNAMIC TESTING	**ACTUAL EXECUTION OF THE PROGRAM, GENERATING OUTPUT FOR SETS OF INPUT CONDITIONS.**
3.1 General Testing	Generic and statistical methods for exercising program.
3.2 Special Input Testing	Special methods for generating test-cases to explore the domain of possible system inputs.
3.2.1 Random Testing	Selecting test-cases according to some random statistical procedure.
3.2.2 Domain Testing	Analysis of the boundaries and partitions of the input space and selection of interior, boundary, extreme, and external test-cases as a function of the orthganality, closedness, symetry, linearity, and convexity of the boundaries.
3.3 Functional Testing	Selecting test-cases to assess required functionality of program.
3.4 Realistic Testing	Choosing inputs/environments comparable to intended installation situation.
3.5 Stress Testing	Choosing inputs/environments which stress the design/implementation of the code.
3.6 Performance Testing	Measuring various performance aspects for a list input.
3.7 Execution Testing	Actively following (and possibly interrupting) sequence of program execution steps.
3.8 Competency Testing	Comparing the output "effectiveness" against some pre-existing standard.
3.9 Active Interface Testing	Testing various interfaces to the program.
3.10 Structural Testing	Testing selected aspects of the program structure.
3.11 Error-Introducing Testing	Systematically introducing errors into the program to assess various effects.

Table 5.2.1-1 Description of the conventional requirements/design V&V methods

V&V Classes/Subclasses	Description
1.1 Formal Methods	
1.1.1 General Requirements Language Analysis/ Processing (Davis, 1990)	Expression of requirements specifications in a special requirements language and analysis of execution of that expression to assess the adequacy of the requirements.
1.1.2 Mathematical Verification of Requirements (Jones, 1986)	Translation of requirements into mathematical form for proving various properties (security, ultra-hi reliability).
1.1.3 EHDM (Rushby, 1991)	A specification (and verification) language based on a strongly typed higher-order logic, incorporating elements of the Hoare relational calculus, with complete formal semantic characterization.
1.1.4 Z (Chisholm, 1990)	A typed set-theoretic language employing mathematical expressions, schema, to describe aspects of a system; the schema consist of declarations grouped with property predicates about the declarations.
1.1.5 Vienna Definition Method (Jones, 1986)	A discrete-mathematical formalism for rigorously defining the semantics specification processes.
1.1.6 Refine Specification Language (Ng, 1990)	A knowledge-based commercial specification language and environment based on transformational programming concepts.
1.1.7 Higher Order Logic (HOL) (Gordon, 1985)	An implemented logic notation that allows specifications to be written in terms of hierarchically structured collection of logical theories which contain axiomatic properties of the operations that are introduced.
1.1.8 Concurrent System Calculus	Provides a calculus for the description and specification of concurrent systems. Similar to Milner's (1986) calculus, and basis for LOTOS (ISO, 1987) Language for Temporal Ordering Specification.
1.2 Semi-Formal Methods	
1.2.1 Ward-Mellor Method (Ward, 1986)	An extension of Structured Analysis system specification techniques (e.g., Ross, 1977; DeMarco, 1978) developed at Yourdon, Inc. for real-time systems, emphasizing data flow diagrams & control-flow annotations.

Table 5.2.1-1 (Continued).

V&V Classes/Subclasses		Description
1.2.2	Hatley-Pirbhai Method (Hatley & Pirbhai, 1987)	Like Ward-Mellor, except that the techniques were developed at the Boeing and Lear companies; emphasizes control flow diagrams; considered to have superior architectural modeling capability.
1.2.3	Harel Method (Harel, 1987)	Like Ward-Mellor, but using unique **Statechart** notations to accomplish similar modeling as the above, but generally considered richer and more elegant. Implemented in a set of tools called STATEMATE (cf. Harel et al., 1990).
1.2.4	Extended Systems Modeling Language (ESML; Bruyn et. al., 1988)	A modeling language with elements of the Ward-Mellor and Hatley-Pirbhai methods, currently under development.
1.2.5	Systems Engineering Methodology (SEM; Wallace, 1987)	A method combining structured analysis techniques and software cost reduction methods (Heninger, 1980), similar to ESML.
1.2.6	System Requirements Engineering Methodology (SREM; Alford, 1977)	A hardware/software specification language for describing both data-and control-flow of systems, used extensively in US DoD weapons systems development (also known as the DCDS method; later implemented in a commercial system called TAGS). Now fully implemented, with complete system support, as the RDD-100 tool (supports design animation; Ascent, 1990).
1.2.7	FAM (Chisholm, 1990)	Representation of systems in terms of graphical annotated **flow-nets**, based on extensions to Petri-net theory, permitting symbolic execution with an automated theorem prover.
1.2.8	Critical Timing/Flow Analysis (Wallace, 1989)	Modeling and (usually) simulation of process and control timing aspects of the design to determine if such requirements are satisfied (e.g., with Petri Nets).
1.2.9	Simulation-Language Analysis (Hartway, 1990)	Representation of a system design in a general purpose simulation language (e.g., SLAM II), and analysis of the execution results.
1.2.10	Petri-Net Safety Analysis (Leveson & Stolzy, 1987)	Systems modeling with untimed (and timed) Petri nets to assure design adequacy for catastrophic-failure and other safety problems.
1.2.11	PASL/PSA (Teichroew, 1977)	Constrained natural language-like representation of requirements and specifications with automated support.

Table 5.2.1-1 (Continued).

V&V Classes/Subclasses	Description
1.3 REVIEWS AND ANALYSES	
1.3.1 Formal Requirements Review (NBS500-93, 1982)	Review by special personnel of the adequacy of the requirements specification according to detailed pre-established set of criteria and procedures.
1.3.2 Formal Design Review (NBS500-93, 1982)	Review by special personnel of the adequacy of the design according to detailed pre-established set of criteria and procedures.
1.3.3 System Engineering Analysis (DSMC, 1990)	A variety of activities associated with developing a complete operational system which satisfies certain requirements; these activities include creation of a functional architecture, allocating function to hardware and software, accomplishing trade-off and make/buy studies, and development of a work-breakdown structure.
1.3.4 Requirements Analysis	Analysis of requirements to ensure completeness, consistency, clarity, explicitness, etc.
1.3.5 Prototyping (Schulmeyer, 1992)	Building a model of a design to evaluate one's approach or to better define the requirements; prototypes may range from mock-ups to initial versions which are retained and built on.
1.3.6 Database Design Analysis (Nijssen, 1989)	Checking the design of the structure, normal form, declarations, and values of a database.
1.3.7 Operational Concept Design Review (Rasmussen, 1987)	Review of the design of the concept of operations for the system, especially the interaction with human operators.
1.4 TRACEABILITY ASSESSMENTS	
1.4.1 Requirements Tracing Analysis (NBS500-93, 1982)	Identification of individual requirement aspects and tracing of these to design aspects, and from the design to aspects of the implemented program.
1.4.2 Design Compliance Analysis (Wallace, 1989)	Verification process that design is compliant with--realizes--all aspects of requirements.

The following four trends could accelerate the development and use of this front-end class of V&V techniques:

1. CASE (Computer Aided Software Engineering) tools are becoming widely available. They provide disciplined and feature-rich environments for developing specifications and designs, providing all manner of data-dictionary and consistency-checking support.

2. The development of languages for specifications outside the software community is emerging. These activities are coming from the advanced manufacturing areas and are driven significantly by the need for communication among CAD/CAM tools among various suppliers at various stages of manufacturing. These range from the older entity-relation **IDEF** family of languages to the more recent process-property relations and semantics, such as **NIAM, EXPRESS**, and the emerging **PDES/STEP**, (Chen, 1990).

3. Standards and guideline activities, particularly in Europe, are emphasizing the utility and the necessity of formal proving methods. This is evidenced by the British draft standards MOD 0055 and 0056, (UK, 1989).

4. The capability of automatic code-generators is increasing rapidly. This trend has been stimulated by the success of automatic generation of application code for fourth-generation language query and other systems, and by the burgeoning interest in reverse-engineering.

5.2.2 Static Testing Methods[6]

This category comprises five major classes of techniques, as described in Table 5.2.2-1. All of these involve examinations based on the system source code without actual execution of that code. These major classes are: algorithm analysis, control analysis, data analysis, fault/failure analysis, and inspection. Some involve only the source code, while others involve transformations or simulations of that code into special analysis languages or formats.

The **Algorithm Analysis** methods, the first static-testing category, involve analysis of the algorithm(s) embodied in the program and, in most techniques, the translation of the algorithm(s) into some kind of language or structured format (all but Technique 2.1.9). Algorithm analysis provides the means for microscopic examination of the software system's process-logic and its adequacy, in the area of the requirements and design. This examination permits the detection of unimplemented or unintended functions.

Some control analysis techniques detect and characterize program control-flow with the sequential and hierarchical aspects (2.2.1-2.2.3, 2.2.5). Two techniques detect the timing aspects (2.2.6-2.2.8). However, one method focuses on the concept of operations, particularly as it involves the user (2.2.4).

---

[6] To make this a comprehensive report it includes the few non-conventional V&V static testing methods that were discovered, in a different task, to be regularly applied to AI systems. The methods are 2.1.12, 2.1.13, and 2.4.8 - 2.4.11.

Table 5.2.2-1 Description of the Conventional Static Testing V&V Methods

Static Testing V&V Methods	Description
2.1 Algorithm Analysis	
2.1.1 Analytic Modeling (Jones, 1986)	Representing the program logic and processing in some kind of model and analyzing it for sufficiency.
2.1.2 Cause-effect Analysis (Davis, 1990)	Identifying the triggers of processes, their effect during activation in states of variables, and the final terminating conditions.
2.1.3 Symbolic Execution (King, 1976)	Representing the data computations as algebraic equations and solving these algebraically through the whole program.
2.1.4 Decision Tables (Omar, 1991)	Tables which represent different logical combinations of events or conditions that might occur and the actions to take when they do occur. Used as a static method to identify functionality and to provide the basis for selection of dynamic testing test-class.
2.1.5 Trace-assertion Method (Parnas, 1988)	An algorithm specification (or representation) method involving description of the sequence of invocations of system modules, including I/O values, in terms of v. axiomatic assertions about the traces, in a "black-box" fashion. Used in conjunction with "A-7 Table Format" representation (similar to decision tables).
2.1.6 Functional Abstraction (Mills, 1987)	Representing design or program as a series of mathematical functions based on a small set of primitive programming functions (e.g., iteration, sequence, select, etc.) then, recursively, dividing parent functional specifications into sub-specifications and mathematically verify equivalence.
2.1.7 L-D Relation Methods (Parnas, 1988)	An alternative to functional specifications for non-deterministic programs using relations and the competence set of states in which termination is guaranteed.
2.1.8 Program Proving (Mills, 1987)	For each code segment, developing formal specifications of functional intent and specific I/O characteristics; for actual or symbolic input then proving via some proof procedure that the segment performed as intended.
2.1.9 Metric Analyses (Jensen, 1985)	Computation of various complexity metrics for the program.
2.1.10 Algebraic Specification (Uhrig, 1985)	Specification of program procedures in terms of algebraic expressions.

Table 5.2.2-1 (Continued).

Static Testing V&V Methods	Description
2.1.11 Induction-Assertion Method (Hoare, 1985)	Use of abstract data types to represent a specification so that proofs of correctness can be implemented using these.
2.1.12 Confidence Weights Sensitivity Analysis (O'Leary, 1990)	Using statistical analyses to measure the sensitivity, accuracy or bias in the confidence factors/weights placed on rules' conclusions.
2.1.13 Model Evaluation (Hamscher, 1992)	Evaluation of models in the system by modeling experts and by subject matter experts.
2.2 Control and Performance Analyses	
2.2.1 Control Flow Analysis (Ward, 1985)	Analyzing the program into a series of decision and process actions and representing all of the possible alternative process sequences in the program. Often used to assess whether program is well-structured and has no unreachable code.
2.2.2 State Transition Diagram Analysis (NBS500-93, 1982)	Determining the condition (variable states, etc.) that trigger the onset and cessation of program processes.
2.2.3 Program Control Analysis (NUREG/CR-4640, 1987)	Related to 2.2.1 and 2.2.2 but concerned more with the sequential aspects leading to a particular execution path in a program.
2.2.4 Operational Concept Analysis (Rasmussen, 1987)	Analysis of the manner in which the software system interacts with and is dependent upon states of the environment and external decisions, especially of human operators.
2.2.5 Calling Structure Analysis (NBS 500-93, 1982)	Module by module analysis of hierarchically structured programs involving procedure calls to determine what sequence of higher level module calls led to the invocation of a particular module, and what modules in turn are called by it.
2.2.6 Process Trigger/Timing Analysis (Hatley, 1987)	Analysis of the conditions which activate a process (similar to 2.2.2) with special concern for the timing of activation relative to other processes.
2.2.7 Worst-case Timing Analysis (Wallace, 1989)	Analysis to determine the longest execution-time path through a program often comparing this to a reference safety limit.
2.2.8 Concurrent Process Analysis (Rattray, 1990)	Analysis of the overlap or concurrency of different processes in multi-tasking, parallel processing, or concurrent processing programs.

Table 5.2.2-1 (Continued).

Static Testing V&V Methods	Description
2.3 Data Analysis	
2.3.1 Data Flow Analysis (Deutsch, 1982)	Analysis of the data inputs, outputs, and controls to all program processes.
2.3.2 Signed Directed Graphs(Suddath, 1991)	Graphical qualitative representation of the directional effects of system states on other states of certain other system components (also called Influence Diagrams).
2.3.3 Dependency Analysis (Dunn, 1984)	Determining what variables depend on what other variables, similar to "influence diagrams".
2.3.4 Qualitative Causal Models (Oyeleye, 1990)	Development of models of the process and event causes of qualitative states and changes of a system.
2.3.5 Look-up Table Generator (NBS 500-93, 1982)	Generating the location within various modules of data and control variables.
2.3.6 Data Dictionary Generator (Ng, 1990)	Generating a defined/used table of locations of all program variables.
2.3.7 Cross-reference List Generator (NBS 500-93, 1982)	Generating the location of all data variables, in form of cross-reference table of modules.
2.3.8 Aliasing Analysis (NBS 500-93, 1982)	Analysis of the aliases of variables used in the main procedure and passed as parameters/arguments to its called procedures (and theirs).
2.3.9 Concurrency Analysis (Rattray, 1990)	Analyzing programs for existing or potential concurrent data-paths and processing.
2.3.10 Database Analysis (Nijssen, 1989)	Checking the implementation of structure, normal form, declarations, and values of a database.
2.3.11 Database Interface Analyzer (NBS 500-93, 1982)	Checking the interface(s) of a program with primary data input for error-detection and handling, consistency-checking, etc.
2.3.12 Data-Model Evaluation (Davis, 1990)	Evaluating the adequacy and features of the data schema or meta-schema used to organize the DB and data structures.

Table 5.2.2-1 (Continued).

Static Testing V&V Methods	Description
2.4 Fault/Failure Analysis	
2.4.1 Failure Mode, Effects, Causality Analysis (FMECA) (MIL-STD-1629A, 1984)	Identification of the failure modes of each system component and analyzing the consequences of each failure type. Information gathered includes failure description, cause(s), defect(s), detection means, resultant safety consequences or other hazards, and recovery methods and conditions.
2.4.2 Criticality Analysis (Wallace, 1989)	Identification of the critical points of failure in a program and the development of test cases to verify their accuracy and robustness.
2.4.3 Hazards/Safety Analysis (Rushby, 1988)	Analysis of failure data to develop metrics and casual hypotheses about system components' failure rates, fault-sources, and future behavior. (BSI89)
2.4.4 Anomaly Testing, (Ng, 1990)	Checking the program for irregularities of style, syntax, or practice, or for signs of potential defects.
2.4.5 Fault-tree Analysis (Event-tree Analysis) (Leveson, 1983)	Beginning with a system hazard or failure, the analysis identifies or hypothesizes immediate and proximal causes, and describes the combination of environment and events that preceded, usually in the form of a directed graph or "and-or tree". Often accompanied by an "event tree analysis" showing relevant event-propagation information.
2.4.6 Failure Modeling (Davis, 1990)	Analysis of the system (and requirements) to identify potential hazards or safety events, or to consider such events determined by dynamic testing. Full analysis involves 2.4.2 in addition.
2.4.7 Common-Cause Failure Analysis (UK, 1989)	Identifying failures that affect apparently indepedent modules.
2.4.8 KB Syntax Checking (Preece, 1991)	Examination of a knowledge base for syntactic errors or anomalies in the composition of rules, frames, and other knowledge elements; no additional external information is required (see below).
2.4.9 KB Semantic Checking (also "Knowledge-Checking"; Stachowitz, 1987)	Use of external meta-rules, constraints, or engineering knowledge to check the internal semantic consistency of a knowledge base.

Table 5.2.2-1 (Continued).

Static Testing V&V Methods	Description
2.4.10 Knowledge Acquisition/Refinement Aid (Desimone, 1990)	Use of an automated tool during knowledge acquisition or refinement (maintenance) to prevent some types of errors from being created in the first place or to assure complete coverage of possible input values, possibly through machine learning techniques.
2.4.11 Knowledge Engineering Analysis (Hart, 1986)	Similar to Knowledge Acquisition/Refinement Aid, but a manual process.
2.5 Inspections	
2.5.1 Informed Panel Inspection (Culbert, 1987)	Convocating a qualified group to review the quality of a program.
2.5.2 Structured Walkthroughs (Fagan, 1986)	An analysis of a program module, usually by programmer, with audience of programming team members.
2.5.3 Formal Customer Review (NUREG/CR-4640, 1987)	A formal evaluation by the customer representatives of the adequacy of a program.

Table 5.2.2-1 (Continued).

Static Testing V&V Methods		Description
2.5.4	Clean-room Techniques (Mills, 1987)	A number of analytic (and procedural) practices for extremely high reliability code production.
2.5.5	Peer Code-checking (Mills, 1987)	Fellow programmers checking each other's code.
2.5.6	Desk Checking (Dunn, 1984)	Inspecting program source code without benefit of automated tools.
2.5.7	Data Interface Inspection (Ng, 1990)	Inspection of all data interfaces of a program for adherence to specification, error-handling, consistency-checking, and other features.
2.5.8	User Interface Inspection (NUREG/CR-4227, 1985)	Inspection of all aspects of the interface to the user and operator for adequacy and other criteria.
2.5.9	Standards Audit (Dunn, 1984)	Evaluation of the program to determine its compliance with the set of governing standards and guidelines.
2.5.10	Requirements Tracing (NBS 500-93, 1982)	Tracing forward from each unique requirement element to specific code modules which are intended to implement that requirement. Requirements which cannot be so mapped are flagged as "unfulfilled requirements." Program elements which are not traceable back to any requirement are flagged as "unintended functions."
2.5.11	Software Practices Review (Humphrey, 1990)	A review of a software organization to advise its management and professionals on how they can improve their operation.
2.5.12	Process Oriented Audits (Humphrey, 1990)	An examination of products of the software development effort (such as unit development folders) with the emphasis on improving the software development process.
2.5.13	Standards Compliance (Bryan, 1988)	Comparison of system to imposed standards (e.g., documentation content, coding style, communication protocols, etc.).
2.5.14	System Engineering Review (DMSC 1990)	A variety of checks and analyses to determine that good sysem engineering principles have been followed in the implementation (see 1.3.3).

Several of the **Data Analysis** methods are concerned with the relationships of named variables in one module to those in other modules (2.3.5-2.3.8). Several methods are concerned with the database or data-model (2.3.10-2.3.12). The remaining methods focus on the flow of data from input to output or one process to another; two of these methods

Some control analysis techniques detect and characterize program control-flow with the sequential and hierarchical aspects (2.2.1-2.2.3, 2.2.5). Two techniques detect the timing aspects (2.2.6-2.2.8). However, one method focuses on the concept of operations, particularly as it involves the user (2.2.4).

Several of the Data Analysis methods are concerned with the relationships of named variables in one module to those in other modules (2.3.5-2.3.8). Several methods are concerned with the database pr data-model (2.3.10-2.3.12). The remaining methods focus on the flow of data from input to output or one process to another; tow o these methods deal with sequential and concurrent processes (2.3.1 and 2.3.9, respectively), while three methods deal with qualitative relationships among system states as determined by data-flow (2.3.2-2.3.4).

The Fault/Failure Analysis methods examine programs for defects, using a general failure-analysis strategy carried over from hardware testing. This approach uses high-level functional and operational descriptions to identify how the system might logically fail. The program code is examined to determine if any of those failure-mode possibilities could logically occur and in what context and under what conditions. Such program examinations often lead to identification of software defects. In any case, they can provide the basis for the development of special failure test cases to be executed with subsequent dynamic testing techniques. Examples of these failure-detecting methods are FMECA (Failure mode, effects, causality analysis; 2.4.1), Criticality Analysis (2.4.2), and Hazards/Safety Analysis (2.4.3). If the notion of failures is relaxed to include any type of apparent defect or anomaly, then the Anomaly Testing method (2.4.4) is also included in this subset. Alternatively, these methods begin with actual (or suspected) program failure data and work backwards seeking specific causes for these failures. Examples of these methods are Fault-tree Analysis (2.4.5), Failure Modeling (2.4.6), and Common-Cause Failure (2.4.7).

Inspections are general examinations of programs. They are less focused on specific program problems than the previous four classes. The first four (2.5.1-2.5.4) involve groups which serve as a critical review audience for a discussion of program code. Clean-room techniques involves a variety of specific defect-exposure techniques, only a few of which involve group review. Peer Code-checking (2.5.5) involves programmers trading code elements among themselves for review, while Desk Checking (2.5.6) typically involves the programmer checking his or her own code. The interface inspection techniques examine the data and the user interfaces (2.5.7 and 2.5.8) respectively. The Standards Audit (2.5.9) examines the code for compliance with governing guidelines and standards. Requirements Tracing (2.5.10) concerns the mapping of requirements to program elements. Software Practices Review and Process Oriented Audits (2.5.11 and 2.5.12) examine the products and process of a software organization. System Engineering Review (2.5.14) involves experts with a system engineering background examining the requirements and design.

5.2.3 Dynamic Testing Methods

The dynamic testing methods were the most discussed techniques for assessing the quality of implemented software systems. In terms of the number of citations and publication pages, dynamic testing methods account for 44% of the 153 methods identified in this report. Table 5.2.3-1 lists and describes these methods.

Table 5.2.3-1 Description of the conventional dynamic testing V&V methods

Dynamic Testing Methods	Description
3.1 GENERAL TESTING	
3.1.1 Unit/Module Testing (Ng, 1990)	General testing of single program modules.
3.1.2 System Testing (Dunn, 1984)	Testing of the overall completed software system with test-cases representative of general program characteristics including its logic and computation, and its timing.
3.1.3 Compilation Testing (Davis, 1990)	Using compiler diagnostics and problem-reports to test system.
3.1.4 Reliability Testing (Boehm, 1981)	Selecting test-cases to exercise particular aspects of the system believed to be unreliable; also extensive testing to assess component failure rates.
3.1.5 Statistical Record-Keeping (Boehm, 1981)	Collecting data on errors discovered for particular system modules to suggest which modules should be tested more thoroughly or even redesigned; especially important in the maintenance phase of the lifecycle.
3.1.6 Software Reliability Estimation (BSI, 1989)	Similar to 2.1.8 but applying sophisticated statistical estimation techniques to fault and error data, to guide continued data collections and to predict system and sub-system failures.
3.1.7 Regression Testing (NBS500-93, 1982)	Repetition of a test suite after program modification to assess effects of changes.
3.1.8 Metric-Based Testing (Jones, 1986)	Selection of some aspect of the program for testing on the basis of the value of some metrics computed for it (usually "complexity").
3.1.9 Ad Hoc Testing (Dunn, 1984)	Test-cases defined at whim by programmer without careful planning.
3.1.10 Beta Testing (Dunn, 1984)	Early release of the system to one or more "beta" user sites for a final testing under realistic field conditions.
3.2 SPECIAL INPUT TESTING	
3.2.1 Random Testing[1] (Barnes, 1987)	Selecting test-cases according to some random statistical procedure.

Table 5.2.3-1 (Continued).

Dynamic Testing Methods	Description
3.2.1.1 Uniform whole program testing	Selecting test-cases such that each input variable is assigned any value inside its range with equal probability, over the whole program domain. (Best of the 4 techniques).
3.2.1.2 Uniform boundary testing	Selecting test-cases, with equal probability, around the boundaries of the ranges of input variables (within the range, at range limit, and outside range).
3.2.1.3 Gaussian whole program testing	Selecting test-cases for input variables according to Gaussian distribution (usually with a mean in the middle of the variable range, and with a standard variation of 1/12 its range).
3.2.1.4 Gaussian boundary testing	Selecting test-cases for input variables drawn from a Gaussian distribution across the boundaries of their valid ranges.
3.2.2 Domain Testing[1] (Beizer, 1990)	Analysis of the boundaries and partitions of the input space and selection of interior, boundary, extreme, and external test-cases as a function of the orthganality, closedness, symetry, linearity, and convexity of the boundaries.
3.2.2.1 Equivalence Partitioning (Myers, 1979)	A type of domain testing which partitions the input domain into equivalence classes such that a test of a representative value from a class is assumed to be a test of all the class values.
3.2.2.2 Boundary-value Testing (Myers, 1979)	A type of domain testing which selects test values at and around (just inside, just outside) input boundaries.
3.2.2.3 Category-Partition Method (Ostrand, 1988)	Similar to equivalence partitioning, but the equivalence classes are more rigorously derived from a functional decomposition, and there is much more attention in selecting representative value to co-occurrence constraints among classes.
3.2.2.4 Revealing Subdomains Method (Weyuker, 1980)	A type of domain testing which takes into account both the partitioning of the overall input space by functional decomposition and also the internal program path structure.

[1] Both of the sub-categories of 3.2 (3.2.1 and 3.2.2) are included as distinct methods as well as their (non-exhaustive) sub-sub-categories.

Table 5.2.3-1 (Continued).

Dynamic Testing Methods	Description
3.3 FUNCTIONAL TESTING	
3.3.1 Specific Functional Requirement Testing (Howden, 1980)	Selecting test-cases to assess the implementation of specific required functions.
3.3.2 Simulation Testing (Pritsker, 1986)	Generating special code to emulate various aspects of code to-be-implemented system.
3.3.3 Model-Based Testing (Davis, 1990)	Use of an analytic or process model of desired function to assess the implemented function.
3.3.4 Assertion Checking (NUREG/ CR4640, 1987)	Bracketing code segments with assertions which can be compiled into executable code to verify assertions during code operation. Similar to the static technique of program proving but involves actual execution of code and assertions.
3.3.5 Heuristic Testing (Miller, L., 1990)	Emphasizes the importance of prior fault prioritization and analysis to determine fault-enabling conditions and appropriate test cases.

Table 5.2.3-1 (Continued).

Dynamic Testing Methods	Description
3.4 Realistic Testing	
3.4.1 Field Testing (Rushby, 1988)	Testing of the program under actual installed conditions.
3.4.2 Scenario Testing (Ng, 1990)	Lab or Field testing with highly ralistic cases or situations.
3.4.3 Qualification/Certification Testing (Jensen, 1979)	Extensive testing to meet some set of high standards of quality or performance.
3.4.4 Simulator-Based Testing (Ng, 1990)	Use of a simulator to generate ealistic input data streams to the system to be tested.
3.4.5 Benchmarking (Mayrhauser, 1990)	Use of standard widely supported tests to exercise a number of aspects of system performance.
3.4.6 Human Factors Experimentation (CHI, 1988)	Evaluation of the human-user performance with the real or simulated system to determine adequacy of the human-computer interface.
3.4.7 Validation Scenario Testing (ASME, 1990)	A realistic dynamic system test which samples important subsets of functional capability. Is usually the last type of testing done,and is intended to provide assurance to the end-user/customer (or regulator). More restrictive in extent and focus than 3.4.2.
3.4.8 Knowledge Base Scenario Generation (Vol. 6 of this report)	A proposed automated method for automatically generating validation scenarios from knowledge bases.
3.5 STRESS TESTING	
3.5.1 Stress/Accelerated Life Testing (NBS500-75, 1981)	Exercising the program as rapidly, with as much data input, CPU tasking, and memory load, as possible.
3.5.2 Stability Analysis (Dunn, 1984)	Choosing test-cases to exercise and stress the stability of the system.
3.5.3 Robustness Testing (Miller, L., 1990)	Testing the program with bizarre inputs under variously degraded conditions.
3.5.4 Limit/Range Testing (Ng, 1990)	Selecting test-cases to test (and exceed) the extreme ranges of allowable limits on variables/ parameters. (Also called Boundary Testing; strategy for automating test-case generation for rule basis given in Miller, 1990).
3.5.5 Parameter Violation (NBS 500-93, 1982)	Determining the various design parameters which led to the present implementation and systematically generating test-cases which violate these parameters.

Table 5.2.3-1 (Continued).

Dynamic Testing Methods	Description
3.6 PERFORMANCE TESTING	
3.6.1 Sizing/Memory Testing (Wallace, 1989)	Assessing the CPU and memory requirements of the program under various conditions.
3.6.2 Timing/Flow Testing (Dunn, 1984)	Assessing the rate of operation (and concurrency) of various program components and the rate of flow of information.
3.6.3 Bottleneck Testing (Ng, 1990)	Determining the location of undesired delays and processing queries in the program's operation.
3.6.4 Queue size, register allocations, paging, etc. (Beizer, 1990)	Assessing any other performance aspect of the program.
3.7 EXECUTION TESTING	
3.7.1 Activity Tracing (Dunn, 1984)	Monitoring and evaluating the results of a particular program function or activity.
3.7.2 Incremental Execution (Ng, 1990)	Halting program execution at multiple points to assess performance variable values and data storage characteristics.
3.7.3 Results Monitoring (NBS 500-93, 1982)	Similar to 1.6.1 but more focused on a particular outcome regardless of the activity that generated it.
3.7.4 Thread Testing (Jensen, 1979)	Following control and data for a single function through multiple modules.
3.7.5 Using Generated Explanations (Miller, 1989)	Examining the explanations or rule traces produced by the expert system to evaluate if the reasoning process is correct.
3.8 COMPETENCY TESTING	
3.8.1 Gold Standard (Rushby, 1988)	Measuring program results against widely accepted standards.
3.8.2 Effectiveness Procedures (Llinas, 1987)	Assessing the sequential effectiveness of the program against some external standard.
3.8.3 Workplace Averages (Rushby, 1988)	Measuring program results against averages established in some workplace.

Table 5.2.3-1 (Continued).

Dynamic Testing Methods	Description
3.9 ACTIVE INTERFACE TESTING	
3.9.1 Data Interface Testing (Ng, 1990)	Testing the data interfaces to insure that all aspects of data I/O are correct, including buffering, change detection, checking, etc.
3.9.2 User Interface Testing (NUREG/CR-4227, 1985)	Evaluation of the user interface from low level ergonomic aspects to instrumentation and controls human factors to global consideration of ease-of-use and appropriateness, taking into account CONOPS and information analyses (below).
3.9.3 Information System Analysis (NUREG/CR-4227, 1985)	Determining that operator-needed information is well organized and is available directly, and quickly, neither too much nor too little, neither inaccurate nor contradictory.
3.9.4 Operational Concept Testing (CONOPS Testing) (Miller, L., 1990)	Testing that the concept of operations is adequate, appropriate, and sufficiently flexible.
3.9.5 Organizational Impact Analysis/Testing (Booher, 1990)	Testing or analyzing the effect of the system on the user organization/corporate structure and/or methods after installation.
3.9.6 Transaction-flow testing (Beizer, 1990)	Identifying the flow of information between people and computers, for user-driven systems, as well as the internal computer processing transformation of that information, and developing a suite of tests to exercise each of the processing steps.

Table 5.2.3-1 (Continued).

Dynamic Testing Methods	Description
3.10 STRUCTURAL TESTING	
3.10.1 Statement Testing (Beizer, 1983)	Generating test-cases to exercise specific (or all) program statements in the source code.
3.10.2 Branch Testing (Miller, E., 1990)	Generating test-cases to exercise branches from conditional or case control structures.
3.10.3 Path Testing (Tung, 1990)	Augmenting branch testing to test various repetitions of program flow through interactive or loop structures of the program.
3.10.4 Call-Pair Testing (Miller, E., 1990)	Developing cases to test the argument and parameter interfaces among programs.
3.10.5 Linear Code Sequence and Jump (LCSAJ) (Miller, E., 1990)	Selecting of test-cases based on control-flow analysis, similar to branch testing, but often used in "lower" level languages such as assembler.
3.10.6 Test-Coverage Analysis Testing (Beizer, 1983)	Determining what statements, paths, branches, etc. are exercised by a set of test-cases.
3.10.7 Conditional Testing (Beizer, 1990)	Testing of statements involving Boolean or Relational (e.g., "A<B") tests. Selecting test-cases corresponding to values equal to, less than, and greater than the values in the conditions.
3.10.8 Data-flow Testing (Beizer, 1990)	Selection of test-cases to explore data anomalies discovered by examination of the program's control flow graph.
3.11 Error-Introduction Testing	
3.11.1 Error Seeding (Dunn, 1984)	Introducing errors of arbitrary kinds in a software system to assess the effects of such errors on system performance.
3.11.2 Fault Insertion (Rushby, 1988)	Introducing modifications to a program which will induce a failure or fault of a particular kind.
3.11.3 Mutation Testing (Ng, 1990)	Introducing errors of various kindsin a program and determining whether a given suite of test-cases detect the errors. Used to assess power of one's testing techniques for discovering problems.

Dynamic testing requires actual execution of system source code on hardware platform under an operating system. When the code is interpreted (as with BASIC), the code execution involves the appropriate interpreter. When the code is compiled before execution, a number of other utilities are also involved, including the link-editor, the loader, and the compiler. The use of these facilities may involve one or more program editors, the creation or invocation of some minimum test-data set, a number of program-execution commands within one or more operating environments, and possibly the use of several application software packages. It is a complex operation to test run a system appropriately. Testers need to design a number of real world test-cases to exercise the system in various ways for various test objectives. For example, testers may wish to ensure that a minimum number of program paths are activated. To cause program control to transfer to a particular path, designated as **P**, requires working backwards from the end of **P** through all preceding decision-points and external-inputs, recording the exact values of data which would cause the program control to ultimately lead to **P**. When intermediate external inputs are required after the initial primary inputs to the program are provided, the testers may be required to force the needed values by writing special data drivers, creating data scripts for the data channel to access, or modifying parts of the tested program to dummy these inputs. In addition, the testers may have to enter pre-determined values in response to program requests.

The testers are also involved in other kinds of program set-up activities. These activities include crafting and seeding of specific types of errors, setting up monitoring software, recording performance and processing results, calculating actual function coverage or structure, interpreting the results, analyzing, planning, and fixing and recording program errors. Tester activities also include maintaining and running special regression test suites to ensure that program fixes are free of side-effects. This testing is done in accordance with an overall test plan which specifies the nature, purpose, and sequence of each designed test-case. Even with effective support tools, dynamic testing is a very complex and labor-intensive activity.[7]

Each of the 67 dynamic testing techniques involves the complex execution activities described above. They have been divided into 11 major categories. The first category, **General Testing** (3.1), is a catch-all for techniques which did not fit into a focused class. Unit/Module Testing (3.1.1) involves intensive testing of the smallest program unit, while System Testing (3.1.2) involves assessing total system functionality and performance over many program interfaces (Table 2.4.3-1). Compilation Testing (3.1.3) involves using a specific language compiler (e.g., the IBM Extended H Fortran Compiler) to check the program for errors. Some systems also have post-compiler checkers, such as SUN Computer Corporation's LINT program (SUN, 1990) which automatically checks for suspected problems and examples of poor programming practices. Reliability Testing (3.1.4) is used to gather information on the "reliability", or the incidence of flaws, of various modules and also to select particular program sections for testing based on prior evidence of unreliability.[8] Statistical Record-Keeping (3.1.5) is a related, more specific technique which involves tracking errors reported for specific modules to build an empirical basis for prioritizing module testing.

---

[7] A further complicating factor is the fact that system source code may take on a wide variety of machine-code compilations depending on the particular compiler, the version of the operating system and the platform used on the day of testing, and so on. These different versions do not usually lead to noticeable functional differences, but there may be significant performance effects (e.g., when an optimizing compiler is or is not used).

[8] The concept of "reliability" when applied to software is fundamentally different when it is applied to hardware. In the latter case, it is quite reasonable to assume various kinds of physical deterioration of the hardware elements over time, as a function of the operating environment. However, software source code, the symbolic representation of a to-be-stored and to-be-activated machine-language program, is impervious to such changes. If a correctly compiled and executed software module produces an "error", then that module has been incorrectly designed and/or implemented in source code. Therefore, the term "unreliability" as applied to software really implies an inherent design error which happens to manifest itself at a certain time under certain test conditions.

Software Reliability Estimation (3.1.6) is the third of the reliability-assessing techniques and employs statistical estimation techniques applied to failure data such as recorded by Statistical Record-Keeping. Regression Testing (3.1.7) involves running a previously-used test-suite on the whole program after modifications have been made to parts of it. This procedure checks to see if the additions affected previous work. Metric-Based Testing (3.1.8) evaluates program units in terms of specially-computed metrics developed to detect modules with high "complexity" indices, which are more likely to fail. Ad Hoc Testing (3.1.9) involves the arbitrary execution of a program without a test plan and without careful consideration of functionality or performance requirements. Unfortunately, this method is highly popular for module testing. Finally, Beta Testing (3.1.10) involves the early release of the system to "beta" user sites for realistic testing.

The second dynamic testing category is **Special Input Testing** (3.2), containing two subclasses. Random Testing (3.2.1) is a technique covering a wide variety of possible means for randomly selecting input test-cases. The four highly-related subcategorized methods under the Random Testing subclass deal with assumptions of four different sampling distributions. This method randomly selects system-input test-cases. In empirically tested detection of seeded and real errors for various test-case samplings using these four subclasses at the Halden STEM project, the Uniform method (3.2.1.1), which selects cases with equal probability, was found to be the most powerful (Barnes, 1987). Domain Testing (3.2.2) is the stalwart technique of seasoned testers and involves detailed analyses of the input space to design a careful selection of test-cases which specifically sample its features. It is itself a specific technique but also has four specialized sub-category methods, each of which has strong proponents. Both Random Testing and Domain Testing are considered separate methods in their own right, since they comprise a variety of means not covered by the listed specializations.

The third category, **Functional Testing** (3.3) uses five methods to assess whether the system exhibits specific functionality as described in the requirements. This category is sometimes called "black box testing" since it focuses on the input/output functionality of the system without taking into account the internal program structure. The most common approach, Specific Functional Requirement Testing (3.3.1), generates test cases from two lines of reasoning. The first approach focuses on a particular function and selects inputs and environment values to produce pre-determined outputs if that function is implemented correctly. It is also known as the customary approach. The second approach selects deviant or marginal input values for a particular function so that this function would not be activated, or so that an error-condition would be returned. This is a lesser-used strategy. Simulation Testing (3.3.2) is a technique often used for systems with particularly complex control or data-input streams. It compares the results of a separately-developed simulated model of the system to the actual system outputs. All discrepancies signal problems in the implemented code. Model-based Testing (3.3.3) is similar but implies a less-detailed, more analytic, qualitative, or mathematical, modeling of a system, usually with much less data-dependency. Assertion Checking (3.3.4) occurs when parts of the code are bracketed with executable assertions about what the code is supposed to do, and the asserted outputs are compared to the actual code-generated ones to test the adequacy of the function. Lastly, Heuristic Testing (3.3.5) involves analysis to identify and prioritize the potential faults of greatest concern and to generate test cases which exercise the system to check for the next-remaining highest-priority fault.

The **Realistic Testing** methods (3.4) test the system under realistic conditions, often in the field (3.4.1), frequently with realistic scenarios (3.4.2), and sometimes with special data-stream simulators (3.4.4). Qualification/Certification Testing (3.4.3) provides for thorough testing under realistically stressful conditions for special widely-used programs. Benchmarking (3.4.5) tests the program against accepted standard test situations, while

Human Factors Experimentation (3.4.6) involves conducting actual performance experiments with human users under realistic conditions. Validation Scenario Testing (3.4.7) and Knowledge Scenario Generation (3.4.8) involve generation and execution of realistic dynamic system tests which sample subsets of important functionality.

Stress Testing (3.5) exercises programs under heavy load conditions (3.5.1), using test cases which assess stability (3.5.2), involve unusual inputs under degraded conditions (3.5.3), test the boundaries of variables (3.5.4), or systematically violate design parameters (3.4.5).

The **Performance Testing** techniques (3.6) focus on various aspects of system performance, including CPU and memory usage (3.6.1), timing characteristics (3.6.2), delays (3.6.3), and queues and paging (3.6.4).

The monitoring methods, **Execution Testing** (3.7), track various aspects of a program's execution (3.7.1-3.7.3). Thread Testing (3.7.4) involves following the path ("thread") of execution of a particular functional capability within and across program modules. Using Generated Explanations (3.7.5) evaluates the correctness of an expert system's reasoning process.

The three **Competency Testing** methods (3.8) all involve constructing test cases to compare software system output to an external standard.

Active Interface Testing (3.9), assesses the data interfaces (3.9.1), the user interfaces (3.9.2), the total information organization and display characteristics (3.9.3), the concept of operations of the system (3.9.4), and the impact of the system's interfaces on the users' organization and procedures (3.9.5). Transaction Flow Testing (3.9.6) is based on the concept of a transaction, a meaningful unit of information-exchange between users and the system, and involves generation of a test suite to exercise these.

Structural Testing (3.10) is the traditional standard of dynamic testing. This class is sometimes called "white box testing". This indicates that the actual program composition and structure is taken into account in generating the test-cases. This contrasts "black box testing" which is concerned only with the external input and output. Conceptually, the simplest structural testing technique is to select test-cases so that each source code statement is executed at least once, as in Statement Testing (3.10.1). Branch Testing (3.10.2) develops test-cases to follow each branch from transfer-of-control structures (e.g., in an **IF-THEN-ELSE** construction, testing the branch that occurs after the **THEN** as well as testing the branch that occurs on the **ELSE** side). Path Testing (3.10.3) is an extension of branch-testing. It involves testing of various numbers of repetitions of program loops. Call-Pair and Linear Code Sequence and Jump testing (3.10.4, 3.10.5) test interfaces among modules and assembly-language control flow. Test-coverage analysis testing (3.10.6) involves setting a goal for any kind of structural coverage. It measures previous test-cases results and selects new test-cases on the basis of what remains to be covered. Conditional testing (3.10.7) tests statements containing Boolean ("A AND B") and relational expressions at the boundaries where these expressions are True and False. Data-flow testing (3.10.8) can follow from branch (or path) testing. This involves a control-flow graph of the program created by reducing all non-branching statements to a single process element. This control-graph is examined for various kinds of suspicious data-flow characteristics.

The last class of structural testing is **Error-Introduction Testing** (3.11). The first two techniques, error seeding and fault insertion involve adding various kinds of errors into the program (3.11.1-3.11.2). The last of these

methods, Mutation Testing (3.11.3), also introduces errors into a program, to assess the sensitivity of the test-case suite. If a test-suite previously used on an unmodified program can detect the inserted problems this test-suite has high level detection capability.

5.3 Discussion

This survey has revealed a large array of testing and V&V techniques available for conventional system software. It might be argued that some methods have been too finely sub-categorized or others should have been more defined. Undoubtedly some important techniques have been omitted, and others should have been omitted. Nonetheless, the availability of numerous methods clearly emerges. Some methods have very precise capabilities and some have very general capabilities. However, most methods are oriented towards a particular aspect of the life-cycle with a few techniques cutting across phases (e.g., reviews, requirements tracing).

The last two sections of this report discuss the tasks of more fully characterizing these methods and of gauging their utility for use with expert systems. Note that if an expert system has a requirements or design phase, the conventional techniques appropriate for testing requirements or design should be applicable to the expert system documents that are created in these phases. This leaves more than a 100 implementation-phase Static and Dynamic testing methods which also might apply to expert system implementations.

Finally, in addition to the methods (and commercial products embodying them) that were identified, there are a number of automated tools and environments that are being used with considerable success for all aspects of testing and V&V, especially in the nuclear industry. They and their descriptions are listed in Table 5.3-1. There is an increasing number of commercial CASE (Computer Aided Software Engineering) tools available to support many aspects of V&V. The review of these tools is beyond the scope of this paper. The reader is referred to published articles and reviews (e.g., CASE Trends) for further discussion of this topic.

Table 5.3-1 CASE tools for full life cycle support

Tools/Methodologies	Description
SAGA (Oakes, 1991)	A French tool for 2167A like lifecycle support, formal specification language, grid for desired programmer practices, promotion of reuse of certified subroutines, used in Design of Display and Control Software (N4 French NPPs).
SPACE (Beltracchi, 1991)	German tool with specification and coding environments, strong semantics, specification language, graphic interface for code specification used for digital safety system design.
CAL - Disassembler (Dahll, 1990)	Translate machine code in CAL (Common Assembler Language). Written in Pascal for analysis.
STAN (Static Analyzer) (Dahll, 1987)	Takes in CAL code, translate into directed graph computer minimal C1 closure (full path coverage). Develop data flows of program variables, single entry single exit analysis, dynamic structural testing analyses. Translates CAL into Pascal.
REMAINDER (Ng, 1990)	A fast, simple and accurate program to record all different instruction-execution paths for a set of test-cases, with much better performance and memory requirements than other methods (linked-tree, binary-tree).
LINT (Sun, 1990)	Sun microsystems program verifier for C programs with checks for post-compiler bugs and violations of good programming practices.
ATRON Evaluator (Beltracchi, 1991)	Runs regression test suite, records results, from Cadre Technologies.
MAT (Beltracchi, 1991)	Maintainability analysis tools, quantifies statement complexity via weightings of elements, reports instances of poor wage and other understandability problems (including unnecessary elements).
OASIS (Beltracchi, 1991)	Workstation based simulator for liquid metal cooled reactor, static, and dynamic analysis tools, safety analysis, analysis of decay heat removal loops (Super Phoenix 1).
MOTH (Barnes, 1987)	Generates various types of test case data for Halden testing compares results of 3 trip code and logs discrepancies (SIM-MOTH generates simulation data) (SOSAT Tool)

Table 5.3-1 (Continued)

Tools/Methodologies	Description
GOMO (Barnes, 1988)	Provides statistical information on (45) inserted faults in Halden TRIP Code testing - number detected/undetected faults, MTBF estimators (SOSAT tool).
SETH (Barnes, 1988)	Runs 6 trip programs, applying test cases, studying common mode factors between them, recording number of failures.
COVER (Barnes, 1988)	Measures branch and statement coverage of test cases for the trip program.
SPADE (Carre, 1986)	Convert source code for PASCAL into an intermediate language (FOL) for various analyses. - dependency analysis - path analysis - symbolic execution for given variables - conformance to pre-and-post-processing conditions
RXVP80 (Carre, 1986)	Static Analysis of Fortran Code, including - data type consistency - subroutine parameter consistency checks - consistency/usage checks on COMMON data - check for inaccessible code - derivation of control flow
LRDA (Liverpool, 1985)	For a given suite of test cases. Computes fractional coverage of statements, branches, linear code sequences, and jumps (in terms of Test Effectiveness Ratios).

6 CHARACTERIZATION OF CONVENTIONAL V&V METHODS

This section builds the framework for characterizing conventional V&V techniques. It begins by developing a taxonomy of defects. Each method is rated by the types of defects it can detect. This classification system will provide the fundamental basis for evaluating the power of the conventional V&V techniques.

The overall utility of particular V&V methods will depend on several factors. Initially, a number of features that relate to the general power and ease-of-use aspects of the techniques are identified in Section 6.2. These features provide the basis for two novel metrics on V&V methods -- "cost-benefit" and effectiveness. These measures are developed and discussed in Section 6.3. Finally, Section 6.4 considers practical issues associated with selecting a set of techniques to use for any system (expert systems or otherwise).

Please note that a new methodology for reviewing and evaluating V&V methods is being proposed. Under this new methodology, individual software V&V techniques are rated on a number of features, then combined in various ways to assess the cost-benefits and the effectiveness of each technique. While this general methodology has been used successfully in other fields, it has not been applied to software testing. Therefore, this approach has yet to receive acceptance among testing professionals. In addition, there is little empirical data detailing the effectiveness of specific techniques. The rating judgments on each V&V technique are subjective. These issues might cause the readers to have questions about the ratings of individual techniques, the derivation of the measures, and the methodology itself. However, since this methodology is clearly set forth, an individual using this approach can easily substitute different values or compute measures in a different way.[9]

6.1 Defect Detection

Most texts on testing provide something of a taxonomy of possible defects (e.g., Boehm, 1981; Dunn, 1984; Beizer, 1990). However, these texts were variously too broad or much too detailed. This was particularly true of the early life-cycle phase descriptions. A classification scheme was therefore developed (Section 6.1.1) to obtain what was believed to be the right level of detail and coverage.

In Section 6.1.2, each V&V technique was rated by the types of defects it could reasonably detect. **Broad Power** describes the range of types of defects detectable by a technique. It is an important feature in the subsequent rating of the cost-benefit value of the techniques (Section 6.3).

6.1.1 A Taxonomy of Defect Types for Conventional Software

Although there are various collections of data on actual defect occurrence (e.g., Beizer, 1990), these data are relatively sparse. Additionally, these studies typically confound programmer skill, development environment, application type, and management. There are also very few predictive theory principles concerning how and where defects occur. The decision of defect importance is dependent upon the system's goals and functions. Finally, there is scant data on the effectiveness of defect detecting techniques, coupled with empirical distribution of defect frequency, to provide a sound basis for a software reliability and detectability analysis. All of these factors make construction of a

[9] All of the ratings and measure computations were developed in a model using a commercial spreadsheet package; all of the tables characterizing the V&V techniques are output from the spreadsheet model.

taxonomy of defects very difficult. The present taxonomy has face and construct validity, but it lacks empirical validity. Nonetheless, it reflects a consensus of views on types of defects and is sufficiently detailed for this task. The reader is reminded that the defects developed here are for conventional software. Defects peculiar to expert systems are not included.[10]

The taxonomy, shown in Table 6.1.1-1, lists 52 types of defects according to the three main Life-cycle phases: requirements, design, and coding. The coding types of defects were further divided into three sub-categories of logic and control, data operations and computations, and other. The two key criteria for defects were: (1) they did not overlap or subsume each other, and (2) they were not specific to a particular language or environment, rather having some commonality across programming languages, development environments, and system development approaches.

Unfortunately, there is a great deal of unevenness among the types of defects; some are broad while some are very specific. Nevertheless, these are all things that software testers and V&V personnel are concerned about, and the presence of any of them could be quite significant.

6.1.2 Detection of Defects by Conventional V&V Methods

In order to estimate the detection capability of the testing techniques, the 153 conventional V&V methods were evaluated against the set of 52 types of defects. The question was whether individual techniques could be expected to detect a specific type of bug. If the answer was "yes", the defect number was entered opposite the technique, and the next defect for that technique was considered. Note that the question being asked here was whether the technique could conceivably detect a particular defect. Not only was this an informed but still subjective assessment, but also it ignored questions of how well or how easily a defect could be found by that method. The results of these subjective judgments are shown in the second column of Table 6.1.2-1. The total number of detectable defect types by a particular method is shown in parentheses in the second column. While every effort was made to be consistent and thorough in these ratings, the reader should use them only as a general guide; abstract ratings in the absence of actually using the method are bound to be variable and differ from one set of raters to another.

While Table 6.1.2-1 shows the most likely detectable defect types of a particular technique, this does not mean that other defects might not be detected by that same technique. On the contrary, any technique, as a side-effect, could theoretically expose any type of defect.

The number of detected defect types range from a low of 3 to a high of 52. The total number of detected defect types is considered to be an index of what is called **Broad Power. Broad Power** is the capability of a technique to detect a broad variety of types of defects. Given the defect taxonomy of Table 6.1.2-1, there are a number of techniques which cover the majority of defects and three which were judged to cover all (2.5.4 Clean-room, 3.4.3 Qualification/Certification testing, and 3.4.4 Simulator-based testing). This is an important aspect in determining the benefits of comparing one technique to another. However, using the number of types of defects detected as an indicator of the Broad Power of a technique is a problem. It implies that all types of defects are equally weighted in importance. This assumption is untrue. In order to improve the measure, a basis for weighing each defect type would have to be developed. Unfortunately, there is no body of data to estimate the average economic consequences of each type of

[10] Nor are they, with any confidence, known.

Table 6.1.1-1 Types of software defects

Type	Description	Occurs
1.0 Requirements	**Originate in Requirements Phase; found in the Requirements Specification**	
.1 Incomplete Decomposition	Failure to adequately decompose a more abstract specification.	System, Sub, Mod
.2 Omitted Requirement	Failure to specify one or more of the next lower levels of abstraction of a higher level specified.	System, Sub, Mod
.3 Improper Translation	Failure to carry detailed requirement through decomposition process, resulting in ambiguity in the specification.	System, Sub, Mod
.4 Operational Environment Incompatibility	Specification which does not accommodate the operational environment, such as data rates, data formats, etc.	System, Sub, Mod
.5 Incomplete Requirement Description	Failure to fully describe all requirements of a function.	Mod
.6 Infeasible Requirement	Requirement which is unfeasible or impossible to achieve given other system factors, e.g., process speed, memory available.	Mod
.7 Conflicting Requirement	Requirements which are pairwise incompatible.	System, Sub, Mod
.8 Incorrect Assignment of Resources	Over-or-Under stating the computing resources assigned to a specification.	Mod
.9 Conflicting Inter-system Specification	Requirements of cooperating systems, or parent/embedded systems, which taken pairwise are incompatible.	System
.10 Incorrect or missing external constants	Specification of an incorrect value or variable, or a missing value or variable in a requirement	Mod
.11 Incorrect or missing description of initial system state	Failure to specify the initial system state, when that state is not equal to 0.	Mod
.12 Overspecification of Requirements	Requirements or specification limits that are excessive for the operational need, causing additional system cost.	System, Sub
.13 Incorrect input or output descriptions	Failure to fully describe system input or output.	Mod

Table 6.1.1-1 (Continued).

Type	Description	Occurs
2.0 Design	**Generated in design and appear in design documentation**	
.1 Omitted requirement	Failure to address a requirement or specification in design.	System, Sub, Mod
.2 Misinterpreted requirement	Failure to accurately represent a requirement or specification in design.	System, Sub, Mod
.3 Data limitation	Failure to accommodate the full range of possible data.	Mod
.4 Unintended Design Element	Inclusion of design elements that cannot be traced to a requirement or specification.	Sub, Mod
.5 Hardware incompatibility	Non-existent or more capable hardware resources prescribed beyond those available, e.g. process cycles or memory.	Mod
.6 Software incompatibility	Assumes commercial package, utilities or operating system capabilities which are not available or function differently than assumed.	Mod
.7 Poor man-machine interface design	Man-machine interface is clumsy, hard to learn/use, hard to see/read, etc.	Sub, Mod
.8 Incorrect analyses of computational error	Design fails to adequately address factors of computation error, such as round-off, truncation, numerical approximation.	Mod
.9 Non-compliance	Design fails to conform to standards.	System, Sub, Mod
.10 Lack of adequate error traps	Failure to provide adequately frequent error traps, or sufficient in scope, or error traps do not provide a recovery mechanism.	Mod
.11 Failure to handle exceptions	Failure to handle unique conditions, or boundary conditions.	Mod
.12 Weak modularity	Design inadequately groups functions or requirements to modules.	Mod
.13 Rigid control structure	Control structure is designed with in-line logic or in other ways which preclude ease of expansion or modification.	Mod
.14 Missing or incorrect processing priorities	Control structure design does not allow processing priorities to be established or modified to satisfy requirements or implements them incorrectly.	System, Sub, Mod
.15 Breakdown between top-level & detail design	Design modifications at one level are not reflected at the other.	Sub, Mod

Table 6.1.1-1 (Continued).

Type	Description	Occurs
3.0 Code	**Originate in code; exclisive of defects regularly detected by an assembler or compiler.**	
3.1 Logic and Control		
.1 Unreachable Code	Code which fails to be accessed due to redundant, contradictory branching conditions.	Mod
.2 Improperly used flow control constructs	Improperly formed or used looping or branching constructs undetected by the compiler.	Mod
.3 Inverted predicates	A predicate (e.g. "if" statement) which is incomplete, transposed or incorrect.	Mod
.4 Improper process sequencing	Processing sequential errors, e.g., attempt to read a file before it is opened and timing/synchronization errors in concurrent processing.	Sub, Mod
.5 Halting problem	A loop or recursive or non-deterministic machine without a meetable exit condition or halting condition.	Mod
.6 Instruction modification	Dynamic instruction modification.	Mod
.7 Failure to save or restore process communication	Failure to save the contents of registers to be used later or restore them upon exit, or to correctly handle interprocess communication mechanisms, e.g., semaphores, file/record locks.	Mod
.8 Unauthorized or incorrect recursion	Code developed to be recursive with a language which doesn't support recursion or improper use of recursion.	Mod
.9 Incorrect labels or control flags	A referenced but uncoded statement label or control flag, or a missing statement label or control flag or unreferenced labels, flags which remain in the code.	Mod

Table 6.1.1-1 (Continued).

Type	Description	Occurs
3.2 Data Operations and Computations		
.1 Missing validity test	Failure to test data imported by a procedure.	Mod
.2 Incorrect data referencing	Conditions which potentially allow a subscript, pointer or index to exceed the boundaries of a declared array or other data structure.	Mod
.3 Mismatched parameter list	Procedure calls where the parameter or argument list of the calling program differs in number or type from that of the called program unit.	Mod
.4 Definition or initialization fault	Failure to initialize or incorrect initialization, or variables used before they are defined.	Mod
.5 Anachronistic data	A mix of data pertinent to the current iteration and data erroneously included from previous iterations.	System, Sub, Mod
.6 Improperly used data handling construct	Errors in use of data handling constructs such as type mismatches, improper transformations, moves or subsetting.	Sub, Mod
.7 Variable misuse	Any misuse of a variable, either locally or globally.	Sub, Mod
.8 Incompatible data representation	Inconsistency in units of data, e.c. pounds, kilograms.	Mod
.9 Insufficient data transport	Poor handling of input and output operations which have an effect on throughput, i.e., input-output statements, library routines or database, indexing.	System, Sub, Mod
.10 Input-Output faults	Incorrect communication protocols and external data mismatches.	System, Sub, Mod
3.3 Other		
.1 Calls to non-existent subprograms	A call to a subprogram that is not yet in the system.	Mod
.2 Improper program linkages	Involving a variable of one data type in a program being declared as another data type in the calling program, or a mismatch in control information between a called and calling program.	System, Sub, Mod
.3 Failure to implement design element	A design element is missing from the code.	System, Sub, Mod
.4 Improperly implemented design element	Code that does not conform to the definition of its corresponding design element.	System, Sub, Mod
.5 Unintended function	"Extra" code that cannot be mapped to any design element.	System, Sub, Mod

Table 6.1.2-1 Capability of testing techniques to detect defects

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)		Types of Defects Detected (from Table 2.3.1-1)
Requirements/Design Methods		
1.1.1	General Requirements Language Analysis/Processing	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.13 (11)[1]
1.1.2	Mathematical Verification of Requirements	1.1, 1.2, 1.3, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13 (11)
1.1.3	EHDM	1.1, 1.2, 1.3, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8-2.11, 2.14, 2.15 (19)
1.1.4	Z	1.1, 1.2, 1.3, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8-2.11, 2.14, 2.15 (19)
1.1.5	Vienna Definition Method	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8-2.11, 2.14, 2.15 (19)
1.1.6	Refine Specification Language	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8-2.11, 2.14, 2.15 (19)
1.1.7	Higher Order Logic (HOL)	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8, 2.9, 2.10, 2.11, 2.14, 2.15 (19)
1.1.8	Concurrent System Calculus	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8, 2.9, 2.10, 2.11, 2.14, 2.15 (19)
1.2.1	Ward-Mellor Method	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8, 2.9, 2.10, 2.11, 2.14, 2.15 (19)
1.2.2	Hatley-Pirbhai Method	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8, 2.9, 2.10, 2.11, 2.14, 2.15 (19)
1.2.3	Harel Method	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.8, 2.9, 2.10, 2.11, 2.14, 2.15 (19)
1.2.4	Extended Systems Modeling Language	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15 (23)

[1] Numbers in parenthesis are sums

Table 6.1.2-1 (Continued).

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)	Types of Defects Detected (from Table 2.3.1-1)
1.2.5 Systems Engineering Methodology	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15 (23)
1.2.6 System Requirements Engineering Methodology	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15 (23)
1.2.7 FAM	1.1, 1.2, 1.3, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.2, 2.4, 2.7, 2.8, 2.9, 2.10, 2.11, 2.12, 2.13, 2.14, 2.15 (23)
1.2.8 Critical Timing/Flow Analysis	1.4, 1.6, 1.8, 2.4, 2.6, 2.7, 3.1.2, 3.1.6, 3.1.9, 3.3.1 (10)
1.2.9 Simulation-Language Analysis	1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.12, 1.13, 2.10-2.12, 2.14 (14)
1.2.10 Petri-Net Safety Analysis	1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.12, 1.13, 2.1, 2.2 (12)
1.2.11 PSL/PSA	1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.12, 2.15 (24)
1.3.1 Formalized Requirements Review	All of 1.0 (13)
1.3.2 Formal Design Review	1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.12 (23)
1.3.3 System Engineering Analysis	1.1-1.10, 1.12, 1.13, 2.1, 2.2, 2.7, 2.9, 2.11-2.15 (21)
1.3.4 Requirements Analysis	All of 1.0 (13)
1.3.5 Prototyping	1.4, 1.5, 1.6, 1.8-1.13, 2.2, 2.3, 2.5-2.7, 2.10-2.15 (20)
1.3.6 Database Design Analysis	1.2, 2.3 (2)
1.3.7 Operational Concept Design Review	1.13, 2.1, 2.2, 2.7 (4)
1.4.1 Requirements Tracing Analysis	1.1, 1.10, 1.11, 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.12 (13)
1.4.2 Design Compliance Analysis	2.4, 2.9, 2.15 (3)
Static Methods	
2.1.1 Analytic Modeling	1.1 thru 1.11, 1.13, 2.1 thru 2.3, 2.5, 2.6, 2.8, 2.9, 2.11, 2.14, 2.15, 3.1.1, 3.2.6, 3.3.1 thru 3.3.4 (28)

Table 6.1.2-1 (Continued).

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)	Types of Defects Detected (from Table 2.3.1-1)
2.1.2 Cause-Effect Analysis	1.1, 1.9, 2.6, 2.8, 3.1.2 (5)
2.1.3 Symbolic Execution	1.13, 2.3, 2.9, 3.1.1-3.1.5, 3.1.9, 3.3.1, 3.3.2 (11)
2.1.4 Decision Tables	1.2, 1.5, 1.13, 2.1, 2.2, 2.4, 2.7, 2.11, 2.13, 2.14, 3.1.2, 3.1.3, 3.1.8, 3.2.6, 3.3.3, 3.3.4, 3.3.5 (17)
2.1.5 Trace-Assertion Method	1.1, 1.2, 1.5, 1.11, 1.13, 2.1, 2.2, 2.11, 2.14, 2.15, 3.1.4, 3.1.5, 3.2.6, 3.3.1, 3.3.2, 3.3.3 (16)
2.1.6 Functional Abstraction	1.1, 1.2, 1.5, 1.11, 1.13, 2.14, 2.15, 3.1.5, 3.1.9, 3.2.6, 3.3.1 thru 3.3.4 (14)
2.1.7 L-D Relation Methods	1.1, 1.2, 1.5, 1.11, 1.13, 2.15, 3.1.3 thru 3.1.6, 3.2.6, 3.3.1 thru 3.3.4 (15)
2.1.8 Program Proving	1.7 thru 1.10, 2.1, 2.6, 2.8, 2.9, 2.11, 2.14, 3.1.1-3.1.6, 3.1.8, 3.2.6 (18)
2.1.9 Metric Analysis	3.1.1 thru 3.1.6, (6)
2.1.10 Algebraic Specification	1.9, 1.11, 1.12, 1.13, 2.1, 2.4, 2.8, 2.15, 3.1.2, 3.1.3, 3.1.4, 3.2.3, 3.2.4, 3.2.6-3.2.9, all of 3.3 (22)
2.1.11 Induction-Assertion Method	1.9, 1.11, 1.12, 1.13, 2.1, 2.4, 2.8, 2.15, 3.1.2, 3.1.3, 3.1.4, 3.2.3, 3.2.4, 3.2.6-3.2.9, all of 3.3 (22)
2.1.12 Confidence Weights Sensitivity Analysis	2.11, 3.2.6 (2)
2.1.13 Model Evaluation	1.1-1.5, 1.7, 1.9-1.11, 1.13, 2.1-2.3, 2.7, 2.11, 2.14, 3.1.4, 3.2.6-3.2.8, 3.2.10, 3.3.4 (22)
2.2.1 Control Flow Analysis	1.11, 2.4, 2.10, 2.13, All of 3.1 (13)
2.2.2 State Transition Diagram Analysis	1.11, 2.4, 2.10, 2.13, All of 3.1 (13)
2.2.3 Program Control Analysis	All of 3.1, 3.3.1, 3.3.2 (11)
2.2.4 Operational Concept Analysis	1.1 thru 1.6, 2.1, 2.2, 2.11, 2.14 (10)
2.2.5 Calling Structure Analysis	1.11, 2.4, 2.10, 2.11, 2.12, 2.13, All of 3.1 (15)
2.2.6 Process Trigger/Timing Analysis	1.6, 1.7, 1.8, 1.9, 1.12, 2.4, 2.7, 2.10, 2.11 (9)
2.2.7 Worst-Case Timing Analysis	1.8, 1.10, 2.3, 2.5, 2.6, 2.7, 2.8, 2.14 (8)

Table 6.1.2-1 (Continued).

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)		Types of Defects Detected (from Table 2.3.1-1)
2.2.8	Concurrent Process Analysis	1.6, 1.7, 1.8, 1.9, 2.5, 2.6, 2.7, 2.14, 3.1.7 (9)
2.3.1	Data Flow Analysis	1.10, 1.11, 1.13, 3.1.4, 3.1.8, All of 3.2 (15)
2.3.2	Signed Directed Graphs	1.6, 1.7, 1.9, 2.5, 2.6, 3.2.2, 3.2.3 (7)
2.3.3	Dependency Analysis	1.6, 1.7, 1.9, 2.5, 2.6, 3.2.2, 3.2.3 (7)
2.3.4	Qualitative Causal Reasoning Analysis	1.6, 1.7, 1.9, 2.5, 2.6, 3.2.2, 3.2.3 (7)
2.3.5	Look-up Table Generator	2.9, All of 3.2 (11)
2.3.6	Data Dictionary Generator	2.9, All of 3.2 (11)
2.3.7	Cross-Reference List Generator	2.9, 3.1.7, 3.1.9, All of 3.2 (13)
2.3.8	Aliasing Analysis	3.2.3, 3.2.4, 3.2.5, 3.2.6, 3.2.7 (5)
2.3.9	Concurrency Analysis	All of 3.1 (9)
2.3.10	Database Analyzer	All of 3.2 (10)
2.3.11	Database Interface Analyzer	All of 3.2 (10)
2.3.12	Data-Model Evaluation	2.3, 2.4, 2.7, 2.8, 2.9, 2.10, 2.11, 2.14 (8)
2.4.1	Failure Mode, Effects, Causality Analysis	1.2, 1.7, 1.8, 2.1, 2.8, 2.10, 2.11, 3.1.2, 3.1.4, 3.1.7, 3.1.8, 3.1.9, all of 3.2 (22)
2.4.2	Criticality Analysis	1.2, 1.9, 2.8, 3.1.4, 3.1.7, 3.1.9, all of 3.2 (16)
2.4.3	Hazards/Safety Analysis	1.1, 1.2, 1.3, 2.1, 2.2, 2.4, 2.7, 2.11, 2.14, 3.1.4, 3.1.9, all of 3.2, 3.3 (26)
2.4.4	Anomaly Testing	2.3, 2.10, 2.11, 3.2.1, 3.3.3, 3.3.4 (6)
2.4.5	Fault-Tree Analysis	1.7, 1.9, 1.12, 2.10, 3.1.2, 3.1.3, 3.1.4, 3.2.1-3.2.4, 3.2.6, 3.2.10 (16)
2.4.6	Failure Modeling	1.2, 1.3, 1.4, 1.6, 1.7, 2.1, 2.3, 2.4, 2.7, 2.11, 2.14, 3.1.4, 3.1.9, all of 3.2, 3.3 (28)
2.4.7	Common-Cause Failure	3.1.2, 3.1.3, 3.1.4, 3.1.5, 3.1.6, 3.1.7, 3.1.8, (7)
2.4.8	Knowledgebase Syntax Checking	1.3, 1.10, 1.11, 1.13, 2.3, 2.11, 2.14, 3.1.1, 3.1.2, 3.1.3, 3.1.5, 3.1.9, 3.2.1-3.2.4, 3.2.6, 3.2.7, 3.2.10, 3.3.3, 3.3.4 (22)
2.4.9	Knowledgebase Semantic Checking	1.3, 1.10, 1.11, 1.13, 2.3, 2.11, 2.14, 3.1.1, 3.1.2, 3.1.3, 3.1.5, 3.1.9, 3.2.1-3.2.4, 3.2.6, 3.2.7, 3.2.8, 3.2.10, 3.3.2, 3.3.3, 3.3.4 (24)

Table 6.1.2-1 (Continued).

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)	Types of Defects Detected (from Table 2.3.1-1)
2.4.10 Knowledge Acquisition/Refinement Aid	1.3, 1.7, 1.10, 1.11, 1.13, 2.3, 2.11, 3.2.6, 3.2.8 (9)
2.4.11 Knowledge Engineering Analysis	1.1-1.3, 1.5, 1.7, 1.10, 1.11, 1.13, 2.1-2.4, 2.7, 2.11, 2.14, 3.2.2, 3.2.4, 3.2.6, 3.2.10, 3.3.4 (20)
2.5.1 Informed Panel Inspection	All of 1.0, 2.0 (28)
2.5.2 Structured Walkthroughs	All of 3.1, 3.2.3, 3.2.6, 3.2.7, 3.3.1, 3.3.2 (14)
2.5.3 Formal Customer Review	All of 1.0, all of 2.0 (28)
2.5.4 Clean-room Techniques	All of 1.0, 2.0, all of 3.0 (52)
2.5.5 Peer Code-Checking	2.7, 2.9, 2.11-2.15, 3.1.1, 3.1.2, 3.1.3, 3.1.9, 3.2.3, 3.2.6, 3.2.7, 3.3.1, 3.3.2 (16)
2.5.6 Desk Checking	2.7, 2.9, 2.11-2.15, 3.1.1, 3.1.2, 3.1.3, 3.1.9, 3.2.3, 3.2.6, 3.2.7, 3.3.1, 3.3.2 (16)
2.5.7 Data Interface Inspection	1.4, 1.7, 1.9, 1.10, 1.11, 2.3, 2.9, 2.11, 3.2.1-3.2.4, 3.2.6, 3.2.8, 3.2.10, 3.3.2 (16)
2.5.8 User Interface Inspection	1.4, 1.8, 2.15 (3)
2.5.9 Standards Audit	2.5, 2.6, 2.8, 2.12, 3.3.4 (5)
2.5.10 Requirements Tracing	1.1, 1.2, 1.3, 1.7, 1.9, 1.10, 1.11, 1.12, 1.13, 2.1, 2.2, 2.4, 2.8, 2.9, 2.15 (15)
2.5.11 Software Practices Review	2.9, 3.3.4 (2)
2.5.12 Process Oriented Audits	2.9, 3.3.3 (2)
2.5.13 Standards Compliance	2.9, 3.2.10 (2)
2.5.14 System Engineering Review	1.1-1.10, 1.12, 1.13, 2.1, 2.2, 2.7, 2.9, 2.11-2.15, 3.1.7, 3.2.1, 3.2.8, 3.2.9, 3.2.10, 3.3.3, 3.3.4 (27)
Dynamic Methods	
3.1.1 Unit/Module Testing	1.2, 1.4, 1.9, 1.10, 2.3, 2.5, 2.6, 2.7, 2.8, 2.10, 2.11, 2.12, 2.13, 2.14, 3.1.2-3.1.9, all of 3.2, 3.3 (37)
3.1.2 System Testing	1.2, 1.4, 1.9, 1.10, 2.3, 2.5, 2.6, 2.7, 2.8, 2.10, 2.11, 2.12, 2.13, 2.14, 3.1.2-3.1.9, all of 3.2, 3.3 (37)

Table 6.1.2-1 (Continued).

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)	Types of Defects Detected (from Table 2.3.1-1)
3.1.3 Compilation Testing	All of 3.1, 3.2.1-3.2.4, 3.2.6, 3.2.10, 3.3.1, 3.3.2 (20)
3.1.4 Reliability Testing	3.1.3, 3.1.4, 3.1.7, all of 3.2, 3.3.2 (14)
3.1.5 Statistical Record-Keeping	3.1.1, 3.1.2, 3.1.5, 3.1.6, 3.1.7, 3.1.9, (6)
3.1.6 Software Reiliability Estimation	3.1.1, 3.1.2, 3.1.5, 3.3.1, 3.3.2 (5)
3.1.7 Regression Testing	1.2, 1.4, 1.9, 1.10, 2.3, 2.5, 2.6, 2.7, 2.8, 2.10, 2.11, 2.12, 2.13, 2.14, 3.1.2-3.1.9, all of 3.2, 3.3 (37)
3.1.8 Metric-Based Testing	3.1.2, 3.1.4, 3.1.8, 3.1.9, 3.2.2, 3.2.10, 3.3.2 (7)
3.1.9 Ad Hoc Testing	3.1.2-3.1.4, 3.1.9, 3.2.1-3.2.4, 3.2.6-3.2.9, 3.3.1 (14)
3.1.10 Beta Testing	1.2, 1.4, 1.8, 1.13, 2.1-2.4, 2.7, 2.8, 2.10, 2.11, 2.14, 3.1.4, 3.1.5, 3.2.5-3.2.10 (21)
3.2.1 Random Input Testing	All of 2.0, all of 3.0 (39)
3.2.2 Domain Testing	All of 2.0, all of 3.0 (39)
3.3.1 Specific Functional Requirement Testing	All of 2.0, all of 3.0 (39)
3.3.2 Simulation Testing	1.2, 1.4, 1.6, 1.7, 1.9, 1.10, 1.11, 1.12, 2.3, 2.6, 2.7, 2.10, 2.14, 3.1.2, 3.1.4, 3.1.5, all of 3.2, 3.3, 3.3.3 (33)
3.3.3 Model-Based Testing	2.1-2.8, 2.10, 2.11, 2.14, 3.1.2, 3.1.4, 3.1.7, 3.2.1, 3.2.4, 3.2.7, 3.2.8, 3.2.10 (19)
3.3.4 Assertion Checking	3.1.3, 3.1.4, 3.1.7, 3.2.1, 3.2.2, 3.2.3, 3.2.4, 3.2.6, 3.2.7, 3.2.10, 3.3.2 (11)
3.3.5 Heuristic Testing	All of 2.0, all of 3.0 (39)
3.4.1 Field Testing	All of 2.0, 3.0 (39)
3.4.2 Scenario Testing	All of 2.0, 3.0 (39)
3.4.3 Qualification/Certification Testing	All of 1.0, 2.0, 3.0 (52)
3.4.4 Simulator-Based Testing	All of 1.0, 2.0, 3.0 (52)
3.4.5 Benchmarking	All of 3.1, 3.2 (19)
3.4.6 Human Factors Experimentation	2.7, 2.13, 2.14, 3.2.10 (4)
3.4.7 Validation Scenario Testing	1.2, 1.4, 1.8, 1.13, 2.1-2.4, 2.7, 2.8, 2.10, 2.11, 2.14, 3.1.4, 3.1.5, 3.2.5-3.2.10 (21)

Table 6.1.2-1 (Continued).

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)	Types of Defects Detected (from Table 2.3.1-1)
3.4.8 Knowledgebase Scenaio Generation	3.1.2, 3.1.4, 3.1.5, 3.1.7, 3.1.9, all of 3.2, 3.3.2-3.3.5 (19)
3.5.1 Stress/Accelerated Life Testing	All of 3.0 (24)
3.5.2 Stability Analysis	All of 3.0 (24)
3.5.3 Robustness Testing	All of 3.0 (24)
3.5.4 Limit/Range Testing	2.3, 2.8, 2.10, 2.11, 3.1.3, 3.1.4, 3.1.7, 3.2.1, 3.2.2, 3.2.8 (10)
3.5.5 Parameter Violation	2.3, 2.8, 2.10, 2.11, 3.1.3, 3.1.4, 3.1.7, 3.2.1, 3.2.2, 3.2.3, 3.2.6, 3.2.8, 3.3.2 (13)
3.6.1 Sizing/Memory Testing	1.6, 1.8, 2.5, 2.7 (4)
3.6.2 Timing/Flow Testing	2.4, 2.7, 3.1.2, 3.1.6, 3.1.9, 3.3.1 (6)
3.6.3 Bottleneck Testing	2.7, 2.14, 3.2.9 (3)
3.6.4 Queue Size, Register Allocations, Paging, Etc.	1.4, 1.5, 1.7, 1.8, 2.7, 2.14, 3.2.9, 3.3.4 (8)
3.7.1 Activity Tracing	3.1.1 thru 3.1.9, 3.2.4 (10)
3.7.2 Incremental Execution	All of 3.1, 3.2, 3.3.1, 3.3.2 (21)
3.7.3 Results Monitoring	2.3, 2.5, 2.6, 2.7, 2.8, 2.9, 2.10, 2.11, 2.14, 3.2, 3.3 (24)
3.7.4 Thread Testing	All of 2.0, 3.1.1-3.1.5, 3.2.1-3.2.4, 3.2.6-3.2.10, 3.3.1-3.3.5 (34)
3.7.5 Using Generated Explanations	3.1.4, 3.1.7, 3.2.1, 3.2.2, 3.2.4, 3.2.6-3.2.8, 3.2.10, 3.3.3-3.3.5 (12)
3.8.1 Gold Standard	2.1-2.9, 2.11, 2.14, 3.1.5, all of 3.2, 3.3.3-3.3.5 (25)
3.8.2 Effectiveness Procedures	2.1-2.9, 2.11, 2.14, 3.1.5, all of 3.2, 3.3.3-3.3.5 (25)
3.8.3 Workplace Averages	2.1-2.9, 2.11, 2.14, 3.1.5, all of 3.2, 3.3.3-3.3.5 (25)
3.9.1 Data Interface Testing	1.4, 1.13, 2.1, 2.2, 2.3, 2.9, 2.10, 2.11, 2.14, 3.1.5, 3.1.7, 3.2.1, 3.2.8, 3.2.9, 3.3.3 (15)
3.9.2 User Interface Testing	1.1-1.7, 2.1-2.4, 2.7, 2.10, 2.11, 2.14, 3.1.5, 3.1.7, 3.2.8 (18)
3.9.3 Information System Analysis	1.1-1.7, 2.1-2.4, 2.7, 2.10, 2.11, 2.14, 3.1.5, 3.1.7, 3.2.8 (18)

Table 6.1.2-1 (Continued).

Conventional V&V Testing Technique (from Tables 2.2.1-1, 2.2.2-1, and 2.2.3-1)	Types of Defects Detected (from Table 2.3.1-1)
3.9.4 Operational Concept Testing	1.4, 2.1, 2.2, 2.7, 3.2.10 (5)
3.9.5 Organizational Impact Analysis/Testing	1.1 thru 1.4,1.10, 1.12, 1.13, 2.1, 2.2, 2.3, 2.8, 2.9, 2.11, 2.14, 2.15 (15)
3.9.6 Transaction-Flow Testing	All of 2.0, 3.0 (39)
3.10.1 Statement Testing	All of 3.0 (24)
3.10.2 Branch Testing	All of 3.0 (24)
3.10.3 Path Testing	All of 3.0 (24)
3.10.4 Call-Pair Testing	3.2.3, 3.2.8, 3.3.2 (3)
3.10.5 Linear Code Sequence and Jump	3.1.5, 3.1.8, 3.2.10, 3.3.2 (4)
3.10.6 Test-Coverage Analyzer	All of 3.0 (24)
3.10.7 Conditional Testing	3.1.3, 3.1.4, 3.2 (12)
3.10.8 Data-Flow Testing	All of 2.0, 3.0 (39)
3.11.1 Error Seeding	3.1.3, 3.1.4, 3.1.7, 3.1.9 (4)
3.11.2 Fault Insertion	All of 3.1, 3.2 (19)
3.11.3 Mutation Testing	All of 3.1, 3.2 (19)

defect,[11] if a reasonable "importance" metric were developed, it could easily be incorporated into the derivation of the Broad Power index.

Using the data in Table 6.1.2-1, one could ask how many techniques can detect each type of defect. This question assesses whether there are sufficient alternative means of detecting a particular type of defect. In turn, this indicates how critical it is to use a particular technique. That is, if finding a particular type of defect is considered to be highly important, but only one or two techniques can detect it, then it is critical that one or both of these techniques be included in the V&V plan. The results of this inversion of the technique/defect data are shown in Table 6.1.2-2. The defect with the most covering methods is the Design defect 2.11, Failure To Handle Exceptions, with 35 techniques judged to be able to detect this flaw. The least covered defect is the Code defect 3.1.6, (Dynamic) Instruction Modification, with 10 applicable techniques. On average, each defect can be detected by 21 techniques. These findings suggest that the identified 153 methods, taken together, do fairly well in covering the total set of system defects.

6.2 Definition of the Cost and Benefit Factors Evaluation of Conventional Technique Effectiveness

Seven additional factors in addition to Broad Power are now introduced to characterize the costs and benefits (and other measures) of V&V techniques. The four factors which define the benefits all measure the **power** aspects of each technique.

1) **Broad Power** was defined in the previous section as a function of the number of different defects detectable by the technique.

2) **Hard Power** is a judgment about the capability of the technique to detect hard problems; problems which are not at all obvious on inspection. These problems may also be intermittent because they depend on non-obvious aspects of the running context,or they are enabled by various obscure means.

3) **Formalizability** assesses the extent to which a technique lends itself to formal calculus or algebraic representations of the specification, design, or implemented system. This would allow automated theorem-provers (if developed) to detect anomalies, contradictions, inconsistencies, etc.

4) **HCI Testability** characterizes whether the Human-Computer Interface is directly testable using the technique. Since the applications of strongest interest are decision-support type systems to help users process and interpret information as well as to advise them on alternative actions, the HCI is an important aspect to be tested. Techniques that test the HCI are seen as having higher power.

[11] By "economic consequences" is meant all the costs associated with an error being present in a system and actually occurring. Such costs will include costs to identify, locate, and repair the problem as well as any costs due to impact on safety or loss of capability.

Table 6.1.2-2 Applicability of conventional tchniques to defects in conventional software

Software Defects Type	Number of Test Techniques Applicable	Ranking of Defect Coverage[1]
1.0 REQUIREMENTS		
.1 Incomplete decomposition	15	39
.2 Omitted requirement	23	18
.3 Improper translation	19	30
.4 Operational environment incompatibility	17	35.5
.5 Incomplete requirement description	13	47
.6 Infeasible requirement	14	43.5
.7 Conflicting requirement	17	35.5
.8 Incorrect assignment of resources	13	47
.9 Conflicting inter-system specification	15	39
.10 Incorrect or missing external constants	19	30
.11 Incorrect or missing description of initial system state	18	33.5
.12 Overspecification of requirements	12	49.5
.13 Incorrect input or output description	18	33.5

Table 6.1.2-2 (Continued).

Software Defects Type	Number of Test Techniques Applicable	Ranking of Defect Coverage[1]
2.0 DESIGN		
.1 Omitted requirement	23	18
.2 Misinterpreted requirement	27	10.5
.3 Data limitation	28	8.5
.4 Unintended design element	21	22
.5 Hardware incompatibility	14	43.5
.6 Software incompatibility	15	39
.7 Poor man-machine interface	29	6.5
.8 Incorrect analyses of computational error	20	25.5
.9 Noncompliance	21	22
.10 Lack of adequate error traps	23	18
.11 Failure to handle exceptions	35	1
.12 Weak modularity	13	47
.13 Rigid control structure	14	43.5
.14 Missing or incorrect processing priorities	34	2
.15 Breakdown between top-level & detail design	14	43.5

Table 6.1.2-2 (Continued).

Software Defects Type	Number of Test Techniques Applicable	Ranking of Defect Coverage[1]
3.0 CODE		
3.1 Logic and Control		
.1 Unreachable code	12	49.5
.2 Improperly used flow control constructs	20	25.5
.3 Improper predicates	19	30
.4 Improper process sequencing	25	14
.5 Halting problem	24	15
.6 Instruction modification	10	52
.7 Failure to save or restore process communication data	20	25.5
.8 Unauthorized or incorrect recursion	11	51
.9 Labels or control flags	19	30
3.2 Data Operations and Computations		
.1 Missing validity test	28	8.5
.2 Incorrect data referencing	27	10.5
.3 Mismatched parameter list	26	12.5
.4 Definition or initialization fault	26	12.5
.5 Anachronistic data	19	30
.6 Improperly used data handling construct	33	3
.7 Variable misuse	29	6.5
.8 Incompatible data representation	31	5
.9 Insufficient data transport	23	18
.10 Input-output faults	32	4
3.3 Other		
.1 Calls to non-existent subprograms	15	39
.2 Improper program linkages	20	25.5
.3 Failure to implement design element	21	22
.4 Improperly implemented design element	23	18
.5 Unintended function	15	39

1 = defect is covered by the most number of techniques 52 = defect is covered by the least number of techniques

The four factors which are used here to define the costs all measure the **ease-of-use** aspects of each technique.

1) **Ease of Mastery** is the ease with which a technique can be taught, understood, and applied. It is measured in terms of the educational or professional level required to deal with the technique concepts and the amount of training time needed to teach the concepts. These issues all impact cost.

2) **Ease of Setup** refers to the labor, time, and resources required to have the V&V technique ready to be applied to the program.

3) **Ease of Running/Interpretation** measures the ease (or difficulty) of actually applying the technique and interpreting the findings.

4) **Usage** is the extent to which the technique is generally and commonly used. The inference is that the higher the usage the greater the general ease-of-use of the technique, or the easier it might be to get approval to use it. This factor also reduces costs.

An inverse relationship is assumed between the ease-of-use factors and cost: the greater the ease-of-use, the lower the costs associated with the technique.

It is believed that the ease-of-use factors are general and applicable across many application domains, levels of system complexity, and implementation approaches. These were chosen to reflect the problems inherent in testing the most complicated and important system; that is, a system having the highest levels of complexity and required integrity, as discussed in Section 2.4.3. **Hard Power** emphasizes the capability to address extremely complicated systems and to find the most difficult of types of defects within them; such types of defects rarely occur in simple systems. **Formalizability** favors formal approaches that permit mathematical reasoning about the presence of defects and anomalies, and **HCI Testability** emphasizes techniques and systems that concentrate on user interactions and the system. These biases will restrict designations of greatest power to those techniques which are inherently formalizable and are designed to assess complex systems with a high degree of human-computer interaction. Selective though this may be, it is believed that these are exactly the kinds of techniques and systems which must be emphasized in this review.

All eight factors are associated with a five-point rating scale (1-5), with 1 representing the least value (power or ease-of-use), and 5 representing the greatest value. The interpretation of each of the five values for each of the eight factors is given in Table 6.2-1. For Broad Power, the scale is based on percentage of applicable defects detectable. For example, Requirements and Design Methods can conceivably detect 28 defects. A Broad Power of 1 indicates that 0-6 defects are detectable (0-23%). The assumption is that all eight factors are based on continuous underlying distributions of values, and that these distributions are of the same kind (e.g., Normal). It is also assumed that the definitions of the five points of the eight scales are partitioned into five equal parts. These assumptions make it reasonable to compare values across different factors. That is, a Hard Power value of 4 is comparable in degree to a

Table 6.2-1 Interpretation of the 1-5 rating scale values for each of the eight cost/benefit factors

Factors	Rating Scale Values	
	Value	Explanation
Power Factors		
Broad Power	1	0-23% of Applicable defects detectable
	2	24-44% of Applicable defects detectable
	3	45-65% of Applicable defects detectable
	4	66-86% of Applicable defects detectable
	5	87-100% of Applicable defects detectable
Hard Power	1	Not good for finding hard defects
	2	Good at finding a few hard defects
	3	Good at finding several kinds of hard defects
	4	Very good at finding a number of hard defects
	5	Excellent at finding a wide number and variety of difficult defects
Formalizability	1	Not really possible
	2	Partially feasible, but requires extensive effort
	3	Feasible to formalize simply in small software systems
	4	Very feasible
	5	Been done at least once or else designed completely
Human-Computer Interaction Tested	1	Not really at all
	2	Somewhat, as a side effect
	3	Tests some aspects OK
	4	Quite thorough in testing HCI
	5	A primary focus of the technique

Table 6.2-1 (Continued).

FACTORS	Rating Scale Values	
	Value	Explanation
Ease-of-Use Factors		
Ease of Mastery	1	Very difficult, requires specialized mathematical or programming skills and then specialized training
	2	Quite difficult, but required skill level and training is somewhat less
	3	Requires concentrated training, but most people can acquire the method without difficulty
	4	Requires only a little training, almost everyone can acquire the skill
	5	Requires virtually no training, anybody can pick it up while using it after a few minutes
Ease of Setup	1	Have to do a considerable amount of programming or specification in some language, days to weeks
	2	Takes a significant amount of programming or specification to set up, a day or so
	3	Takes a moderate amount of time and thought, several hours
	4	Sets up rapidly in an hour or so
	5	Set-up is almost immediate
Ease of Run/ Interpretation	1	Very complicated to run, and the findings take quite a while to interpret, several hours to days
	2	Difficult to run, interpretation requires detailed analysis over a number of hours
	3	Requires some time and care to run; interpretation requires several hours
	4	Quite easy to run, interpretation is accomplished within a few minutes
	5	Completely easy to run, interpretation is almost immediately made
Usage	1	Almost nobody uses it or is even familiar with it
	2	Used by a few, some people have heard of it
	3	Used by quite a few, most people have heard of it
	4	Used by a majority, almost everyone has heard of it
	5	Highly familiar to all and almost always used, no matter what other techniques are employed

Usage value of 4. These assumptions also justify combining the scores in various arithmetic formulas to derive the "cost-benefit" and "effectiveness" metrics discussed in the next section.[12]

6.3 Evaluating "Cost-Benefit" and "Effectiveness" of Conventional V&V Methods

In this section, "metrics" for comparing techniques are developed. Table 6.3-1 lists all 153 methods, with the identification number and technique name given in columns B and C, respectively. The authors rated each of these techniques by using the previously described eight factors (Broad Power, Hard Power, Formalizability, Human-Computer Interface (HCI) Testability, Ease of Mastery, Ease of Setup, Ease of Running/Interpretation, and Usage). These ratings are based on the authors' judgments of the individual technique's capabilities, personal software experiences, and extensive review of software literature. The results are shown in columns E-H and K-N of Table 6.3-1. The primary objective of these ratings is to develop reasonable **Cost-Benefit** and **Effectiveness** measures for each technique, given in column S and columns T-V, respectively. This is achieved by combining the eight rating factors into single scores so that the higher score indicates a higher "cost-benefit" or more "effective" technique.

As will be demonstrated, these metrics were designed to be adaptable to the user's purpose by changing either the method of computing the metrics themselves (Sections 6.3.1 and 6.3.2.1) or by changing the various weights associated with them (Section 6.3.2.2). That is, if a user believes that a power rating or an ease-of-use measure should be changed, then the user should simply make that change in the table (and recompute the value that depends on the value changed). Similarly, if a user wishes to modify the definition of either the cost-benefit or the effectiveness measure, this can be accomplished by substituting a new computation for combining the values of the eight factors (or new factors could be added). Finally, the emphases given to the various factors under increasing need for V&V, Classes 3 through 1, are explicitly expressed as a set of numerical weights. A user can easily modify these also if a different set of emphases is preferred.

The complexity and required integrity of the system to be tested by V&V methods are taken into account in computing the **Effectiveness** measures for the various techniques. The three Classes of V&V developed in Section 2.4.3 were used to represent the stringency with which V&V techniques need to be applied and the effectiveness required of them. For example, the V&V Class 1 systems require the greatest capability for hard power, formalizability, and human-computer interaction quality. Since these systems are the most complex and require the highest integrity, the weighing of these three factors is increased relative to the other five factors. However, V&V Class 3 systems are different. The ease of use and broad power are more important; therefore, the weights for these factors were increased (Section 6.3.2.2).

6.3.1 A Simple Cost-Benefit Metric

It is proposed that the four power measures generally assess the "benefits" of the technique, while the ease-of-use factors indirectly address "costs". The proposed metric simply subtracts a measure of the total costs from a measure of the total benefit. This measure can be expressed, generally, as follows, where benefits are represented by the power factors and costs are represented by the "difficulty of use" factors discussed below:

---

[12] The authors have made these assumptions to justify the mathematics involved in the cost-benefit and, later, effectiveness measures. However, there is no empirical data one way or the other. Another implicit assumption is that the factors are independent and uncorrelated. Although this is somewhat unlikely, the authors feel that the factors are sufficiently independent to assess distinct underlying aspects of the techniques.

Table 6.3-1: Conventional V and V Techniques\ Power and Ease-of-Use Factor Ratings\Cost-Benefit and Effectiveness Measures

	B	C	E	F	G	H	I	K	L	M	N	O	Q	S	T	U	V
1		TECHNIQUE	P	O	W	E	R			E A S E	O F	U S E	Difficulty	COST-		E F F E C T I V E N E S S :	
2	Techniqu		P1	P2	P3	P4	TOTAL	E1	E2	E3	E4	TOTAL	of	BENEFI	V&V	V&V	V&V
3	Number	NAME	Hard	Broad	Formal	HCI	POWER	Learn	Setup	un/inter	Usage	EASE	use	MEASURE	Class 3	Class 2	Class 1
4																	
5	1.0	REQS./DESIGN Methods															
6	1.1	FORMAL METHODS															
7	1.1.1	Gen'l Reqs. Lang. Analysis	2	4	3	2	11	2	4	2	2	10	14	-3	275	263	246
8	1.1.2	Mathematical Verification	3	4	4	1	12	1	2	1	2	6	18	-6	218	247	260
9	1.1.3	EHDM	3	4	4	1	12	1	1	2	1	5	19	-7	197	235	265
10	1.1.4	Z	3	4	4	1	12	1	1	1	1	4	20	-8	185	225	252
11	1.1.5	Vienna Definition Method	3	4	4	1	12	1	1	1	1	4	20	-8	185	225	252
12	1.1.6	Refine Specification Language	4	4	4	2	14	2	2	2	2	8	16	-2	260	300	324
13	1.1.7	Higher Order Logic	2	4	4	2	12	1	2	1	1	5	19	-7	203	220	236
14	1.1.8	Concurrent System Calculus	3	4	4	1	12	1	1	1	1	4	20	-8	185	225	252
15	1.2	SEMI-FORMAL METHODS															
16	1.2.1	Ward-Mellor Method	4	4	4	2	14	2	2	2	3	9	15	-1	280	312	327
17	1.2.2	Hatley-Pirbhai Method	3	4	4	2	13	2	2	1	2	7	17	-4	243	265	277
18	1.2.3	Harel Method	3	4	4	2	13	2	1	2	2	7	17	-4	242	265	285
19	1.2.4	Extended Sys. Model. Lang	3	4	4	2	13	1	2	2	1	6	18	-5	220	255	283
20	1.2.5	Sys. Eng. Methodology	3	4	4	2	13	1	1	2	1	5	19	-6	207	245	278
21	1.2.6	Sys. Req. Eng. Method.	4	4	4	2	14	3	2	2	3	10	14	0	295	320	331
22	1.2.7	FAM	3	4	4	1	12	1	2	2	2	7	17	-5	230	257	273
23	1.2.8	Critical Timing/Flow Analysis	4	2	2	2	10	2	1	3	2	8	16	-6	209	250	276
24	1.2.9	Simulation-Language Analysis	4	3	4	3	14	2	2	3	2	9	15	-1	258	302	342
25	1.2.10	Petri-Net Safety Analysis	3	2	4	1	10	1	2	2	2	7	17	-7	182	221	257
26	1.2.11	PSL/PSA	2	4	3	2	11	1	1	2	2	6	18	-7	221	225	227
27	1.3	REVIEWS AND ANALYSES															
28	1.3.1	Formal Requirements Review	2	5	1	3	11	5	4	4	4	17	7	4	416	345	271
29	1.3.2	Formal Design Review	2	4	1	3	10	5	4	4	4	17	7	3	392	327	263
30	1.3.3	System Engineering Analysis	3	4	1	3	11	1	3	3	2	9	15	-4	272	276	257
31	1.3.4	Requirements Analysis	3	5	2	3	13	4	4	4	4	16	8	5	407	369	321
32	1.3.5	Prototyping	2	4	2	4	12	3	2	4	3	12	12	0	327	296	275
33	1.3.6	Database Design Analysis	3	1	1	1	6	3	2	4	3	12	12	-6	229	230	226
34	1.3.7	Operational Concept Design Review	3	1	1	4	9	3	3	3	3	12	12	-3	260	260	257
35	1.4	TRACEABILITY ANALYSES															
36	1.4.1	Requirements Tracing	2	3	1	3	9	3	4	3	4	14	10	-1	326	283	234
37	1.4.2	Design Compliance Analysis	2	1	1	2	6	3	3	2	2	10	14	-8	203	193	181
38																	
39																	
40	2.0	STATIC TESTING METHODS															
41	2.1	ALGORITHM ANALYSIS															
42	2.1.1	Analytic Modeling	2	3	1	1	7	2	2	2	1	7	17	-10	193	189	172
43	2.1.2	Cause-effect Analysis	2	1	1	1	5	2	2	2	2	8	16	-11	165	165	159
44	2.1.3	Symbolic Execution	3	2	5	1	11	1	1	2	1	5	19	-8	150	206	269
45	2.1.4	Decision Tables	3	3	3	2	11	2	3	2	3	10	14	-3	263	272	270
46	2.1.5	Trace-assertion Method	4	2	4	2	12	2	1	1	1	5	19	-7	167	232	287
47	2.1.6	Functional Abstraction	2	2	4	1	9	2	1	1	1	5	19	-10	147	172	206
48	2.1.7	L-D Relation Method	3	2	4	2	11	2	1	1	1	5	19	-8	162	207	253
49	2.1.8	Program Proving	4	2	4	1	11	1	2	1	1	5	19	-8	155	224	275
50	2.1.9	Metric Analyses	1	1	3	1	6	3	3	3	3	12	12	-6	222	194	190

Table 6.3-1: Conventional V and V Techniques\ Power and Ease-of-Use Factor Ratings\Cost-Benefit and Effectiveness Measures

	B	C	E	F	G	H	I	K	L	M	N	O	Q	S	T	U	V
1		TECHNIQUE	P	O	W	E	R			E A S E	O F	U S E		Difficulty	COST-	E F F E C T I V E N E S S :	
2	Techniqu		P1	P2	P3	P4	TOTAL	E1	E2	E3	E4	TOTAL		of	BENEFI	V&V	V&V	V&V
3	Number	NAME	Hard	Broad	Formal	HCI	POWER	Learn	Setup	un/inter	Usage	EASE		use	MEASURE	Class 3	Class 2	Class 1
4																		
51	2.1.10	Algebraic Specification	3	2	4	1	10	1	2	1	1	5		19	-9	150	199	241
52	2.1.11	Induction-Assertion Method	3	2	4	1	10	1	1	1	1	4		20	-10	137	189	236
53	2.1.12	Confidence Weights Sensitivity Analysis	2	1	3	1	7	3	2	1	2	8		16	-9	170	177	190
54	2.1.13	Model Evaluation	3	3	2	2	10	2	2	2	1	7		17	-7	209	231	239
55	2.2	CONTROL ANALYSIS																
56	2.2.1	Control Flow Analysis	2	2	2	2	8	3	4	3	5	15		9	-1	313	274	236
57	2.2.2	State Transition Diagram	3	2	3	3	11	2	3	3	3	11		13	-2	261	274	288
58	2.2.3	Program Control Analysis	3	1	2	2	8	3	3	3	5	14		10	-2	281	271	257
59	2.2.4	Operational Concept Analysis	3	1	1	5	10	4	3	3	3	13		11	-1	285	278	274
60	2.2.5	Call Structure Analys	3	2	2	2	9	4	3	3	5	15		9	0	320	297	269
61	2.2.6	Process Trigger/Timing Analysis	4	1	2	2	9	2	2	3	3	10		14	-5	218	254	276
62	2.2.7	Worst-case Timing Analysis	2	1	3	1	7	2	2	3	2	9		15	-8	179	189	212
63	2.2.8	Concurrent Process Analysis	4	1	2	2	9	2	2	3	1	8		16	-7	178	230	270
64	2.3	DATA ANALYSIS							0			0						
65	2.3.1	Data Flow Analysis	3	2	3	2	10	2	1	3	4	10		14	-4	245	256	268
66	2.3.2	Signed Directed Graphs	3	1	2	2	8	2	3	3	2	10		14	-6	206	227	244
67	2.3.3	Dependency Analysis	2	1	3	2	8	2	3	4	3	12		12	-4	234	231	246
68	2.3.4	Qualitative Causal Analysis	3	1	3	2	9	2	3	3	2	10		14	-5	207	234	264
69	2.3.5	Look-up Table Generator	2	1	2	1	6	4	5	4	4	17		7	-1	299	262	234
70	2.3.6	Data Dictionary Generator	2	1	2	1	6	4	4	4	4	16		8	-2	286	252	229
71	2.3.7	Cross-reference List Generator	2	2	2	1	7	4	5	4	4	17		7	0	323	280	242
72	2.3.8	Aliasing Analysis	3	1	2	2	8	4	4	3	2	13		11	-3	249	253	257
73	2.3.9	Concurrency Analysis	2	1	2	1	6	2	2	2	1	7		17	-11	146	160	176
74	2.3.10	Database Analysis	2	1	2	1	6	2	3	3	3	11		13	-7	211	204	200
75	2.3.11	Database Interface Analysis	2	1	2	1	6	3	4	3	3	13		11	-5	239	222	209
76	2.3.12	Data-Model Evaluation	3	1	2	1	7	2	3	3	2	10		14	-7	196	217	231
77	2.4	FAULT/FAILURE ANALYSIS																
78	2.4.1	Failure-mode Effects Caus. Analysis	5	2	1	2	10	2	2	2	2	8		16	-6	214	268	282
79	2.4.2	Criticality Analysis	4	2	1	2	9	2	2	2	2	8		16	-7	209	243	248
80	2.4.3	Hazards/Safety Analysis	3	3	1	2	9	2	2	2	3	9		15	-6	248	248	225
81	2.4.4	Anomaly Testing	3	1	3	3	10	2	2	3	3	10		14	-4	224	246	275
82	2.4.5	Fault-Tree Analysis	4	2	1	2	9	2	2	2	2	8		16	-7	209	243	248
83	2.4.6	Failure Modeling	4	3	2	2	11	3	2	2	1	8		16	-5	229	264	277
84	2.4.7	Common-cause Failure Analysis	3	1	2	2	8	3	2	2	2	9		15	-7	196	215	230
85	2.4.8	Knowledgebase Syntax Checking	4	2	5	2	13	4	3	3	2	12		12	1	268	307	354
86	2.4.9	Knowledgebase Semantic Checking	5	3	5	3	16	4	2	4	1	11		13	3	286	348	414
87	2.4.10	Knowledge Acquisition/Refinement Aid	2	1	4	2	9	3	4	2	1	10		14	-5	199	212	243
88	2.4.11	Knowledge Engineering Analysis	3	2	3	2	10	2	2	2	2	8		16	-6	206	232	254
89	2.5	INSPECTIONS																
90	2.5.1	Informed Panel Inspection	3	3	1	2	9	5	4	4	3	16		8	1	343	312	273
91	2.5.2	Structured Walk-throughs	3	2	1	2	8	4	5	4	4	17		7	1	337	308	269
92	2.5.3	Formal Customer Review	3	3	1	2	9	5	4	4	4	17		7	2	363	324	276
93	2.5.4	Clean-room Techniques	5	5	2	2	14	2	3	2	2	9		15	-1	300	339	331
94	2.5.5	Peer Code Checking	3	2	1	2	8	4	5	4	4	17		7	1	337	308	269
95	2.5.6	Desk Checking	2	2	2	2	8	4	5	5	5	19		5	3	365	312	271
96	2.5.7	Data Interface Inspection	3	2	1	3	9	5	4	4	3	16		8	1	329	304	278

Table 6.3-1: Conventional V and V Techniques\ Power and Ease-of-Use Factor Ratings\Cost-Benefit and Effectiveness Measures

	B	C	E	F	G	H	I	K	L	M	N	O	Q	S	T	U	V
1		TECHNIQUE	P	O	W	E	R			E A S E	O F	U S E	Difficulty	COST-	E F F E C T I V E N E S S :		
2	Techniqu		P1	P2	P3	P4	TOTAL	E1	E2	E3	E4	TOTAL	of	BENEFI	V&V	V&V	V&V
3	Number	NAME	Hard	Broad	Formal	HCI	POWER	Learn	Setup	un/inter	Usage	EASE	use	MEASURE	Class 3	Class 2	Class 1
4																	
97	2.5.8	User Interface Inspection	3	1	1	5	10	4	4	4	4	16	8	2	330	310	295
98	2.5.9	Standards Audit	2	1	1	2	6	3	3	3	3	12	12	-6	235	215	197
99	2.5.10	Requirements Tracing	3	2	2	3	10	4	3	3	4	14	10	0	310	295	279
100	2.5.11	Software Practices Review	1	1	1	1	4	3	3	3	3	12	12	-8	220	180	150
101	2.5.12	Process Oriented Audits	1	1	1	1	4	4	3	3	3	13	11	-7	235	188	154
102	2.5.13	Standards Compliance	2	1	3	2	8	3	3	3	3	12	12	-4	237	229	237
103	2.5.14	System Engineering Review	3	3	1	3	10	1	3	3	2	9	15	-5	248	258	249
104																	
105																	
106	3.0	DYNAMIC TESTING METHODS															
107	3.1	GENERAL TESTING															
108	3.1.1	Unit/Module Testing	2	4	2	1	9	5	2	4	4	15	9	0	347	294	247
109	3.1.2	System Testing	3	4	2	3	12	5	5	5	5	20	4	8	443	391	338
110	3.1.3	Compilation Testing	2	2	2	2	8	5	5	5	3	18	6	2	340	296	269
111	3.1.4	Reliability Testing	2	1	2	2	7	4	4	5	3	16	8	-1	288	260	252
112	3.1.5	Statistical Record-Keeping	2	1	2	1	6	4	2	4	2	12	12	-6	220	208	213
113	3.1.6	Software Reliability Estimation	2	1	1	1	5	3	2	2	2	9	15	-10	180	173	163
114	3.1.7	Regression Testing	2	4	2	3	11	5	5	4	5	19	5	6	426	356	291
115	3.1.8	Metric-based Testing	1	1	1	1	4	3	2	5	2	12	12	-8	211	178	168
116	3.1.9	Ad-hoc Testing	1	1	1	2	5	5	5	5	3	18	6	-1	310	246	207
117	3.1.10	Beta Testing	2	2	1	3	8	4	4	3	2	13	11	-3	277	249	224
118	3.2	SPECIAL INPUT TESTING															
119	3.2.1	Random Testing	4	4	2	2	12	3	4	5	4	16	8	4	375	368	343
120	3.2.2	Domain Testing	2	4	2	2	10	3	3	4	3	13	11	-1	320	286	254
121	3.3	FUNCTIONAL TESTING															
122	3.3.1	Functional Reqs. Testing	3	4	1	3	11	4	3	4	3	14	10	1	349	322	285
123	3.3.2	Simulation Testing	4	3	3	2	12	3	1	4	3	11	13	-1	281	305	324
124	3.3.3	Model-based Testing	4	2	4	3	13	2	2	3	2	9	15	-2	234	284	334
125	3.3.4	Assertion Checking	3	1	4	1	9	1	2	1	2	6	18	-9	146	193	236
126	3.3.5	Heuristic Testing	4	4	2	3	13	3	3	3	3	12	12	1	328	336	322
127	3.4	REALISTIC TESTING															
128	3.4.1	Field Testing	3	4	1	3	11	5	4	5	5	19	5	6	429	374	313
129	3.4.2	Scenario Testing	3	4	2	3	12	4	3	5	3	15	9	3	362	339	318
130	3.4.3	Qualification/Certification	4	5	1	3	13	3	3	5	3	14	10	3	375	367	336
131	3.4.4	Simulator-based Testing	4	3	3	3	13	3	2	3	3	11	13	0	292	315	329
132	3.4.5	Benchmarking	2	2	1	1	6	3	2	3	3	11	13	-7	236	213	187
133	3.4.6	Human Factors Experimentation	3	1	1	5	10	3	3	3	3	12	12	-2	270	270	270
134	3.4.7	Validation Scenario Testing	2	2	1	3	8	4	3	3	3	13	11	-3	284	251	222
135	3.4.8	Knowledgebase Scenario Generation	3	2	4	3	12	4	4	4	2	14	10	2	297	305	331
136	3.5	STRESS TESTING															
137	3.5.1	Stress/Accelerated Life Testing	3	3	1	1	8	5	3	4	3	15	9	-1	320	292	255
138	3.5.2	Stability Analysis Testing	3	3	1	2	9	2	3	3	3	11	13	-4	273	268	243
139	3.5.3	Robustness Testing	4	3	1	1	9	5	3	5	2	15	9	0	317	315	299
140	3.5.4	Limit/Range Testing	3	1	2	1	7	5	3	5	3	16	8	-1	285	273	272
141	3.5.5	Parameter Violation	3	1	2	2	8	4	3	5	3	15	9	-1	280	275	281
142	3.6	PERFORMANCE TESTING															

Table 6.3-1: Conventional V and V Techniques\ Power and Ease-of-Use Factor Ratings\Cost-Benefit and Effectiveness Measures

	B	C	E	F	G	H	I	K	L	M	N	O	Q	S	T	U	V
1		TECHNIQUE	P O W E R					E A S E O F U S E					Difficulty	COST-	E F F E C T I V E N E S S :		
2	Techniqu		P1	P2	P3	P4	TOTAL	E1	E2	E3	E4	TOTAL	of	BENEFI	V&V	V&V	V&V
3	Number	NAME	Hard	Broad	Formal	HCI	POWER	Learn	Setup	un/inter	Usage	EASE	use	MEASURE	Class 3	Class 2	Class 1
4																	
143	3.6.1	Sizing/Memory Testing	2	1	1	1	5	4	2	4	3	13	11	-6	239	213	196
144	3.6.2	Timing/Flow Testing	4	1	1	2	8	4	1	3	3	11	13	-5	234	253	259
145	3.6.3	Bottleneck Testing	3	1	1	2	7	4	1	3	3	11	13	-6	229	228	225
146	3.6.4	Queue size, etc.	3	1	1	1	6	4	2	3	3	12	12	-6	232	228	217
147	3.7	EXECUTION TESTING															
148	3.7.1	Activity Tracing	3	1	1	1	6	4	2	5	4	15	9	-3	276	260	246
149	3.7.2	Incremental Execute	4	2	1	1	8	4	2	4	4	14	10	-2	293	293	275
150	3.7.3	Results Monitoring	3	3	1	3	10	4	2	4	4	14	10	0	332	306	275
151	3.7.4	Thread Testing	3	3	2	1	9	4	2	3	3	12	12	-3	281	271	253
152	3.7.5	Using Generated Explanations	3	1	2	2	8	3	2	3	3	11	13	-5	228	237	246
153	3.8	COMPETENCY TESTING															
154	3.8.1	Gold Standard Testing	2	3	1	4	10	4	3	3	3	13	11	-1	318	279	243
155	3.8.2	Effectiveness Procedures	2	3	1	3	9	4	3	4	1	12	12	-3	280	255	237
156	3.8.3	Workplace Averages	2	3	1	2	8	4	3	5	3	15	9	-1	322	279	243
157	3.9	ACTIVE INTERFACE TEST															
158	3.9.1	Data Interface Testing	3	2	2	3	10	4	2	5	4	15	9	1	321	305	300
159	3.9.2	User Interface Testing	3	2	1	5	11	4	3	4	5	16	8	3	361	330	301
160	3.9.3	Information System Analysis	2	2	1	4	9	3	3	3	3	12	12	-3	279	253	231
161	3.9.4	Operational Concept Testing	3	1	1	5	10	5	4	5	3	17	7	3	337	316	309
162	3.9.5	Organizational Impact Analysis	1	2	1	4	8	4	4	4	3	15	9	-1	314	256	219
163	3.9.6	Transaction-flow Test	3	4	2	4	13	3	3	3	2	11	13	0	313	309	298
164	3.10	STRUCTURAL TESTING															
165	3.10.1	Statement Testing	2	3	1	1	7	4	1	4	3	12	12	-5	274	239	207
166	3.10.2	Branch Testing	2	3	1	1	7	4	1	4	3	12	12	-5	274	239	207
167	3.10.3	Path Testing	2	3	1	1	7	4	1	4	2	11	13	-6	254	227	204
168	3.10.4	Call-pair Testing	2	1	1	1	5	4	1	4	3	12	12	-7	226	203	191
169	3.10.5	Linear Code Sequence	3	1	1	1	6	4	1	4	2	11	13	-7	211	216	222
170	3.10.6	Test Coverage Analysis	2	3	1	3	9	4	3	3	3	13	11	-2	308	269	230
171	3.10.7	Conditional Testing	2	1	1	1	5	4	1	4	3	12	12	-7	226	203	191
172	3.10.8	Data-Flow Testing	3	4	2	3	12	3	2	3	3	11	13	-1	310	301	283
173	3.11	ERROR INTRODUCTION															
174	3.11.1	Error Seeding	2	1	1	1	5	4	2	4	3	13	11	-6	239	213	196
175	3.11.2	Fault Insertion	2	1	1	1	5	4	2	4	2	12	12	-7	219	201	193
176	3.11.3	Mutation Testing	2	1	1	1	5	4	2	3	2	11	13	-8	207	191	180

Relative Cost-Benefit = Benefits - Costs (Eq. 6.3.1-1)

The cost-benefit values resulting from this metric, discussed below, are shown in column S of Table 6.3-1. The values in column S are the values of the costs (Difficulty-of-use values) found in column Q subtracted from the values of the benefits (total of Power values) found in column I. Simply stated, column I minus column Q equals column S.

Difficulty-of-use is defined as the obverse of ease-of-use. If the ease-of-use of a specific method is ranked at the maximum of 5, then **difficulty-of-use** is the lowest possible minimum of 1; if ease-of-use is ranked very low as 2, then difficulty-of-use would be near the top of 4. Mathematically, this relationship is defined as:

Difficulty-of-use Score = 6 - (Ease-of-use Score) (Eq. 6.3.1-2)

Thus, for example, consider method 1.1.1, General Language Requirements Analysis. Its ease-of-use for learning (Column K) is rated as 2. By the above definition, the difficulty of learning for this method is a 4, using a 5-point rating scale in which 1 = Least Difficult and 5 = Most Difficult.

To find the total Difficulty-of-use score for a method, one could subtract each of the scores for the four Ease-of-use factors (columns K-N) from 6 and total these four values. Alternatively, an equivalent procedure is to total the four Ease-of-use factors (as shown in column O) and subtract this total from **24** to produce the total Difficulty-of-use for that method (as shown in column Q). To find the total Power, the estimate of **benefit**, add the four power values listed in Table 6.3-1 columns E-H; the total of the Power factors is shown in column I.

A further word is needed to explain the Difficulty-of-use component as a measure of **cost**. The less easy a technique is to use in the four ways discussed, the more difficult it is to implement. Consequently, such a technique will increase the amount of personnel time involved with the technique. Since most of the techniques require very few costs other than labor, this approach will estimate most of the actual costs. In the case of those techniques which require special equipment or expensive software, it is further assumed that the cost of these will be amortized over many uses and will, therefore, contribute very little to any one instance of use.

Returning to the details of the metric, if $\mathbf{d}_{ij}$ is the difficulty for the $_i$**th** difficulty-of-use factor for the $_j$**th** V&V technique, and if $\mathbf{p}_{ij}$ is the power of the $_i$**th** power-factor for the same $_j$**th** V&V technique, then the cost-benefit measure for the $_j$**th** V&V technique is defined as:

$$\textbf{Relative Cost-Benefit} = \sum_{i=1}^{4} p_{ij} - \sum_{i=1}^{4} d_{ij} \quad \text{(Eq. 6.3.1-3)}$$

for technique $_j$

This measure has a maximum positive value of +16, when all four power factors are at a maximum 5 value (4 x 5 = 20), and all four difficulty-of-use factors are at a minimum 1 value (4 x 1 = 4; Relative Cost-Benefit = 20 - 4 = 16). This value of positive 16 means that the technique produces maximum-possible benefits at minimum-possible costs.

The Relative Cost-Benefit measure has a maximum negative value of negative 16 when power factors are minimum and difficult-of-use factors are maximum (4 - 20 = -16). This value means that a technique produces the minimum-possible benefits at the maximum-possible cost. Although the relation between costs and benefits are sometimes expressed as a ratio of the two, this is a non-linear measure and is not as easily interpreted as the present linear one. Note that a value of 0 means that the costs equal the benefits.[13]

To assist in the interpretation of the Relative Cost-Benefits scores, shown in column S of Table 6.3-1, the following provides verbal descriptions. There is a range of plus or minus 2 points around zero where the benefits are roughly equivalent to the costs:

RANGE	INTERPRETATION
-2 to +2	Benefits and costs are roughly equivalent

For positive values of the measure, the following intervals and descriptions are suggested:

RANGE	INTERPRETATION
+3 to +8	Benefits significantly exceed costs
+9 to +12	Benefits greatly exceed costs
+13 to +16	Benefits maximally exceed costs

For negative values of this measure, the corresponding negative intervals are suggested:

RANGE	INTERPRETATION
-3 to -8	Costs significantly exceed benefits
-9 to -12	Costs greatly exceed benefits
-13 to -16	Costs maximally exceed benefits

The closer a score is to zero, the more uncertain it is. A score of +4 could be produced by having either four very high power factors (e.g., 5-5-4-5) with a corresponding high level of difficulty-of-use (e.g., 4-4-3-4), or else it could result from low power factors (e.g., 2-2-2-2) and even lower difficulty-of-use (e.g., 1-1-1-1).

This Relative cost-benefit measure identifies those techniques that have high negative values indicating that costs greatly exceed benefits. The use of such costly methods is warranted only if some aspect of their power is greatly needed, and cost effective methods are unavailable.

The conventional V&V techniques sorted by decreasing relative cost-benefit measures are shown in Table 6.3.1-1A through 6.3.1-1C. Each table, A through C, represents a different V&V Class: Requirements/Design, Static

---

[13] Another, equivalent, way of computing the cost-benefit metric is to add the total of all eight factors (e.g., column I + column O) and subtract 24 from that sum.

Table 6.3.1-1A Conventional Requirements and Design V and V Methods Ranked by Decreasing Cost-Benefit Values (Range = +12 to -12)

	COST BENEFIT MEASURE	
1.3.4	Requirements Analysis	5
1.3.1	Formal Requirements Review	4
1.3.2	Formal Design Review	3
1.2.6	Sys. Req. Eng. Method.	0
1.3.5	Protyping	0
1.2.1	Ward-Mellor Method	-1
1.2.9	Simulation-Language Anal.	-1
1.4.1	Requirements Tracing	-1
1.1.6	Refine Spec'n. Language	-2
1.1.1	Gen'l Reqs. Lang. Anal.	-3
1.3.7	Operational Concept Design Review	-3
1.2.2	Hatley-Pirbhai Method	-4
1.2.3	Harel Method	-4
1.3.3	System Engineering Analysis	-4
1.2.4	Extended Sys. Model. Lang	-5
1.2.7	FAM	-5
1.1.2	Mathematical Verification	-6
1.2.5	Sys. Eng. Methodology	-6
1.2.8	Critical Timing/Flow Anal.	-6
1.3.6	Database Design Analysis	-6
1.1.3	EHDM	-7
1.1.7	Higher Order Logic	-7
1.2.10	Petri-Net Safety Analysis	-7
1.2.11	PSL/PSA	-7
1.1.4	Z	-8
1.1.5	Vienna Definition Method	-8
1.1.8	Concurrent System Calculus	-8
1.4.2	Design Compliance Analysis	-8

Table 6.3.1-1B Conventional Static Testing V and V Methods, Sorted by Decreasing Cost-Benefit Measure Values (range = +12 to -12)

		COST BENEFIT MEASURE		
2.4.9	Knowledgebase Semantic Checking		3	
2.5.6	Desk Checking		3	
2.5.3	Formal Customer Review		2	
2.5.8	User Interface Inspection		2	
2.4.8	Knowledgebase Syntax Checking		1	
2.5.1	Informed Panel Inspection		1	
2.5.2	Structured Walk-throughs		1	
2.5.5	Peer Code Checking		1	
2.5.7	Data Interface Inspection		1	
2.2.5	Call Structure Analys		0	
2.3.7	Cross-reference List Gen'r		0	
2.5.10	Requirements Tracing		0	
2.2.1	Control Flow Analysis		-1	
2.2.4	Operational Concept Anal.		-1	
2.3.5	Look-up Table Generator		-1	
2.5.4	Clean-room Techniques		-1	
2.2.2	State Transition Diagram		-2	
2.2.3	Program Control Analysis		-2	
2.3.6	Data Dictionary Generator		-2	
2.1.4	Decision Tables		-3	
2.3.8	Aliasing Analysis		-3	
2.3.1	Data Flow Analysis		-4	
2.3.3	Dependency Analysis		-4	
2.4.4	Anomaly Testing		-4	
2.5.13	Standards Compliance		-4	
2.2.6	Process Trigger/Timing Anal		-5	
2.3.4	Qualitative Causal Analysis		-5	
2.3.11	Database Interface Analysis		-5	
2.4.6	Failure Modeling		-5	
2.4.10	Knowledge Acquisition/Refinement Aid		-5	
2.5.14	System Engineering Review		-5	
2.1.9	Metric Analyses		-6	
2.3.2	Signed Directed Graphs		-6	
2.4.1	Failure-mode Effects Caus.		-6	
2.4.3	Hazards/Safety Anal		-6	

Table 6.3.1-1B Conventional Static Testing V and V Methods, Sorted by Decreasing Cost-Benefit Measure Values (range = +12 to -12)

2.4.1	Failure-mode Effects Caus.		-6	
2.4.3	Hazards/Safety Anal		-6	
2.4.11	Knowledge Engineering Analysis		-6	
2.5.9	Standards Audit		-6	
2.1.5	Trace-assertion Meth		-7	
2.1.13	Model Evaluation		-7	
2.2.8	Concurrent Process Analysis		-7	
2.3.10	Database Analysis		-7	
2.3.12	Date-Model Evaluation		-7	
2.4.2	Criticality Analysis		-7	
2.4.5	Fault-Tree Analysis		-7	
2.4.7	Common-cause Failure Anal		-7	
2.5.12	Process Oriented Audits		-7	
2.1.3	Symbolic Execution		-8	
2.1.7	L-D Relation Method		-8	
2.1.8	Program Proving		-8	
2.2.7	Worst-case Timing Analysis		-8	
2.5.11	Software Practices Review		-8	
2.1.10	Algebraic Specification		-9	
2.1.12	Confidence Weights Sensitivity Analysis		-9	
2.1.1	Analytic Modeling		-10	
2.1.6	Functional Abstraction		-10	
2.1.11	Induction-Assertion Method		-10	
2.1.2	Cause-effect Analysis		-11	
2.3.9	Concurrency Analysis		-11	

Table 6.3.1-1C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Cost-Benefit Measure Values (Range = +12 to -12)

	COST BENEFIT MEASURE		
3.3.5	Heuristic Testing		1
3.1.2	System Testing		8
3.1.7	Regression Testing		6
3.4.1	Field Testing		6
3.2.1	Random Testing		4
3.4.2	Scenario Testing		3
3.4.3	Qualification/Certification		3
3.9.2	User Interface Testing		3
3.9.4	Operational Concept Testing		3
3.1.3	Compilation Testing		2
3.4.8	Knowledgebase Scenario Generation		2
3.3.1	Functional Reqs. Testing		1
3.9.1	Data Interface Testing		1
3.1.1	Unit/Module Testing		0
3.4.4	Simulator-based Testing		0
3.5.3	Robustness Testing		0
3.7.3	Results Monitoring		0
3.9.6	Transaction-flow Test		0
3.1.4	Reliability Testing		-1
3.1.9	Ad-hoc Testing		-1
3.2.2	Domain Testing		-1
3.3.2	Simulation Testing		-1
3.5.1	Stress/Accelerated Life Tst		-1
3.5.4	Limit/Range Testing		-1
3.5.5	Parameter Violation		-1
3.8.1	Gold Standard Testing		-1
3.8.3	Workplace Averages		-1
3.9.5	Organizational Impact Anal.		-1
3.10.8	Data-Flow Testing		-1
3.3.3	Model-based Testing		-2
3.4.6	Human Factors Experiment'n		-2
3.7.2	Incremental Execute		-2
3.10.6	Test Coverage Analysis		-2
3.1.10	Beta Testing		-3
3.4.7	Validation Scenario Testing		-3
3.7.1	Activity Tracing		-3
3.7.4	Thread Testing		-3
3.8.2	Effectiveness Procs		-3
3.9.3	Information System Analysis		-3
3.5.2	Stability Analysis Testing		-4
3.6.2	Timing/Flow Testing		-5
3.7.5	Using Generated Explanations		-5
3.10.1	Statement Testing		-5
3.10.2	Branch Testing		-5
3.1.5	Statistical Record-Keeping		-6
3.6.1	Sizing/Memory Testing		-6
3.6.3	Bottleneck Testing		-6

Table 6.3.1-1C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Cost-Benefit Measure Values (Range = +12 to -12)

	COST BENEFIT MEASURE		
3.6.4	Queue size, etc.		-6
3.10.3	Path Testing		-6
3.11.1	Error Seeding		-6
3.4.5	Benchmarking		-7
3.10.4	Call-pair Testing		-7
3.10.5	Linear Code Sequence		-7
3.10.7	Conditional Testing		-7
3.11.2	Fault Insertion		-7
3.1.8	Metric-based Testing		-8
3.11.3	Mutation Testing		-8
3.3.4	Assertion Checking		-9
3.1.6	Software Reliability Estim'n		-10

Testing, and Dynamic Testing. Findings indicate that there are more negative cost-benefit techniques than positive ones and that the range of values is skewed towards the negative end. The mean value of all the techniques is around -6. The reader is cautioned not to over-interpret the absolute values on this simplistic measure because it incorporates many assumptions. One of the most important assumptions is that each of the factors carries equivalent cost or benefit value. Nonetheless, the relative rankings shown in Tables 6.3.1-1A to C are quite consistent with intuitions, with the proven high-usage high-benefit methods having the highest values.

For the Requirements/Design methods, the Reviews are judged to be the most cost-beneficial of the general engineering methodologies. The least cost-beneficial are the
formal specification languages. The human inspection techniques are the most beneficial of the static testing techniques, followed by the intensely analytical methods.

None of the above orderings are counter-intuitive, but the ordering of the dynamic methods may be surprising to some readers. Many of the familiar "tried and true" methods are positioned in the middle and even towards the end of the grouping, particularly those that accomplish structural "white-box" testing of program paths (e.g., 3.10.1 and 3.10.2). The reason these are rated lower than their widespread use might indicate is because of their generally much lower power rating. The top three are familiar and used regularly: system, regression, and field testing. The next 15 non-negative methods are less frequently used and may be unfamiliar to many. They are at the higher position because of their higher power (that is, their "benefit"). Given these results many V&V plans, with their recommended dynamic testing techniques for the nuclear industry and elsewhere, might consider revising their V&V techniques. This table suggests that it might be possible to compose a set of testing techniques with very effective cost-benefit values.

6.3.2 The Effectiveness Metrics

This section defines a metric which considers the four power factors (Broad Power, Hard Power, Formalization, and Human-Computer Interface Testability) and the four ease-of-use factors (Ease of Mastery, Ease of Setup, Ease of Running/Interpretation, and Usage) as contributing varying amounts of effectiveness to a particular technique. Difficulty-of-use is not a consideration.

6.3.2.1 Deriving the Basic Metric

The extent to which any one of the eight factors is "beneficial" depends on the benefits one needs. If the only concern was finding techniques to test the human-computer interface factor, then 100% of the benefit derived from the eight factors would evolve from the **HCI-Tested** factor.

If percentage **weights** are attributed to each factor, then one would distribute 100 percentage points across the eight factors in accordance with one's judgment of importance. In the example above, the 4th factor, HCI-Tested, was believed to be the only factor important. Therefore, the percentage weight assignments to the factors would be the following:

Factor No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
% Weight	**0**	**0**	**0**	**100**	**0**	**0**	**0**	**0**

If for all 153 techniques, each factor was multiplied by the weight assigned to it, and then these factor-by-weight products were summed, a total for each method would be produced. The method(s) with the highest total would be the one(s) that best met the system needs - in this particular case, the ones that best tested the HCI aspects of the system. It is known that the maximum rating for any factor is 5. The above procedure would produce a value of 500 for any technique which was rated 5 in its HCI-Tested factor (the values of the remaining seven factors would be zero, since the ratings are all multiplied by a weight of 0 in this example).

To give another example, if there is no basis for judging one factor to be more important than another give equal weight to all of them, splitting the 100 weighing percentage points equally across the 8 factors would produce the following weights:

Factor No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
%Weight (w)	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**

If a technique had a rating of 1 for all eight factors, its score would be 100. However, a method with 5 for all factors would have the maximum score of 500. This latter situation is shown below:

Factor No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
%Weight (w)	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**	**12.5**
Rating (r)	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
(w) x (r)	**62.5**	**62.5**	**62.5**	**62.5**	**62.5**	**62.5**	**62.5**	**62.5**
TOTAL	**8 x 62.5 = 500**							

This procedure of assigning a weight for the expected benefit of each factor and then summing the product of these weights multiplied by the rating is summarized below. This is called the **Effectiveness Metric (EM).** The Effectiveness Metric for technique **j** is:

$$\textbf{Effectiveness Metric}_j = EM_j = \sum_{i=1}^{8} w_{ij} * r_{ij} \qquad \text{(Eq. 6.3.2.1-1)}$$

where **j** is the **j**th V&V technique, and where w_{ij} and r_{ij} are the weight and ranking of the **i**th factor, respectively, both for the **j**th technique. As stated above, a requirement of this metric is that the sum of the weights must equal 100:

$$\sum_{i=1}^{8} w_{ij} = 100 \qquad \text{(Eq. 6.3.2.1-2)}$$

This restriction reflects the fact that one cannot emphasize one aspect of a V&V method without de-emphasizing some other aspect.

The Effectiveness Metric theoretically ranges from a minimum of 100 for a V&V method **j** (if all eight factors are rated 1) to a maximum of 500 (if all eight factors are rated 5):

$$100 \leq EM_j \leq 500 \qquad \text{(Eq. 6.3.2.1-3)}$$

Having defined the metric, the next step is to carefully decide how weights are to be assigned to the eight individual factors. There are many ways to do this, but, however it is done, a disciplined approach for assigning weights is necessary to make the metric a valid tool for measuring "effectiveness".

6.3.2.2 Development of Weights for Effectiveness

Whether or not a technique is "effective" in assuring the quality of a software system depends on the characteristics of that system. For the purpose of this review, three different effectiveness metrics were developed in terms of the three V&V classes proposed in Section 2.4.3 (see Table 2.4.3-1). These classes were based jointly on two aspects of systems: their complexity and their required integrity. (Complexity in turn was previously characterized in terms of six factors; see Table 2.4.3-1). Thus, the effectiveness metric for a V&V Class 3 system should take into account that such a system is a stand-alone system of low complexity and low required integrity. Such a system is unlikely to need extremely powerful error-finding techniques; testing methods should probably be broadly capable but very easy to use. For a V&V Class 1 system, however, ease-of-use of the method is of little concern, but power to find hard defects is. Thus, each V&V Class will have differing requirements for the eight Power and Ease-of-Use factors. This section specifies how the weights for each of the V&V Classes were determined for these eight factors.

The following method was used iteratively to arrive at a stable set of weights. First, 100 percentage weight points were assigned to the overall Power and the Ease-of-use classes. Various considerations yielded the following effectiveness measurement constraints among V&V Classes:

- Class 3, Power and Ease-of-Use weights should be within 20-30 points, but neither should be more than 15 points from the mid-point.
- The number of points allotted to Power should increase significantly from Class 3 to Class 1.
- Since Class 1 and Class 2 are numerically closer in complexity and integrity, the difference in point values in the change from Class 2 to Class 1 should be less than the change from Class 3 to Class 2.
- The weights in should be assigned units of "5" to simplify this procedure.

In addition to the above rules, the eight factors were subject to the following more specific additional constraints as a second step for any set of weights arrived at with the above rules.

- Broad Power and Usage should receive the highest individual number of points for Class 3.
- The importance of Broad Power should decrease from Class 3 to Class 2, and should decrease even more from Class 2 to Class 1.
- The usage factor should parallel the decrease of Broad Power.
- HCI is probably the second most important Power factor for Class 3, and it should increase some to Class 2 and more to Class 1.
- Automatability is not important for Class 3, but is second only to Hard Power for Class 1.
- Hard Power should increase steeply from a low level in Class 3 to the most important power factor in Class 1.
- Ease of Mastery and Ease of Setup should be relatively comparable in value, with the latter given slightly more weight (since Mastery is amortized over a number of situations).
- Ease of Mastery is important for Class 3 but decreases to a minimum for Class 1.
- Ease of Setup is also important for Class 3 and also decreases to a minimum for Class 1.
- Usage should be the most important Ease-of-Use factor for Classes 3 and 2, but it should be of little concern for Class 1.

Each of these two sequential constraint steps' constraints is motivated separately, and together they constitute a formidable set of constraints on the problem. There were several iterations between the first step of deciding on the gross percentage points to allot to the collective power vs. the ease-of-use factors for the three classes of V&V and the second step of breaking down the percentage points among the eight factors. A third step was to take what seemed to be an acceptable set of weights and use them to compute the effectiveness of 3-8 selected techniques for each of the three types of V&V techniques (Requirements/Design, Static Testing, and Dynamic Testing). These techniques were well-understood and familiar, and there were strong -- and defensible -- pre-existing expectations concerning how the orderings of these methods should be for the three V&V classes when the Effectiveness Metric was computed. If a particular weighing scheme produced an ordering that was at odds with these expectations, it was rejected, and step one was begun again.

The results of the accepted first step of assigning percentage-points to the general categories of power and ease-of-use for each of the three V&V Classes, is shown below:

	% Weight for Power	% Weight for Ease-of-Use
Class 3	**40**	**60**
Class 2	**60**	**40**
Class 1	**75**	**25**

The second process was to distribute the above combined weights for Power and Ease-of-Use to the four factors in each of these categories. For example, the 75 percentage points given to Power factors for the Class 1 V&V situation were distributed to the four factors which constituted Power. The distributing of weights among the four factors in each of the two factor categories are shown in Table 6.3.2.2-1. All of the step one and step two constraints are met by this assignment. Additionally, the ordering of the selected techniques was consistent with the expectations, as determined in step three.

The overall measures of effectiveness for each conventional V&V technique was computed using formula (6.3.2.1-1). A spread-sheet product was used to calculate the results of the weighing formulas (refer to the factor ratings in columns E-H and K-N of Table 6.3-1). These ratings resulted in three overall scores for each method. Scores for each of the three V&V classes are shown in columns T-V.

6.3.3 Rank-Ordered Methods

The methods were rank-ordered according to the V&V Class Effectiveness measures, and all of these results are provided and discussed here.

The methods ordered by the Class 3 V&V Effectiveness measure were first examined, as shown in Tables 6.3.3-1A through 6.3.3-1C; the methods were grouped by major V&V category -- Requirements/Design (Table 6.3.3-1A), Static Testing (Table 6.5.5-1B), and Dynamic Testing (Table 6.3.3-1C). The weights for this Class 3 situation emphasize **broad power** strongly over the other power measures (with 24 or 60% of the possible 40 percentage points allocated to it). Thus, if one only considers the power factors, any technique which has a broad defect-detection capability will more likely be selected. If the technique is frequently used (weighted 20 percentage points), it will tend to top the list. It is for these reasons that formal reviews and inspections lead the first two testing categories, and system, field, and regression testing are the top methods of the dynamic testing methods.

Upon closer examination for Requirements/Design techniques, the formal methods are considered least effective for the Class 3 V&V situations. Semi-formal methods are found in the middle. A similar finding occurs for the Static Testing methods, with the highly analytical and formal methods being considered least effective, and more focused techniques being intermediate. The ranking of dynamic methods has the structural testing methods down in the list, along with some of the more difficult simulation and analysis techniques. Note that the ranking of methods according to the Class 3 weights very closely parallels the cost-benefit rankings found in the fourth column even though the rankings are computed by different metrics. This is quite appropriate, since both metrics tend to emphasize Ease-of-Use and Broad Power. Still, this finding tends to cross-validate both measures.

Table 6.3.3-1A Conventional Requirements and Design V and V Methods
Ranked by Decreasing V and V Class 3 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
1.3.1	Formal Requirements Review	4	416	345	271
1.3.4	Requirements Analysis	5	407	369	321
1.3.2	Formal Design Review	3	392	327	263
1.3.5	Prototyping	0	327	296	275
1.4.1	Requirements Tracing	-1	326	283	234
1.2.6	Sys. Req. Eng. Method.	0	295	320	331
1.2.1	Ward-Mellor Method	-1	280	312	327
1.1.1	Gen'l Reqs. Lang. Anal.	-3	275	263	246
1.3.3	System Engineering Analysis	-4	272	276	257
1.1.6	Refine Spec'n. Language	-2	260	300	324
1.3.7	Operational Concept Design Review	-3	260	260	257
1.2.9	Simulation-Language Anal.	-1	258	302	342
1.2.2	Hatley-Pirbhai Method	-4	243	265	277
1.2.3	Harel Method	-4	242	265	285
1.2.7	FAM	-5	230	257	273
1.3.6	Database Design Analysis	-6	229	230	226
1.2.11	PSL/PSA	-7	221	225	227
1.2.4	Extended Sys. Model. Lang	-5	220	255	283
1.1.2	Mathematical Verification	-6	218	247	260
1.2.8	Critical Timing/Flow Anal.	-6	209	250	276
1.2.5	Sys. Eng. Methodology	-6	207	245	278
1.1.7	Higher Order Logic	-7	203	220	236
1.4.2	Design Compliance Analysis	-8	203	193	181
1.1.3	EHDM	-7	197	235	265
1.1.4	Z	-8	185	225	252
1.1.5	Vienna Definition Method	-8	185	225	252
1.1.8	Concurrent System Calculus	-8	185	225	252
1.2.10	Petri-Net Safety Analysis	-7	182	221	257

Table 6.3.3-1B Conventional Static Testing V and V Methods,
Sorted by Decreasing V and V Class 3 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
2.5.6	Desk Checking	3	365	312	271
2.5.3	Formal Customer Review	2	363	324	276
2.5.1	Informed Panel Inspection	1	343	312	273
2.5.2	Structured Walk-throughs	1	337	308	269
2.5.5	Peer Code Checking	1	337	308	269
2.5.8	User Interface Inspection	2	330	310	295
2.5.7	Data Interface Inspection	1	329	304	278
2.3.7	Cross-reference List Gen'r	0	323	280	242
2.2.5	Call Structure Analysis	0	320	297	269
2.2.1	Control Flow Analysis	-1	313	274	236
2.5.10	Requirements Tracing	0	310	295	279
2.5.4	Clean-room Techniques	-1	300	339	331
2.3.5	Look-up Table Generator	-1	299	262	234
2.3.6	Data Dictionary Generator	-2	286	252	229
2.4.9	Knowledgebase Semantic Checking	3	286	348	414
2.2.4	Operational Concept Anal.	-1	285	278	274
2.2.3	Program Control Analysis	-2	281	271	257
2.4.8	Knowledgebase Syntax Checking	1	268	307	354
2.1.4	Decision Tables	-3	263	272	270
2.2.2	State Transition Diagram	-2	261	274	288
2.3.8	Aliasing Analysis	-3	249	253	257
2.4.3	Hazards/Safety Analysis	-6	248	248	225
2.5.14	System Engineering Review	-5	248	258	249
2.3.1	Data Flow Analysis	-4	245	256	268
2.3.11	Database Interface Analysis	-5	239	222	209
2.5.13	Standards Compliance	-4	237	229	237
2.5.9	Standards Audit	-6	235	215	197
2.5.12	Process Oriented Audits	-7	235	188	154
2.3.3	Dependency Analysis	-4	234	231	246
2.4.6	Failure Modeling	-5	229	264	277
2.4.4	Anomaly Testing	-4	224	246	275
2.1.9	Metric Analyses	-6	222	194	190
2.5.11	Software Practices Review	-8	220	180	150
2.2.6	Process Trigger/Timing Anal	-5	218	254	276
2.4.1	Failure-mode Effects Caus.	-6	214	268	282
2.3.10	Database Analysis	-7	211	204	200
2.1.13	Model Evaluation	-7	209	231	239
2.4.2	Criticality Analysis	-7	209	243	248
2.4.5	Fault-Tree Analysis	-7	209	243	248
2.3.4	Qualitative Causal Analysis	-5	207	234	264
2.3.2	Signed Directed Graphs	-6	206	227	244
2.4.11	Knowledge Engineering Analysis	-6	206	232	254
2.4.10	Knowledge Acquisition/Refinement	-5	199	212	243
2.3.12	Date-Model Evaluation	-7	196	217	231
2.4.7	Common-cause Failure Anal	-7	196	215	230
2.1.1	Analytic Modeling	-10	193	189	172
2.2.7	Worst-case Timing Analysis	-8	179	189	212

Table 6.3.3-1B Conventional Static Testing V and V Methods,
Sorted by Decreasing V and V Class 3 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
2.2.8	Concurrent Process Analysis	-7	178	230	270
2.1.12	Confidence Weights Sensitivity Anal	-9	170	177	190
2.1.5	Trace-assertion Meth	-7	167	232	287
2.1.2	Cause-effect Analysis	-11	165	165	159
2.1.7	L-D Relation Method	-8	162	207	253
2.1.8	Program Proving	-8	155	224	275
2.1.3	Symbolic Execution	-8	150	206	269
2.1.10	Algebraic Specification	-9	150	199	241
2.1.6	Functional Abstraction	-10	147	172	206
2.3.9	Concurrency Analysis	-11	146	160	176
2.1.11	Induction-Assertion Method	-10	137	189	236

Table 6.3.3-1C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Class 3 Values (Maximum = 500)

	COST BENEFIT MEASURE		Class 3	Class 2	Class 1
3.1.2	System Testing	8	443	391	338
3.4.1	Field Testing	6	429	374	313
3.1.7	Regression Testing	6	426	356	291
3.2.1	Random Testing	4	375	368	343
3.4.3	Qualification/Certification	3	375	367	336
3.4.2	Scenario Testing	3	362	339	318
3.9.2	User Interface Testing	3	361	330	301
3.3.1	Functional Reqs. Testing	1	349	322	285
3.1.1	Unit/Module Testing	0	347	294	247
3.1.3	Compilation Testing	2	340	296	269
3.9.4	Operational Concept Testing	3	337	316	309
3.7.3	Results Monitoring	0	332	306	275
3.3.5	Heuristic Testing	1	328	336	322
3.8.3	Workplace Averages	-1	322	279	243
3.9.1	Data Interface Testing	1	321	305	300
3.2.2	Domain Testing	-1	320	286	254
3.5.1	Stress/Accelerated Life Tst	-1	320	292	255
3.8.1	Gold Standard Testing	-1	318	279	243
3.5.3	Robustness Testing	0	317	315	299
3.9.5	Organizational Impact Anal.	-1	314	256	219
3.9.6	Transaction-flow Test	0	313	309	298
3.1.9	Ad-hoc Testing	-1	310	246	207
3.10.8	Data-Flow Testing	-1	310	301	283
3.10.6	Test Coverage Analysis	-2	308	269	230
3.4.8	Knowledgebase Scenario Generation	2	297	305	331
3.7.2	Incremental Execute	-2	293	293	275
3.4.4	Simulator-based Testing	0	292	315	329
3.1.4	Reliability Testing	-1	288	260	252
3.5.4	Limit/Range Testing	-1	285	273	272
3.4.7	Validation Scenario Testing	-3	284	251	222
3.3.2	Simulation Testing	-1	281	305	324
3.7.4	Thread Testing	-3	281	271	253
3.5.5	Parameter Violation	-1	280	275	281
3.8.2	Effectiveness Procs	-3	280	255	237
3.9.3	Information System Analysis	-3	279	253	231
3.1.10	Beta Testing	-3	277	249	224
3.7.1	Activity Tracing	-3	276	260	246
3.10.1	Statement Testing	-5	274	239	207
3.10.2	Branch Testing	-5	274	239	207
3.5.2	Stability Analysis Testing	-4	273	268	243
3.4.6	Human Factors Experiment'n	-2	270	270	270
3.10.3	Path Testing	-6	254	227	204
3.6.1	Sizing/Memory Testing	-6	239	213	196
3.11.1	Error Seeding	-6	239	213	196
3.4.5	Benchmarking	-7	236	213	187
3.3.3	Model-based Testing	-2	234	284	334

Table 6.3.3-1C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Class 3 Values (Maximum = 500)

3.6.2	Timing/Flow Testing	-5	234	253	259
3.6.4	Queue size, etc.	-6	232	228	217
	COST BENEFIT MEASURE		Class 3	Class 2	Class 1
3.6.3	Bottleneck Testing	-6	229	228	225
3.7.5	Using Generated Explanations	-5	228	237	246
3.10.4	Call-pair Testing	-7	226	203	191
3.10.7	Conditional Testing	-7	226	203	191
3.1.5	Statistical Record-Keeping	-6	220	208	213
3.11.2	Fault Insertion	-7	219	201	193
3.1.8	Metric-based Testing	-8	211	178	168
3.10.5	Linear Code Sequence	-7	211	216	222
3.11.3	Mutation Testing	-8	207	191	180
3.1.6	Software Reliability Estim'n	-10	180	173	163
3.3.4	Assertion Checking	-9	146	193	236

Some general observations can be made across the three V&V Class Effectiveness measures concerning the extent to which techniques came close to their theoretical maximum value of 500. First of all, the highest scores occurred for the Class 3 V&V weights, and the lowest scores occurred for the Class 1 weights. This can be interpreted as meaning that conventional V&V techniques are most appropriate for the V&V Class 3 systems and are less able to meet the testing needs as the required V&V stringency increases.

A second observation is that for each V&V Class, the Requirements/Design methods always had the lowest maximum scores, followed by the Static Testing methods, and the Dynamic methods being the best. The Dynamic Testing methods had the three highest overall scores, up to 88% effectiveness, for **System Testing** (443), **Field Testing** (429), and **Regression Testing** (426) in the Class 3 weighing. This finding might imply an ordering of difficulty in being able to detect problems with the three types of techniques. Or, it may reflect the fact that the V&V field has more methods and experience with Dynamic techniques than with Static ones, and least for Requirements/Design ones. Thirdly, the next three top methods were **Formal Requirements Review** (Requirements/Design, Class 3; 416), **Requirements Analysis** (Requirements/Design, Class 3; 407), and **Formal Design Review** (Requirements/ Design, Class 3; 392). If there were no other constraints, these six top methods -- the three top Dynamic methods and the next three top requirements/design methods -- would appear to constitute a very impressive suite of V&V techniques, with the high cost-benefit values and a near 80% Effectiveness capability.

The techniques were then ranked in terms of decreasing Effectiveness values as computed by the Class 2 weights. The results are shown in Tables 6.3.3-2A through 6.3.3-2C. For the Requirements/Design methods (Table 6.3.3-2A), the Class 2 weights did not cause a major re-ordering. Rather, only a few techniques moved more than a few spaces.

For the Static Testing methods, the change to Class 2 weights also did not cause significant re-clustering. The **Clean-room** Technique moved up significantly (10 places) to second place, and **Desk-checking, metric analyses**, and a few others dropped substantially. However, the majority held their general positions. The same general finding was also true for the Dynamic methods.

Finally, the techniques were ranked in terms of the Class 1 Effectiveness Metric, as shown in Tables 6.3.3-3A through 6.3.3-3C. This is the highest level of complexity/integrity requirements, and this situation calls for whatever it takes to identify problems in the code, particularly the very hard problems. In this situation, 75 of the 100 percentage points are assigned to the power factors for V&V Class 1. This is due their capability (34 points, 45%) to detect the really hard problems. The HCI is also high for this Class because safety systems of the most concern are decision-support systems with complex human-computer interactions (20 points, 27%).

Examination of the ranking of methods according to the Class 1 measure reveals entirely different orderings than with Class 3 weighing. For the Requirements/Design methods (Table 6.3.3-3A), the semi-formal methods are now very high on the list, with the extremely laborious formal methods now up in the top 40% of methods. The workhorse review and tracing methods, as well as a number of the powerful inspection methods, are high on the list for Static Testing. For Dynamic Testing, **System Testing** is still near the top, (second place) and first place is held by the **Random Testing** method. In the Class 1 situation, the cost-benefit measure is uncorrelated with the effectiveness rankings for the first half of the methods.Insert Table 6.3.2.2-1 hereIt is stressed once more that there is nothing sacrosanct about the rankings of methods according to the weighing matrix developed in Table 6.3.2.2-1. Some

Table 6.3.3-2A Conventional Requirements and Design V and V Methods
Ranked by Decreasing V and V Class 2 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
1.3.4	Requirements Analysis	5	407	369	321
1.3.1	Formal Requirements Review	4	416	345	271
1.3.2	Formal Design Review	3	392	327	263
1.2.6	Sys. Req. Eng. Method.	0	295	320	331
1.2.1	Ward-Mellor Method	-1	280	312	327
1.2.9	Simulation-Language Anal.	-1	258	302	342
1.1.6	Refine Spec'n. Language	-2	260	300	324
1.3.5	Prototyping	0	327	296	275
1.4.1	Requirements Tracing	-1	326	283	234
1.3.3	System Engineering Analysis	-4	272	276	257
1.2.2	Hatley-Pirbhai Method	-4	243	265	277
1.2.3	Harel Method	-4	242	265	285
1.1.1	Gen'l Reqs. Lang. Anal.	-3	275	263	246
1.3.7	Operational Concept Design Review	-3	260	260	257
1.2.7	FAM	-5	230	257	273
1.2.4	Extended Sys. Model. Lang	-5	220	255	283
1.2.8	Critical Timing/Flow Anal.	-6	209	250	276
1.1.2	Mathematical Verification	-6	218	247	260
1.2.5	Sys. Eng. Methodology	-6	207	245	278
1.1.3	EHDM	-7	197	235	265
1.3.6	Database Design Analysis	-6	229	230	226
1.2.11	PSL/PSA	-7	221	225	227
1.1.4	Z	-8	185	225	252
1.1.5	Vienna Definition Method	-8	185	225	252
1.1.8	Concurrent System Calculus	-8	185	225	252
1.2.10	Petri-Net Safety Analysis	-7	182	221	257
1.1.7	Higher Order Logic	-7	203	220	236
1.4.2	Design Compliance Analysis	-8	203	193	181

Table 6.3.3-2B Conventional Static Testing V and V Methods
Sorted by Decreasing V and V Class 2 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
2.4.9	Knowledgebase Semantic Checking	3	286	348	414
2.5.4	Clean-room Techniques	-1	300	339	331
2.5.3	Formal Customer Review	2	363	324	276
2.5.6	Desk Checking	3	365	312	271
2.5.1	Informed Panel Inspection	1	343	312	273
2.5.8	User Interface Inspection	2	330	310	295
2.5.2	Structured Walk-throughs	1	337	308	269
2.5.5	Peer Code Checking	1	337	308	269
2.4.8	Knowledgebase Syntax Checking	1	268	307	354
2.5.7	Data Interface Inspection	1	329	304	278
2.2.5	Call Structure Analys	0	320	297	269
2.5.10	Requirements Tracing	0	310	295	279
2.3.7	Cross-reference List Gen'r	0	323	280	242
2.2.4	Operational Concept Anal.	-1	285	278	274
2.2.1	Control Flow Analysis	-1	313	274	236
2.2.2	State Transition Diagram	-2	261	274	288
2.1.4	Decision Tables	-3	263	272	270
2.2.3	Program Control Analysis	-2	281	271	257
2.4.1	Failure-mode Effects Caus.	-6	214	268	282
2.4.6	Failure Modeling	-5	229	264	277
2.3.5	Look-up Table Generator	-1	299	262	234
2.5.14	System Engineering Review	-5	248	258	249
2.3.1	Data Flow Analysis	-4	245	256	268
2.2.6	Process Trigger/Timing Anal	-5	218	254	276
2.3.8	Aliasing Analysis	-3	249	253	257
2.3.6	Data Dictionary Generator	-2	286	252	229
2.4.3	Hazards/Safety Anal	-6	248	248	225
2.4.4	Anomaly Testing	-4	224	246	275
2.4.2	Criticality Analysis	-7	209	243	248
2.4.5	Fault-Tree Analysis	-7	209	243	248
2.3.4	Qualitative Causal Analysis	-5	207	234	264
2.4.11	Knowledge Engineering Analysis	-6	206	232	254
2.1.5	Trace-assertion Meth	-7	167	232	287
2.3.3	Dependency Analysis	-4	234	231	246
2.1.13	Model Evaluation	-7	209	231	239
2.2.8	Concurrent Process Analysis	-7	178	230	270
2.5.13	Standards Compliance	-4	237	229	237
2.3.2	Signed Directed Graphs	-6	206	227	244
2.1.8	Program Proving	-8	155	224	275
2.3.11	Database Interface Analysis	-5	239	222	209
2.3.12	Date-Model Evaluation	-7	196	217	231
2.5.9	Standards Audit	-6	235	215	197
2.4.7	Common-cause Failure Anal	-7	196	215	230
2.4.10	Knowledge Acquisition/Refinement Aid	-5	199	212	243
2.1.7	L-D Relation Method	-8	162	207	253
2.1.3	Symbolic Execution	-8	150	206	269
2.3.10	Database Analysis	-7	211	204	200

Table 6.3.3-2B Conventional Static Testing V and V Methods
Sorted by Decreasing V and V Class 2 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
2.1.10	Algebraic Specification	-9	150	199	241
2.1.9	Metric Analyses	-6	222	194	190
2.1.1	Analytic Modeling	-10	193	189	172
2.2.7	Worst-case Timing Analysis	-8	179	189	212
2.1.11	Induction-Assertion Method	-10	137	189	236
2.5.12	Process Oriented Audits	-7	235	188	154
2.5.11	Software Practices Review	-8	220	180	150
2.1.12	Confidence Weights Sensitivity Analysis	-9	170	177	190
2.1.6	Functional Abstraction	-10	147	172	206
2.1.2	Cause-effect Analysis	-11	165	165	159
2.3.9	Concurrency Analysis	-11	146	160	176

Table 6.3.3-2C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Class 2 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
3.1.2	System Testing	8	443	391	338
3.4.1	Field Testing	6	429	374	313
3.2.1	Random Testing	4	375	368	343
3.4.3	Qualification/Certification	3	375	367	336
3.1.7	Regression Testing	6	426	356	291
3.4.2	Scenario Testing	3	362	339	318
3.3.5	Heuristic Testing	1	328	336	322
3.9.2	User Interface Testing	3	361	330	301
3.3.1	Functional Reqs. Testing	1	349	322	285
3.9.4	Operational Concept Testing	3	337	316	309
3.5.3	Robustness Testing	0	317	315	299
3.4.4	Simulator-based Testing	0	292	315	329
3.9.6	Transaction-flow Test	0	313	309	298
3.7.3	Results Monitoring	0	332	306	275
3.9.1	Data Interface Testing	1	321	305	300
3.4.8	Knowledgebase Scenario Generation	2	297	305	331
3.3.2	Simulation Testing	-1	281	305	324
3.10.8	Data-Flow Testing	-1	310	301	283
3.1.3	Compilation Testing	2	340	296	269
3.1.1	Unit/Module Testing	0	347	294	247
3.7.2	Incremental Execute	-2	293	293	275
3.5.1	Stress/Accelerated Life Tst	-1	320	292	255
3.2.2	Domain Testing	-1	320	286	254
3.3.3	Model-based Testing	-2	234	284	334
3.8.3	Workplace Averages	-1	322	279	243
3.8.1	Gold Standard Testing	-1	318	279	243
3.5.5	Parameter Violation	-1	280	275	281
3.5.4	Limit/Range Testing	-1	285	273	272
3.7.4	Thread Testing	-3	281	271	253
3.4.6	Human Factors Experiment'n	-2	270	270	270
3.10.6	Test Coverage Analysis	-2	308	269	230
3.5.2	Stability Analysis Testing	-4	273	268	243
3.1.4	Reliability Testing	-1	288	260	252
3.7.1	Activity Tracing	-3	276	260	246
3.9.5	Organizational Impact Anal.	-1	314	256	219
3.8.2	Effectiveness Procs	-3	280	255	237
3.9.3	Information System Analysis	-3	279	253	231
3.6.2	Timing/Flow Testing	-5	234	253	259
3.4.7	Validation Scenario Testing	-3	284	251	222
3.1.10	Beta Testing	-3	277	249	224
3.1.9	Ad-hoc Testing	-1	310	246	207
3.10.1	Statement Testing	-5	274	239	207
3.10.2	Branch Testing	-5	274	239	207
3.7.5	Using Generated Explanations	-5	228	237	246
3.6.4	Queue size, etc.	-6	232	228	217
3.6.3	Bottleneck Testing	-6	229	228	225
3.10.3	Path Testing	-6	254	227	204
3.10.5	Linear Code Sequence	-7	211	216	222

Table 6.3.3-2C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Class 2 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
3.6.1	Sizing/Memory Testing	-6	239	213	196
3.11.1	Error Seeding	-6	239	213	196
3.4.5	Benchmarking	-7	236	213	187
3.1.5	Statistical Record-Keeping	-6	220	208	213
3.10.4	Call-pair Testing	-7	226	203	191
3.10.7	Conditional Testing	-7	226	203	191
3.11.2	Fault Insertion	-7	219	201	193
3.3.4	Assertion Checking	-9	146	193	236
3.11.3	Mutation Testing	-8	207	191	180
3.1.8	Metric-based Testing	-8	211	178	168
3.1.6	Software Reliability Estim'n	-10	180	173	163

Table 6.3.3-3A Conventional Requirements and Design V and V Methods
Ranked by Decreasing V and V Class 1 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
1.2.9	Simulation-Language Anal.	-1	258	302	342
1.2.6	Sys. Req. Eng. Method.	0	295	320	331
1.2.1	Ward-Mellor Method	-1	280	312	327
1.1.6	Refine Spec'n. Language	-2	260	300	324
1.3.4	Requirements Analysis	5	407	369	321
1.2.3	Harel Method	-4	242	265	285
1.2.4	Extended Sys. Model. Lang	-5	220	255	283
1.2.5	Sys. Eng. Methodology	-6	207	245	278
1.2.2	Hatley-Pirbhai Method	-4	243	265	277
1.2.8	Critical Timing/Flow Anal.	-6	209	250	276
1.3.5	Prototyping	0	327	296	275
1.2.7	FAM	-5	230	257	273
1.3.1	Formal Requirements Review	4	416	345	271
1.1.3	EHDM	-7	197	235	265
1.3.2	Formal Design Review	3	392	327	263
1.1.2	Mathematical Verification	-6	218	247	260
1.3.3	System Engineering Analysis	-4	272	276	257
1.3.7	Operational Concept Design Review	-3	260	260	257
1.2.10	Petri-Net Safety Analysis	-7	182	221	257
1.1.4	Z	-8	185	225	252
1.1.5	Vienna Definition Method	-8	185	225	252
1.1.8	Concurrent System Calculus	-8	185	225	252
1.1.1	Gen'l Reqs. Lang. Anal.	-3	275	263	246
1.1.7	Higher Order Logic	-7	203	220	236
1.4.1	Requirements Tracing	-1	326	283	234
1.2.11	PSL/PSA	-7	221	225	227
1.3.6	Database Design Analysis	-6	229	230	226
1.4.2	Design Compliance Analysis	-8	203	193	181

Table 6.3.3-3B Conventional Static Testing V and V Methods
Sorted by Decreasing V and V Class 1 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
2.4.9	Knowledgebase Semantic Checking	3	286	348	414
2.4.8	Knowledgebase Syntax Checking	1	268	307	354
2.5.4	Clean-room Techniques	-1	300	339	331
2.5.8	User Interface Inspection	2	330	310	295
2.2.2	State Transition Diagram	-2	261	274	288
2.1.5	Trace-assertion Meth	-7	167	232	287
2.4.1	Failure-mode Effects Caus.	-6	214	268	282
2.5.10	Requirements Tracing	0	310	295	279
2.5.7	Data Interface Inspection	1	329	304	278
2.4.6	Failure Modeling	-5	229	264	277
2.5.3	Formal Customer Review	2	363	324	276
2.2.6	Process Trigger/Timing Anal	-5	218	254	276
2.4.4	Anomaly Testing	-4	224	246	275
2.1.8	Program Proving	-8	155	224	275
2.2.4	Operational Concept Anal.	-1	285	278	274
2.5.1	Informed Panel Inspection	1	343	312	273
2.5.6	Desk Checking	3	365	312	271
2.1.4	Decision Tables	-3	263	272	270
2.2.8	Concurrent Process Analysis	-7	178	230	270
2.5.2	Structured Walk-throughs	1	337	308	269
2.5.5	Peer Code Checking	1	337	308	269
2.2.5	Call Structure Analys	0	320	297	269
2.1.3	Symbolic Execution	-8	150	206	269
2.3.1	Data Flow Analysis	-4	245	256	268
2.3.4	Qualitative Causal Analysis	-5	207	234	264
2.2.3	Program Control Analysis	-2	281	271	257
2.3.8	Aliasing Analysis	-3	249	253	257
2.4.11	Knowledge Engineering Analysis	-6	206	232	254
2.1.7	L-D Relation Method	-8	162	207	253
2.5.14	System Engineering Review	-5	248	258	249
2.4.2	Criticality Analysis	-7	209	243	248
2.4.5	Fault-Tree Analysis	-7	209	243	248
2.3.3	Dependency Analysis	-4	234	231	246
2.3.2	Signed Directed Graphs	-6	206	227	244
2.4.10	Knowledge Acquisition/Refinement Aid	-5	199	212	243
2.3.7	Cross-reference List Gen'r	0	323	280	242
2.1.10	Algebraic Specification	-9	150	199	241
2.1.13	Model Evaluation	-7	209	231	239
2.5.13	Standards Compliance	-4	237	229	237
2.2.1	Control Flow Analysis	-1	313	274	236

Table 6.3.3-3B Conventional Static Testing V and V Methods
Sorted by Decreasing V and V Class 1 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
2.1.11	Induction-Assertion Method	-10	137	189	236
2.3.5	Look-up Table Generator	-1	299	262	234
2.3.12	Date-Model Evaluation	-7	196	217	231
2.4.7	Common-cause Failure Anal	-7	196	215	230
2.3.6	Data Dictionary Generator	-2	286	252	229
2.4.3	Hazards/Safety Anal	-6	248	248	225
2.2.7	Worst-case Timing Analysis	-8	179	189	212
2.3.11	Database Interface Analysis	-5	239	222	209
2.1.6	Functional Abstraction	-10	147	172	206
2.3.10	Database Analysis	-7	211	204	200
2.5.9	Standards Audit	-6	235	215	197
2.1.9	Metric Analyses	-6	222	194	190
2.1.12	Confidence Weights Sensitivity Analysis	-9	170	177	190
2.3.9	Concurrency Analysis	-11	146	160	176
2.1.1	Analytic Modeling	-10	193	189	172
2.1.2	Cause-effect Analysis	-11	165	165	159
2.5.12	Process Oriented Audits	-7	235	188	154
2.5.11	Software Practices Review	-8	220	180	150

Table 6.3.3-3C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Class 1 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 3	Class 2	Class 1
3.2.1	Random Testing	4	375	368	343
3.1.2	System Testing	8	443	391	338
3.4.3	Qualification/Certification	3	375	367	336
3.3.3	Model-based Testing	-2	234	284	334
3.4.8	Knowledgebase Scenario Generation	2	297	305	331
3.4.4	Simulator-based Testing	0	292	315	329
3.3.2	Simulation Testing	-1	281	305	324
3.3.5	Heuristic Testing	1	328	336	322
3.4.2	Scenario Testing	3	362	339	318
3.4.1	Field Testing	6	429	374	313
3.9.4	Operational Concept Testing	3	337	316	309
3.9.2	User Interface Testing	3	361	330	301
3.9.1	Data Interface Testing	1	321	305	300
3.5.3	Robustness Testing	0	317	315	299
3.9.6	Transaction-flow Test	0	313	309	298
3.1.7	Regression Testing	6	426	356	291
3.3.1	Functional Reqs. Testing	1	349	322	285
3.10.8	Data-Flow Testing	-1	310	301	283
3.5.5	Parameter Violation	-1	280	275	281
3.7.3	Results Monitoring	0	332	306	275
3.7.2	Incremental Execute	-2	293	293	275
3.5.4	Limit/Range Testing	-1	285	273	272
3.4.6	Human Factors Experiment'n	-2	270	270	270
3.1.3	Compilation Testing	2	340	296	269
3.6.2	Timing/Flow Testing	-5	234	253	259
3.5.1	Stress/Accelerated Life Tst	-1	320	292	255
3.2.2	Domain Testing	-1	320	286	254
3.7.4	Thread Testing	-3	281	271	253
3.1.4	Reliability Testing	-1	288	260	252
3.1.1	Unit/Module Testing	0	347	294	247
3.7.1	Activity Tracing	-3	276	260	246
3.7.5	Using Generated Explanations	-5	228	237	246
3.8.3	Workplace Averages	-1	322	279	243
3.8.1	Gold Standard Testing	-1	318	279	243
3.5.2	Stability Analysis Testing	-4	273	268	243
3.8.2	Effectiveness Procs	-3	280	255	237
3.3.4	Assertion Checking	-9	146	193	236
3.9.3	Information System Analysis	-3	279	253	231
3.10.6	Test Coverage Analysis	-2	308	269	230
3.6.3	Bottleneck Testing	-6	229	228	225

Table 6.3.3-3C Conventional Dynamic Testing V and V Methods
Sorted by Decreasing Class 1 Values (Maximum = 500)

		COST BENEFIT MEASURE	Class 1	Class 2	Class 1
3.1.10	Beta Testing	-3	277	249	224
3.4.7	Validation Scenario Testing	-3	284	251	222
3.10.5	Linear Code Sequence	-7	211	216	222
3.9.5	Organizational Impact Anal.	-1	314	256	219
3.6.4	Queue size, etc.	-6	232	228	217
3.1.5	Statistical Record-Keeping	-6	220	208	213
3.1.9	Ad-hoc Testing	-1	310	246	207
3.10.1	Statement Testing	-5	274	239	207
3.10.2	Branch Testing	-5	274	239	207
3.10.3	Path Testing	-6	254	227	204
3.6.1	Sizing/Memory Testing	-6	239	213	196
3.11.1	Error Seeding	-6	239	213	196
3.11.2	Fault Insertion	-7	219	201	193
3.10.4	Call-pair Testing	-7	226	203	191
3.10.7	Conditional Testing	-7	226	203	191
3.4.5	Benchmarking	-7	236	213	187
3.11.3	Mutation Testing	-8	207	191	180
3.1.8	Metric-based Testing	-8	211	178	168
3.1.6	Software Reliability Estim'n	-10	180	173	163

reasonable assumptions and constraints were made about the relative contributions of the eight factors in these three situations and the weightings were adjusted accordingly. What is provided in this section is a methodology which attempts to reduce the problems of subjective bias, habit, or other unreflective technique-selection processes, by forcing consideration of each method on a set of eight relatively independent factors and then combining these judgments according to a reasonable linear additive weighing procedure.

6.4 Which Techniques to Use, and When

The rankings of the V&V methods were accomplished without reference to any particular system or application. In reality, every system will have specific requirements that will make certain techniques necessary regardless of their ranking for any complexity/integrity combination..

Any project will only use a very small fraction of the total 153 V&V techniques. All of these methods are discussed because each has its own special qualifications. However, a single project has a finite V&V budget and great care must be taken to select appropriate methods. The best way to determine which V&V technique to use is to prioritize the problems that might occur. For example, decide which system problems will cause failure, list those concerns, and choose the best methods to test for those problems. A reasonable standard procedure is first to decide the AV&V Class of the system, Table 6.3.3-2 and find the appropriate table ranking of the techniques according to the Effectiveness weighing and metric for that class.

The order of using V&V techniques is also an important consideration. For example, if it is necessary to use functional testing, random testing, and statement testing, it would be a mistake to generate test-cases to test program statements first. Statements will be automatically tested by the other two methods. If one keeps track of the statements covered by these two methods, then one needs only to generate test-cases for the remaining untested statements after the others have been exercised.

Finally, a plan of action is needed when errors are detected. This is a very complicated issue. The worst mistake is to make a quick fix and continue with testing. This will invariably introduce new errors or side effects. Additionally, this event renders the documentation obsolete more rapidly than normal. The better approach is to continue with reasonable testing until a collection of problems is detected. In fact, when an error is encountered, subsequent tests can be designed to effectively explore an analysis of the causes of the error revealed. There may be very serious design flaws due to certain overlooked processing factors. With a collection of problems, the best action is to first analyze them for likely related problems (and test for these) and problems in one's processes of software development; then one can plan a series of modifications to the system design followed by modifications to the code and documentation.

The cost-benefit and effectiveness ratings for each of the 153 V&V techniques in Section 6.3 provide detailed information with an overall ranking of these methods. Such a ranking is useful in assessing the most likely candidates for V&V and analyzing the trends among the most beneficial V&V techniques. Therefore, each technique was ranked within its category of either requirements/design, static, or dynamic testing.

Within each of these categories, each technique was numerically ranked in accordance with its cost benefit measure as delineated in Tables 6.3.1-1A-C. Then, each technique was numerically ranked in accordance with its effectiveness rating for each of the three V&V classes as it is delineated in Tables 6.3.3-1A-C, 6.3.3-2A-C, and 6.3.3-3A-C. These rankings appear as columns 2-5 in Table 6.4-1. After this rank assignment for each technique, the four rankings (cost-benefit and effectiveness for V&V Classes 1, 2, and 3) were summed and appear in column 6. Finally, these summed values were ranked, and the results are given in column 7. For example, technique 1.3.2, Formal Design Review, would be ranked number 1 based on its cost benefit measure in Table 6.3.1-1A, ranked number 1 based on its effectiveness for V&V Class 3 in Table 6.3.3-1A, ranked number 1 for V&V Class 2 effectiveness in Table 6.3.3-2A, and ranked number 15 for V&V Class 1 for effectiveness in Table 6.3.3-3A. The sum of 18 was assigned to this technique. In the same manner, the top ten V&V techniques for requirements/design, static, and dynamic testing were calculated and are delineated in Table 6.4-1 in numerical order from 1 to 10.

The above method provides a way of determining the top-rated techniques based on all four of the measures computed for them, the cost-benefit measure, and the three effectiveness measures. Several generalizations concerning the final top-ten ranked methods in Table 6.4-1 can be made. In requirements and design testing, traditional formal review methods along with some new automated techniques made this top list. For static testing V&V methods, all the highly ranked techniques involve personal inspections and reviews; none are automated. Finally, in the area of dynamic testing, all the techniques on this list were system level, function oriented "black box" methods; no structural "white box" methods made the list. Some of the dynamic testing methods also involved automation.

Many of the high ranked methods listed in Table 6.4-1 are widely used and accepted as conventional software V&V. These methods serve as a starting point in later activities for this project in selecting appropriate V&V methods for expert systems. For those expert system component which are directly
amenable to conventional software V&V methods (to be discussed in Section 7 of this report), some of the highly ranked techniques in Table 6.4-1 may be appropriate. In the case where new V&V techniques need to be developed, the insights gained from examining highly ranked conventional V&V methods will provide guidance in selecting new expert system V&V methods.

Table 6.4-1 Overall highest ranked conventional V&V techniques for all V&V classes

	Conventional V&V Techniques	Cost Benefit Metric[1] Ranking	Effectiveness Metric Ranking			Sum of Ranks	Ranking Over All 4 Metrics[2]
			Class 1	Class 2	Class 3		
REQS. & DESIGN	Requirement Analysis	1	5	1	2	9	1
	Systems Requirements Engineering	4.5	2	4	6	16.5	2
	Formal Requirements Review	2	13	2	1	18	3
	Ward-Mellor Method	7	3	5	7	21	4
	Formal Design Review	3	15	3	3	24	5
	Simulation Language Analysis	7	1	6	12	25	6
	Prototyping	4.5	11	8	4	27.5	7
	Refine Specification Language	9	4	7	10.5	30.5	8
	Harel Method	13	6	11.5	14	44.5	9
	Requirements Tracing	7	25	9	5	45	10
STATIC TESTING	Knowledgebase Semantic Checking	14.5	1	1	14.5	18	1
	User Interface Inspection	6	4	6	6	19.5	2
	Formal Customer Review	2	11.5	3	2	20	3
	Desk Checking	1	17	4.5	1	24	4
	Informed Panel Inspection	3	16	4.5	3	30.5	5
	Clean-room Techniques	12	3	2	12	31.5	6
	Data Interface Inspection	7	9	10	7	33	7
	Knowledgebase Syntax Checking	18	2	9	18	36	8
	Structured Walk-Throughs	4.5	21.5	7.5	4.5	40.5	9.5
	Peer Code Checking	4.5	21.5	7.5	4.5	40.5	9.5

Table 6.4-1 (Continued)

Conventional V&V Techniques		Cost Benefit Metric[1] Ranking	Effectiveness Metric Ranking			Sum of Ranks	Ranking Over All 4 Metrics[2]
			Class 1	Class 2	Class 3		
DYNAMIC TESTING	System Testing	1	2	1	1	5	1
	Random Testing	4	1	3	4.5	12.5	2
	Field Testing	2.5	10	2	2	16.5	3
	Qualification Certification	6.5	3	4	4.5	18	4
	Regression Testing	2.5	16	5	3	26.5	5
	Scenario Testing	6.5	9	6	6	27.5	6
	User Interface Testing	6.5	12	8	7	33.5	7
	Operational Concept Testing	6.5	11	10	11	38.5	8
	Heuristic Testing	12	8	7	13	40	9
	Functional Requirements Testing	12	17	9	8	46	10

[1] Based on the cost benefit measure (Column 5) of Table 2.3.2-2, all the methods (for a phase) have been ranked. Note that only a subset are shown here.

[2] Based on the sums of rankings for the four metrics, ordered from lowest to highest.

7 ASSESSMENT OF THE APPLICABILITY OF CONVENTIONAL V&V TECHNIQUES TO EXPERT SYSTEMS

The previous two sections discussed the results of classifying and characterizing conventional V&V techniques. This section examines whether these V&V techniques are applicable to expert systems.

Section 7.1 describes the heterogeneous components of expert systems and Section 7.2 identifies key V&V features of these and suggests generally appropriate V&V approaches. Section 7.3 provides a detailed examination of the applicability of the 153 conventional techniques to the components and to the system as a whole. In Section 7.4, the limitations of using conventional V&V techniques for expert systems are summarized, along with suggestions for extensions to these methods. This section ends with a suggested strategy for assuring the quality of expert systems.

7.1 Components of Expert Systems

Expert systems, regardless of their application, generally have several essential functions just as spread-sheet packages or data-base management systems have different functional components. From the point of view of developing or managing expert systems, authors have focused on two components: a Knowledge Base and an Inference Engine. However, the best description of components from a V&V point of view is one which lists all the aspects that have to be tested (or certified) in order to achieve the QA and V&V objectives. The four components, and their subcomponents, from this V&V perspective are shown in the left-most column of Table 7.1-1. In addition to the components that were actually developed are all the elements that were used in the expert system development.

The first component is the Inference Engine. It is that part of the expert system which determines what gets done next. It controls the interpretations and decisions, and manages the results. The inference engine works on declarative knowledge which is most frequently in the form of IF-THEN rules. To process such rules, the engine possesses two key subcomponents: the Pattern Matcher, which determines which rules could next be activated, and the Conflict-Set Handler, which priorities these rules.

There are six additional subcomponents of the inference engine. The Proof Procedure subcomponent determines the nature of the decision-making or reasoning. An example is the working backward (top-down) from goals to facts to realize those goals, or working forward (bottom-up) from facts (data) to inferences or goals; the former is a "backwards-chaining" procedure, while the latter is a "forward-chaining" one. Another important aspect of the proof procedure is whether the possible rules or data to be examined are considered exhaustively at each level of chaining ("breadth-first" search) or whether a single rule is followed to its consequence, then a rule linked to that consequence is explored, and so on until a goal or dead-end is reached ("depth-first" search). The Proof Manager records the activity of the proof procedure component in a goal-tree or similar structure and manages (and annotates) that tree following decision-outcomes. The subcomponent called the Knowledge Processor is a specialized unit which understands the formats and structures of the various knowledge representations and makes available those aspects which the inference engine needs for processing. The Fuzzy Value/Uncertainty Handler is uncommon but provides for specialized processing of fuzzy-logic and uncertainty calculations within inferences. The Inheritance Processor deals with knowledge represented as a frame or object (see below). For a particular element, it determines what information about that element is inherited from its parent elements. Finally, the Agenda Handler is the overall scheduling module for the inference engine, determining what specifically is done next, controlling input/output, and managing access to other tools, databases, and environments.

The second major component is the Knowledge Base. It is the heart of expert systems. It contains all the specific domain knowledge for a particular application. Six possible representations of knowledge are identified: rules, frames, objects, facts, external databases, and in-line Demons. The most common representation is that of *IF-THEN Rules*, rules which specify a set of conditions to be satisfied before a set of actions can be taken. A *Frame* is composed of a set of related attributes that describe some knowledge concept. Frames are organized in a hierarchy in which the parent node is a more general concept, while the child nodes are particular and specific types of that parent concept (e.g., there might be a frame describing valves in general, which might have two specialization frames - "relief valves" and "input valves"). If the attributes describing a parent frame are true in all respects for the child, then these need not be actually be specified as attributes in the child-frame; they can be "inherited" by default. *Objects* possess frame characteristics as well as a set of "methods" (special procedures for displaying or activating the objects). Expert systems which involve objects as a major knowledge component are a hybrid between pure expert systems in which the processing is done primarily by the inference engine and object-oriented systems, in which the processing is accomplished by "messages" or sets of instructions which are passed among objects. *Facts* are relatively simple attribute-values which ascribe properties to things, such as the status of a nuclear plant variable, (e.g., "RPV_PRESSURE(BELOW, RCIC_ISOLATION_PRESSURE)"). When there are a very large number of facts which the expert system needs to use, they are often stored, not in some internal form as in the above example, but in an *External Database*. These databases are typically either flat-file or relational in structure and are accessed via the Knowledge Processor through the Database Interface. Finally, knowledge can be represented by *Demons*, which are calls to some conventional software procedure which returns a value. The demons are typically expressed as the values of attributes in the other types of knowledge representations. For example, in a frame describing various features of an airplane in flight, the attribute describing the amount of gas still in the tanks might be expressed as:

Fuel: Call FUEL_REMAINING

The value of the *Fuel* attribute is the demon "FUEL_REMAINING", which is a software procedure which interrogates sensors in the fuel tanks and returns a value of some number of gallons.

The third component of expert systems consists of Interfaces. Other than the primary User-Interface, these are seldom considered in discussions of expert system components since most expert systems, historically, have been stand-alone systems, with just a user-interface. Nevertheless, the typical modern expert system more and more has a variety of data and other interfaces and is embedded in larger conventional software systems. Just as these must be considered carefully in the V&V of conventional systems, so must they be for expert systems. The various aspects of interfaces are: user interface, database interface, data input/output channels, communication ports, hyper-media, and interfaces to other applications and the operating system. Most of these are self-explanatory, and two comments will suffice. The difference between a *data I/O channel* and a *communication port* is that the former is usually a special input/output capability designed specifically for certain types of data streams or sensor inputs, while the latter is a standardized channel for communicating with other platforms or devices. *Hyper-media* involves information retrieval from potentially several data storage and representation methods, including text, audio, and video.

Table 7.1-1 Components and typical testing-related features of knowledge-based systems, with testing recommendations.

Components	Features[1]		
	Typically Written in or Involving Procedural Code	**High Reusability Across Different Applications**	**Potential Defects Known, Fairly Delineated, & Amenable to Formal Test Methods**
Inference Engineer	**Y[2]**	**Y**	**S**
Pattern Matcher	Y	Y	S
Conflict-Set Handler	Y	Y	S
Proof Procedure	Y	Y	S
Proof Manager	S	S	N
Fuzzy Value/Uncertainty Handler	S	Y	S
Knowledge Processor	S	Y	S
Inheritance Processor	Y	Y	S
Agenda Handler	Y	Y	S
Knowledge Base	**N**	**N**	**S**
Rules	N	N	Y
Frames	N	N	Y
Objects	M	N	S
Facts	N	N	S
External Database	M	N	S
In-line Demons, Procedures, Functions	Y	N	N
Interfaces	**Y**	**S**	**N**
User Interface	Y	S	N
Database Interface	Y	S	N
Data I/O Channels	Y	S	S
Communication Ports	Y	N	N
Hyper-media	Y	N	N
Interfaces to Other Applications and the Operating System	Y	S	N

[1] Explanation of table entries: Y=Yes, N=No, S=Sometimes, M=Mixed procedural and non-procedural code
[2] The overall "average" of the cell

Table 7.1-1 (Continued)

Components	Features		
	Typically Written in or Involving Procedural Code	**High Reusability Across Different Applications**	**Potential Defects Known, Fairly Delineated, & Amenable to Formal Test Methods**
Tools and Utilities	**Y**	**Y**	**N**
Compilers	Y	Y	S
Linkers/Loaders	Y	Y	N
Debuggers/Code Checkers	Y	Y	N
Knowledge Engineering Tools	Y	Y	N
Graphics User Interface (GUI)	Y	Y	N
Capability	Y	Y	N
Expert System Shell	Y	N	N
Custom Code			

The fourth component of expert systems, Tools and Utilities, is also seldom considered. This component comprises all of the general applications, tools, and programs that were used to develop the implemented expert systems. Despite the fact that most of these are not actually delivered with the developed expert system, they can introduce errors into the system and must be considered in the V&V plan. Compilers, linker/loaders, and **debugger/code checkers** are standard programming development tools. **Knowledge Engineering Tools** refers to special programs or products used to acquire or represent knowledge. They differ in capability from vendor to vendor. The **Graphics User Interface (GUI)** utility provides the functionality for defining the communication capability in the user interface in terms of graphical entities such as menus, buttons, icon symbols, etc., typically operated with a mouse. This feature is usually provided by commercial packages and is more and more a feature of expert systems development. The **Expert System Shell** is an integrating environment for the above components and is found in many products. Finally, **Custom Code** refers to special systems-level or application code that was written to enable communication among the various expert systems components or to accomplish special functions.

7.2 Key V&V Characteristics of Expert Systems Components

The three "features" columns of Table 7.1-1 are discussed below. These features were chosen in particular because of the general implications they have for how V&V should best be accomplished. They provide the basis for suggesting general V&V strategies for each component and are used to guide the detailed examination of the applicability of the 153 conventional techniques presented in the next section. The three features comprise questions asked of both the main components of expert systems and each of their subcomponents. The outcome is indicated in the table by **Y** for Yes, **N** for No, **S** for sometimes, and (for the first feature asking about code) **M** for Mixed procedural and non-procedural code. The overall rating of a component, based on its subcomponent ratings, is given in the upper left corner of each feature box.

The first feature asks whether conventional, procedural, programming languages such as **C, FORTRAN,** or **Ada** are typically used to program the components. The second feature asks whether the components can easily be reused over a wide variety of different applications; if the component has to be modified for each new application, it has low reusability. The third feature focuses on all the problems or defects that could plague a component; the question is whether these defects are fairly restricted, are pretty well known now, and are amenable to some kind of formal detection procedure. A "yes" answer to this question means that enough progress has been made for practitioners to agree that formal approaches are quite promising and should be further pursued.

"Yes" answers to the three feature questions shown in Table 7.1-1 define three very broad but appropriate V&V strategies. For the first feature, a **Yes** answer for a component suggests that this component should be tested with conventional V&V techniques. Thus, The Inference Engine, the Interfaces, and the Tools and Utilities should, generally, all be tested using conventional approaches.

For the second feature, a **Yes** answer for a component suggests that this component should be bench-marked and subject to certification tests, because it has such wide applicability. The Inference Engine and the Tools and Utilities components qualify for this special treatment. Summarizing the results so far, the Inference Engine, the Interfaces, and the Tools and Utilities should all be developed and tested with conventional V&V techniques, according to the results of the first feature question. If these components are to be made available as a commercial product, according to the second feature the Inference Engine and the Tools and Utilities should then be bench-marked and certified.

Concerning the third feature, a **Yes** answer for a component implies that some kind of automated defect-detecting tool should be developed and used to examine the component for the known defects. Only the sub-types of **Rules** and **Frames** of the Knowledge Base component now qualify for this recommendation..

7.3 Applicability of Conventional Methods

Each of the 153 individual conventional V&V techniques was rated as to its applicability to the four components of expert systems as well as to the system as a whole. A rating of 1 indicates that the technique can be applied to the component directly without modification, while the lowest rating of 4 indicates that the technique is not applicable to that component. Techniques with the highest applicability rating for the expert system components are marked by an asterisk in Table 7.3-1. The reader is cautioned that the present judgments of applicability are necessarily relatively crude and certainly subjective. Moreover, the judgments do not take into account the capability of the conventional techniques to detect defects which are unique to expert systems, whatever these defects might be.

7.3.1 Methods Applicable to the Interface Component

Expert systems' interfaces are typically written in conventional procedural code, but their reusability is not high. Therefore, the best V&V strategy is to select conventional techniques for each sub-component which provide the appropriate required safety, reliability, etc; Table 6.3-1, which rates each technique on eight power and ease-of-use factors, can be used to aid this selection. Alternatively, if specific defects are of concern, one can use Tables 6.1.1-1 (which list the defects) and 6.1.2-1 (which lists the defects judged detectable by each technique). Since the interface component mostly concerns data rather than control aspects of programs, a number of conventional techniques do not apply as strongly to this component.

The techniques with the highest general applicability ratings to the interface component in Table 7.3-1 are the following: Database Analyzer (2.3.10), Database Interface Analyzer (2.3.11), Data-Model Evaluation (2.3.12), Data Interface Inspection (2.5.7), User Interface Inspection (2.5.8), Standards Audit (2.5.9), Requirements Tracing (2.5.10), Regression Testing (3.1.7), Uniform Whole Program Testing (3.2.1.1), Specific Functional Requirements Testing (3.3.1), Stress/Accelerated Life Testing (3.5.1), Data Interface Testing (3.9.1), and User Interface Testing (3.9.2).[14]

7.3.2 Methods Applicable to Tools and Utilities

With the exception of custom code, all of the Tools and Utilities are typically written in procedural code and have high reusability across different applications. Rather than test these subcomponents for each application, it is more appropriate to certify them for all applications. This includes both the commercial developer and independent agents. To accomplish this testing, the qualification/certification testing method (3.4.3) can be used.

---

[14] The suggestions in this section apply equally to the interfaces and the tools and utilities of conventional systems.

Table 7.3-1 Rating of the applicability of conventional techniques to expert systems and their components[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
1.1.1 General Requirements Language Analysis/Processing	1	3	1	1	1
1.1.2 Mathematical Verification of Requirements	1	3	1	1	1
1.1.3 EHDM	1	2-3	1	1	1
1.1.4 Z	1	2-3	1	1	1
1.1.5 Vienna Definition Method	1	3	1	1	1
1.1.6 Refine Specification Language	1	3	1	1	1
1.1.7 Higher Order Logic (HOL)	1	3	1	1	1
1.1.8 Concurrent System Calculus	1	3	1	1	1
1.2.1 Ward-Mellor Method	1	3	1	1	1
1.2.2 Hatley-Pirbhai Method	1	3	1	1	1
1.2.3 Harel Method	1	3	1	1	1
1.2.4 Extended Systems Modeling Language	1	3	1	1	1
1.2.5 Systems Engineering Methodology	1	3	1	1	1
1.2.6 System Requirements Engineering Methodology	1	2-3	1	1	1
1.2.7 FAM	1	3	1	1	1
1.2.8 Critical Timing/Flow Analysis	1	3	1	1	1
1.2.9 Simulation-Language Analysis	1	3	1	1	1

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

Table 7.3-1 (Continued)[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
1.2.10 Petri-Net Safety Analysis	1	3	1	1	1
1.2.11 PSL/PSA	1	3	1	1	1
1.3.1 Formalized Requirements Review	1	1	1	1	1*
1.3.2 Formal Design Review	1	1	1	1	1*
1.3.3 System Engineering Analysis	1	3	1	1	1*
1.3.4 Requirements Analysis	1	1	1	1	1*
1.3.5 Prototyping	1	1	1	1	1
1.3.6 Database Design Analysis	4	2	3-4	3-4	3
1.3.7 Operational Concept Design Review	3	1-2	2	2	2
1.4.1 Requirements Tracing Analysis	1*	1*	1*	1*	1*
1.4.2 Design Compliance Analysis	1	1	1	1	1
2.1.1 Analytic Modeling	1	2	2	1	1
2.1.2 Cause-Effect Analysis	1	3	2	1	1
2.1.3 Symbolic Execution	1	4	2	1	1
2.1.4 Decision Tables	1	3	2	1	1
2.1.5 Trace-Assertion Method	1	4	2	1	1
2.1.6 Functional Abstraction	1	4	2	1	1
2.1.7 L-D Relation Methods	1	4	2	1	1
2.1.8 Program Proving	1	4	2	1	1
2.1.9 Metric Analysis	1	3	2	1	1

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

Table 7.3-1 (Continued)[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
2.1.10 Algebraic Specification	1	4	2	1	1
2.1.11 Introduction-Assertion Method	1	4	2	1	1
2.1.12 Confidence Weights Sensitivity Analysis	3	2	3	3	2
2.1.13 Model Evaluation	4	1-2	3	4	2
2.2.1 Control Flow Analysis	1	3	2	1	1
2.2.2 State Transition Diagram Analysis	1	2*	1	1	1
2.2.3 Program Control Analysis	1	3	2	1	1
2.2.4 Operational Concept Analysis	1	3	3	1	1*
2.2.5 Calling Structure Analysis	1	4	3	1	1
2.2.6 Process Trigger/Timing Analysis	1	3	1	1	1
2.2.7 Worst-Case Timing Analysis	1	4	1	1	1
2.2.8 Concurrent Process Analysis	1	4	1	1	1
2.3.1 Data Flow Analysis	1*	2	1	1	1
2.3.2 Influence Diagrams	1	2	2	1	1
2.3.3 Dependency Analysis	1	2	1	1	1
2.3.4 Qualitative Causal Reasoning Analysis	1	2	1	1	1
2.3.5 Look-up Table Generator	1	4	3	1	1
2.3.6 Data Dictionary Generator	1	2	3	1	1

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

Table 7.3-1 (Continued)[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
2.3.7 Cross-Reference List Generator	1	3	2	1	1
2.3.8 Aliasing Analysis	1	4	2	1	1
2.3.9 Concurrency Analysis	1	4	1	1	1
2.3.10 Database Analysis	1	3	1*	1	1
2.3.11 Database Interface Analyzer	1	4	1*	1	1
2.3.12 Data-Model Evaluation	1	4	1*	1	1
2.4.1 Failure Mode, Effects, Causality Analysis	1	3	2	1	1
2.4.2 Criticality Analysis	1	3	2	1	1
2.4.3 Hazards/Safety Analysis	1	3	2	1	1
2.4.4 Anomaly Testing	1	2*	1	1	1
2.4.5 Fault-Tree Analysis	1	3	2	1	1
2.4.6 Failure Modeling	1	3	1	1	1
2.4.7 Common-Cause Failure	1	3	2	1	1
2.4.8 Knowledgebase Syntax Checking	4	1	4	4	1*
2.4.9 Knowledgebase Semantic Checking	4	1	4	4	1*
2.4.10 Knowledge Acquisition/Refinement Aid	4	1	4	4	1
2.4.11 Knowledge Engineering Analysis	4	1	4	4	1
2.5.1 Informed Panel Inspection	1	2	1	1	1

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

Table 7.3-1 (Continued)[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
2.5.2 Structured Walkthroughs	1	3	1	1	1
2.5.3 Formal Customer Review	1	4	1	1	1
2.5.4 Clean-room Techniques	1	4	1	1	1
2.5.5 Peer Code-Checking	1	3	1	1	1
2.5.6 Desk Checking	1	1	1	1	1
2.5.7 Data Interface Inspection	1	2	1*	1	1*
2.5.8 User Interface Inspection	1	2	1*	1	1*
2.5.9 Standards Audit	1	2	1*	1	1*
2.5.10 Requirements Tracing	1*	1*	1*	1*	1*
2.5.11 Software Practices Review	2	2	1	1	1
2.5.12 Process Oriented Audits	1	1	1	1	1
2.5.13 Standards Compliance	1	1	1	1	1
2.5.14 System Engineering Review	1	2	1	1	1
3.1.1 Unit/Module Testing	1	2	1	1	1
3.1.2 System Testing	1	3	1	1	1
3.1.3 Compilation Testing	1	4	2	1	1
3.1.4 Reliability Testing	1	4	2	1	1
3.1.5 Statistical Record-Keeping	1	4	2	1	1
3.1.6 Software Reiliability Estimation	1	4	2	1	1
3.1.7 Regression Testing	1*	1*	1*	1*	1*
3.1.8 Metric-Based Testing	2	2	3	1	1
3.1.9 Ad Hoc Testing	4	4	4	4	4

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

Table 7.3-1 (Continued)[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
3.1.10 Beta Testing	1	1	1	1	1*
3.2.1 Random Input Testing	1	1	1	1	1
3.2.1.1 Uniform Whole Program Testing	1*	1*	1*	1*	1*
3.2.1.2 Uniform Boundary Testing	1	1	1	1	1
3.2.1.3 Gaussian Whole Program Testing	1	1	1	1	1
3.2.1.4 Gaussian Boundary Testing	1	1	1	1	1
3.2.2 Domain Testing	1	1	1	1	1
3.2.2.1 Equivalence Partitioning	1	1	1	1	1
3.2.2.2 Boundary-value Testing	1	1	1	1	1
3.2.2.3 Category-Partition Method	1	1	1	1	1
3.2.2.4 Revealing Subdomains Method	1	1	1	1	1
3.3.1 Specific Functional Requirement Testing	1*	1*	1*	1*	1*
3.3.2 Simulation Testing	1*	3	1	1	1
3.3.3 Model-Based Testing	1	2*	1	1	1
3.3.4 Assertion Checking	1	4	2	1	1
3.3.5 Heuristic Testing	1	3	1	1	1*
3.4.1 Field Testing	1	1	1	1	1
3.4.2 Scenario Testing	1	1	1	1	1
3.4.3 Qualification/Certification Testing	1*	1	1	1*	1

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

Table 7.3-1 (Continued)[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
3.4.4 Simulator-Based Testing	1	3	1	1	1
3.4.5 Benchmarking	1*	3	1	1*	1
3.4.6 Human Factors Experimentation	1	1*	1*	1	1
3.4.7 Validation Scenario Testing	3	2	2	4	1
3.4.8 Knowledgebase Scenario Generation	4	1	4	4	4
3.5.1 Stress/Accelerated Life Testing	1	3	1	1	1
3.5.2 Stability Analysis	1	2	1	1	1
3.5.3 Robustness Testing	1	2	1	1	1
3.5.4 Limit/Range Testing	1	1*	1	1	1
3.5.5 Parameter Violation	1	2	1	1	1
3.6.1 Sizing/Memory Testing	1	4	1	1	1
3.6.2 Timing/Flow Testing	1	4	1	1	1
3.6.3 Bottleneck Testing	1	4	1	1	1
3.6.4 Queue Size, Register Allocations, Paging, Etc.	1	4	2	1	1
3.7.1 Activity Tracing	1	2*	1	1	1
3.7.2 Incremental Execution	1	2	1	1	1
3.7.3 Results Monitoring	1	2	1	1	1
3.7.4 Thread Testing	1	3	1	1	1
3.7.5 Using Generated Explanations	4	1	4	4	4
3.8.1 Gold Standard	1	1	1	1	1

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

Table 7.3-1 (Continued)[1, 2]

Conventional V&V Testing Technique	Expert System Components				
	Inference Engine	Knowledge Base	Inter-faces	Tools & Utilities	Total System
3.8.2 Effectiveness Procedures	1	1	1	1	1
3.8.3 Workplace Averages	1	1	1	1	1
3.9.1 Data Interface Testing	1	2	1*	1	1
3.9.2 User Interface Testing	1	1	1*	1	1
3.9.3 Information System Analysis	1	4	1	1	1
3.9.4 Operational Concept Testing	1	2	1	1	1
3.9.5 Organizational Impact Analysis/Testing	1	4	2	1	1
3.9.6 Transaction-Flow Testing	1	2	1	1	1
3.10.1 Branch Testing	1	2	1	1	1
3.10.2 Path Testing	1	2	1	1	1
3.10.3 Statement Testing	1	3	1	1	1
3.10.4 Call-Pair Testing	1	4	2	1	1
3.10.5 Linear Code Sequence and Jump	1	4	2	1	1
3.10.6 Test-Coverage Analyzer	1	2	1	1	1
3.10.7 Conditional Testing	1	1*	1	1	1
3.10.8 Data-Flow Testing	1	2	1	1	1
3.11.1 Error Seeding	1	1	1	1	1
3.11.2 Fault Insertion	1	2	1	1	1
3.11.3 Mutation Testing	1	2	1	1	1

[1] Ratings are on a 1-4 scale:
1 = the method can be used directly without any modification; 2 = the method largely applies, but some modifications are necessary
3 = the general concept of the method applies but extensive changes are needed; 4 = the method does not really apply at all

[2] Ratings are the subjective evaluation of the authors

* Technique with highest applicability rating

The certification problems are often considered and addressed for the tools and utilities used in conventional programming and they should be identified and followed for new products.[15] Two utilities, knowledge engineering tools and the expert system shell, have received very little certification activity. A suite of problems or benchmarks which has proven useful in finding problems in these types of components should be developed for certification.

Applications with high complexity and integrity requirements should also be directly tested by other means. Since their source code is unlikely to be available, one is restricted to using Dynamic Testing techniques. Of these, the most efficient techniques are Random Testing (3.2.1) and Robustness Testing (3.5.3). Again, these techniques are listed in Table 6.3-1. The criteria for selecting techniques would be the same as discussed in Section 7.3.1 if source code was available in fully custom-developed expert systems. The overall highest applicability ratings in Table 7.3-1 for Tools and Utilities are for the following conventional V&V techniques: Requirements Tracing Analysis (1.4.1), Requirements Tracing (2.5.10), Regression Testing (3.1.7), Uniform Whole Program Testing (3.2.1.1), Specific Functional Requirement Testing (3.3.1), Qualification/Certification Testing (3.4.3), and Benchmarking (3.4.5).

7.3.3 Methods Applicable to the Inference Engine Component

The two unique expert system components have been left for last. While the inference engine is most often thought of in connection with the knowledge base it is very unlike that component in terms of applicability of conventional V&V techniques. Virtually all of these techniques apply to the inference engine, but only a few fully apply to the knowledge base. The inference engine subcomponents mostly tend to be written in procedural code, and when these have high reusability across applications the certification procedures discussed above in 7.3.2 should be followed, preferably by the vendor of the inference engine. Of course, if the vendor has not certified the inference engine, or if it is custom-built, then it should be tested with individual techniques which have been selected according to the principles developed in 7.3.1.

In assuring the quality of the overall inference engine, it is necessary to have test cases which exercise all of the engine subcomponents, particularly with problems which test the various proof procedures and all the knowledge management and associated goal-reasoning aspects. What is particularly needed is a set of special problems, e.g., planning, which are known to have complicated chains of reasoning, involve the need to re-inference, undo prior reasoning, and are appropriate for various kinds of decision-methods. A number of workers in the field of V&V are aware of this need, but very little work has yet been performed.

The overall highest technique ratings for testing the Inference Engine are found in Table 7.3-1. They are the following: Requirements Tracing Analysis (1.4.1), Data Flow Analysis (2.3.1), Requirements Tracing (2.5.10), Regression Testing (3.1.7), Uniform Whole Program Testing (3.2.1.1), Specific Functional Requirement Testing (3.3.1), Simulation Testing (3.3.2), Qualification/Certification Testing (3.4.3), and Benchmarking (3.4.5).

---

[15] This raises the question for expert (and conventional) systems used in regulated applications whether or not regulatory concerns should be extended to all commercial or custom tools used in the construction of these systems.

7.3.4 Methods Applicable to the Knowledge Base Component

The knowledge base component of expert systems is the least addressed by conventional V&V techniques. It is difficult to test the function of how knowledge is employed or how the inference engine uses the knowledge. Only a few of the conventional static and dynamic techniques can be used. Nevertheless, if the most applicable techniques were used, it is believed that the knowledge base would be borderline tested for Class 3 V&V systems, and partially tested for Class 2 V&V ones. The highest rated V&V testing techniques for the Knowledge Base in Table 7.3-1 are: Requirements Tracing Analysis (1.4.1), State Transition Diagram Analysis (2.2.2), Anomaly Testing (2.4.5), Requirements Tracing (2.5.10), Regression Testing (3.1.7), Uniform Whole Program Testing (3.2.1.1), Specific Functional Requirement Testing (3.3.1), Model-Based Testing (3.3.3), Human Factors Experimentation (3.4.6), Limit/Range Testing (3.5.4), Activity Tracing (3.7.1), and Conditional Testing (3.10.7).

7.3.5 Methods Applicable to Overall System V&V

All conventional V&V techniques apply fully to expert systems when viewed as total systems[16]. Since three of the four components involve conventional procedural languages, many of the techniques apply at the systems-testing level. Although the conventional techniques are appropriate for the overall system, an expert system cannot be considered completely assessed until there are better testing methods for the knowledge base component, particularly for Class 1 V&V type applications.

In Table 7.3-1, the highest rated techniques for overall expert system testing are the following: Formal Requirements Review (1.3.1), Formal Design Review (1.3.2), System Engineering Analysis (1.3.3), Requirements Analysis (1.3.4), Requirement Tracing Analysis (1.4.1), Operational Concept Analysis (2.2.4), Knowledgebase Syntax Checking (2.4.8), Knowledgebase Semantic Checking (2.4.9), Heuristic Testing (3.3.5), Regression Testing (3.1.7), Beta Testing (3.1.10), Uniform Whole Program Testing (3.2.1.1), and Specific Functional Requirement Testing (3.3.1).

7.4 Limitations of Conventional V&V Methods

This section concludes with a discussion of the limitations of conventional techniques when testing expert systems and a proposed strategy for developing new testing techniques.

7.4.1 Aspects of Expert Systems Not Adequately Evaluated with Conventional Methods

Conventional V&V techniques are found to be appropriate for testing three of the four components of expert systems. They also adequately test the overall system. Therefore, conventional techniques are adequate to test expert systems for low and possibly moderate levels of system complexity and integrity, V&V Classes 3 and 2. However, conventional techniques are believed to be inadequate to fully test expert systems for the highest degrees of complexity and integrity since conventional techniques cannot fully test the knowledge base component without modification.

---

[16] An exception to this statement is **ad-hoc testing** (3.1.9). This technique, although widely used, is considered to be so inappropriate, wasteful of effort, and difficult to interpret that the poorest evaluations were assigned for it in Table 7.3-1.

There are three inadequate aspects of conventional techniques for the knowledge base: (1) it is not sufficiently tested as a separate component in the requirements, design or the implementation phases; (2) it is not tested in controlled dynamic interaction with the inference engine; and (3) it is not fully tested in performance in interaction with the interfaces as an integral part of the overall system. However, some suggestions for extending conventional V&V techniques to the knowledge base are provided.[17]

Since the knowledge base contains most of the application-dependent information, it should be analyzed on the basis of the requirements specification and its subsequent design, which was developed in the early phases of the Life-cycle. However, neither the V&V classes/subclasses formal methods (1.1) nor the semi-formal methods (1.2; Table 5.2.1-1) are oriented towards specification or design of rule bases, object-oriented knowledge structures, or sets of knowledge frames[18]. Each of these major knowledge representations has a host of specific AI or expert system engineering considerations associated with it (Wolfgram et al, 1987, for discussion of design of rules and frames; and Booch, 1990, for object-oriented design).

Some appropriate extensions of the formal and semi-formal methods are possible. However, relying heavily on formal methods is questionable. It is difficult with these methods to express system concepts formally and move from representations to a mapping onto the proposed physical components while maintaining traceability. Nevertheless, it does seem that the formal methods of EHDM (1.1.3) and Z (1.1.4) would be among the most appropriate to consider for development of formalism which would extend them to knowledge base specification and design-representation testing. The authors are more optimistic about proposing extensions and adaptations to the SREM methodology (1.2.6) of the Semi-Formal methods, particularly as represented in the RDD-100 tool, for analysis of knowledge base specifications and representation of its design. This methodology, if augmented by proper systems engineering discipline, would permit smooth movement from requirements specification representation into an allocation of requirements to logical functions and finally to a mapping of these onto physical components broken down between software and hardware. Simulation or "animation" of design is easily accomplished, and there are many automated formal checks for inconsistency, contraction, and incompleteness.

During implementation, both static and dynamic conventional techniques are deficient for singularly testing the knowledge base component. Suitable static examination of the knowledge base could reveal a great deal about the effects of data inputs on transitioning to new states, and thus extensions to the State Transition Diagram Analysis technique (2.2.2) could be especially useful for applying to rule bases and objects. Perhaps the most powerful and useful extension would be to Anomaly Testing (2.4.5) to search for various kinds of problems of rules, frames, and objects. Some of the promising expert system rule-base automated syntax checkers can be considered extensions of this method (e.g., D-EVA, COVER, and CRSV; see, respectively: Stachowitz et al; 1987; Preece & Shinghal, 1991; and Culbert et al, 1987).

---

[17] These suggestions are from the constrained point of view of adapting conventional techniques to knowledge base testing. However, these suggestions may be quite limited, since there may well be novel expert system testing approaches already in existence, or which could be developed, that might not fit at all under conventional technique categories. This issue will be addressed in subsequent tasks of this project.

[18] Many expert system shells do provide good support for documenting and handling various types of knowledge representations, but this is intended primarily for the system implementation, and possibly the design, not for requirements.

For dynamic testing of knowledge bases the Heuristic Testing method itself (3.3.5) or extensions of Conditional Testing (3.10.7) or its close associate, the Limit/Range testing method (3.5.4), would seem especially appropriate for rules and frames. The IF part of rules involves a relational test (e.g., A>=B, C~=D), while frames specify the equivalence between an attribute and a set of values (e.g., F = <1,2,3>; G = 'Open'). Test-case generation can be automated for these tests by selecting values in relation to those given: (1) from within the specified values, such that the test is true (e.g., A>B, F=1, G = 'Open'); (2) just outside the range or value specified (e.g., A<B, C=D, G='Oper'); and (3) extremely outside the value/range (e.g., A=B/1000, F=9999, G='XXXX'). For a discussion of this approach to rule testing, and its relation to reliability assurance, see the discussion of "Expert Systems Dynamic Testing" (Miller, 1990).

These three sets of extensions seem to be the most important for assurance of the quality of the knowledge base component in isolation from all the used data and other interfaces. The only way to determine how the knowledge base will interact with the inference engine and the interfaces is by running the inference engine on the knowledge base with active or simulated interfaces to determine the rule firings, frame activations, facts utilization, or object-messages. A modification of one of the methods of Execution Tracing (3.7), such as Activity Tracing (3.7.1), would provide one of the most appropriate ways of testing this interaction. The Activity Tracing would involve simulation of the other components in special test-drivers written for this assessment, so as not to actually require interfaces, tools, or utilities to be activated, so that the "pure" interaction of the knowledge base and the inference engine could be tested. Model-based Testing technique (3.3.3) could also be extended to apply to expert systems. Also, the Functional Requirement Testing (3.3.1) can be utilized, using test-cases obtained from the prior types of testing.

7.4.2 A Proposal for a Generic Testing Strategy

A different classification of testing techniques is presented in this section. This two dimensional classification approach provides a strategic direction for the evolution of new methods for software systems.

In the first classification dimension, a distinction is made in the technique classification. Methods involve either a static examination of the development artifacts (requirements, specification, design, or implementation) or they involve a dynamic execution of the implemented system on some physical platform and operating system for real data inputs. The second classification dimension concerns the targets or objects of detection techniques. It is useful to characterize these as either **anomalies**, which are defined as unusual, deviant, improper, or nonsensical situations; or **invalidates**, which are defined as known incorrect values, definite errors, mistakes, or false representations of external situations.

Since both static and dynamic testing techniques can be applied to both anomalies and invalidates, both classifications can be represented as a 2 x 2 table, as shown in Table 7.4.2-1. The chart provides plus signs (+) and minus signs (-) for each cell combination to indicate positive and negative features of that condition. The following summary observations are based on this table:

1) Some dynamic testing is essential to test run-time system performance and for full customer acceptance.

2) Dynamic testing costs more to accomplish than static testing because of longer technique-learning times, time to generate test-cases, setup time, execution time, and time tointerpret results.

Table 7.4.2-1. Characterization of Techniques for Testing Implemented Systems in Terms of Technique Type and Target Type

Testing Technique	Testing Target: Anomalies	Testing Target: Invalidities
Static	(+) Can by formalized (+) Can be automated (+) Not difficult to understand, run, or interpret (+) Don't require experts (+) Can reduce level of dynamic testing needed (+) Greatly reduce overall testing costs (-) Not sufficient for complete testing	(-) More difficult to detect (+) Some inspection approaches (e.g., Clean Room, 2.5.4) very effective (-) Requires experts to make the judgement (-) Judgements often need run-time context information to make
Dynamic	(+) Test-case generation is simple (+) Detection, during testing, can be greatly automated (+) Don't require experts (+) Especially good for performance problems and run-time violations (e.g., type errors) (-) Moderately expensive to set up and run (-) Run-time anomalies not caught by compilers more difficult to define and develop code for detection	(-) Requires experts to evaluate (-) Labor-intensive and expensive to learn how to do, to generate test cases to set up, to run, and to interpret (+) Some amount is essential to assess whether system is giving right answers and for customer acceptance of testing results (-) Run-time output is usually very complex, making interpretation difficult and errors likely

3) Since anomalies can be formally specified and automated, they are much cheaper and easier to look for than invalidates.

Based on the above observations, the following strategy is proposed for shifting future testing resources and development:

1) Wherever possible, within a technique type, shift the detection target from an invalidity to an anomaly. Provide the system with independent information about the results so that the system can automatically compare its results to these values to detect inconsistencies. For example, give expected ranges and ballpark figures to provide known values of constants that must appear in the output.

2) Wherever possible, within a target type, shift the technique from dynamic testing to static testing by using any variety of inspection, semi-formal, or other static techniques.

3) Give priority to the most important problems/faults to be detected and always conduct testing so that the subsequent testing activity focuses on the remaining most important or untested problem.

4) For maximum cost-effectiveness in use of testing resources always start with static testing techniques; look for anomalies, then investigate invalidates; fix the problems encountered; and use dynamic testing as a last step.

8 SUMMARY and CONCLUSIONS

This report presented the results of a detailed survey of V&V methods currently used for conventional software as the first activity of a project for developing expert system V&V guidelines which is sponsored by the USNRC and EPRI. This survey resulted in the identification of 153 different methods.

The 153 conventional software V&V techniques were classified using a sequential Life-cycle model for the process of software development and maintenance. These techniques were categorized as either in the requirements/design phase or the implementation phase of the software. This first level of classification divided the 153 methods into 28 applicable to the requirements/design phase and 125 applicable to the implementation phase. This skewed distribution of V&V methods reflects the emphasis on testing software after it has been designed rather than during its design phase. In analyzing the 125 implementation phase methods, 58 were determined to involve static testing while the remaining 67 fall within the category of dynamic testing. Static testing deals with software V&V that requires inspecting software documentation without actually running the software while dynamic testing does rely on executing the software.

In order to assess the effectiveness of the 153 identified conventional software V&V techniques, different types of software defects were identified. A total of 52 different types of conventional software defects were categorized as either pertaining to requirements, design, or the computer code itself. These 52 defects are divided into 13 requirements, 15 design, and 24 code types of defects. Then, each of the 153 V&V techniques were evaluated to determine which of these 52 defects they were capable of detecting. Individual V&V techniques were found to be able to detect anywhere from 2 to 52 software defects. It was also found that each software defect was detectable by anywhere from 21 to 50 different V&V methods.

A method to rate each V&V method was developed based on eight factors. Four factors (Broad Power, Hard Power, Formalizability, and Human-Computer Interface Testability) deal with the power of the technique to detect defects while the other four (Ease of Mastery, Ease of Setup, Ease of Running/Interpretation, and Usage) are related to ease-of-use considerations for each technique. The rating of each V&V technique by these eight factors was then used to determine a cost-benefit and effectiveness measure.

Since the end result of this project is to provide V&V guidelines for a wide range of expert systems in the nuclear industry, a scheme was developed for classifying expert system applications in terms of their needed V&V. This scheme used two variables: software complexity and the required integrity of the software. Software with both high integrity and high complexity was placed in V&V Class 1. A medium level of integrity and complexity resulted in V&V Class 2 while a low level of complexity and integrity was used to define V&V Class 3. For each of these three V&V Classes, the 153 techniques were rated by their effectiveness as well as cost-benefit.

As a result of the aforementioned rating of V&V techniques by cost-benefit and effectiveness for each software V&V class, the highest ranking methods were determined in requirements/design, static, and dynamic testing. For requirements/design V&V, the most effective and cost beneficial methods for all V&V classes were found to be a combination of traditional formal reviews and some new automated techniques. In the case of static testing, the highest ranking techniques for all V&V classes involved personal inspections and reviews without the aid of any automation.

Finally, in the area of dynamic testing, the leading V&V techniques for all V&V classes were system-level function-oriented methods that did not involve testing internal features of the software code, i.e., "black box" testing.

After classifying and evaluating conventional software V&V techniques, their applicability to expert systems was assessed. Expert systems were first divided into four basic components: knowledge base, inference engine, interfaces, and tools and utilities. Each of these four components were further divided into subcomponents which were analyzed to determine their features related to testing. The three testing features are: written in or involving procedural language, high reusability across different applications, and potential defects known and amenable to formal test methods.

Each of the 153 conventional software V&V techniques were evaluated in terms of their applicability to the four expert system components. Most of the conventional software V&V techniques were found to be directly applicable to the inference engine, interfaces, and tools-utilities components of expert systems. The conventional software V&V techniques that were judged to be most applicable to all four expert system components were: requirements tracing, regression testing, uniform whole program testing, and specific functional requirements testing.

Conventional software V&V techniques were found to be directly applicable to expert systems, but not completely adequate in finding defects in the knowledge base. These techniques cannot adequately test the knowledge base as a separate component or in dynamic interaction with the inference engine. However, several extensions of conventional techniques were suggested for testing the knowledge base component.

An overall scheme for software testing was developed and presented that is equally applicable to conventional and expert system software. Testing is divided into two types: static and dynamic. These two different types of testing are distinguished by whether they require actually running the software (dynamic) or just inspection and review of the software documentation (static). Static testing is generally less expensive that dynamic testing. Software defects are also divided into two types entitled anomalies and invalidates. Where invalidates are obvious errors or incorrect values, anomalies are unusual or nonsensical. Anomalies are usually easier to look for than invalidaties. A 2 x 2 table can be constructed with static and dynamic testing on one axis and invalidates and anomalies on the other. A test strategy with these parameters would be to emphasize static testing tor anomalies. After looking for anomalies, static testing should be used to search for invalidates. Dynamic testing should not be used until after static testing has been completed and defects have been corrected.

In conclusion, a large number of V&V techniques for conventional software were identified and evaluated for their capability in uncovering software defects. Conventional software V&V methods were found to be directly applicable to three of the four components of expert systems. These three components are the inference engine, external interfaces, and tools or utilities. The fourth expert system component, the knowledge base, is not fully tested with conventional V&V techniques. Therefore, the main emphasis of the balance of this project will be to identify and, if necessary, develop new V&V methods for the knowledge base component of expert systems and present guidelines for specific V&V techniques that should be applied to each software V&V class.

9 REFERENCES

Ackerman, F.A., L.S. Buchwald, and F.H. Lewski, *Software Inspections: An Effective Software Verification Process*, IEEE Software, Vol. 6, No. 3, May 1989.

Alford, M., *SREM At The Age of Eight: The Distributed Computing Design System*, Computer, 18 (4), pp. 36-46, 1985.

Alford, M. *A Requirements Engineering Methodology for Real-Time Processing Requirements*, IEEE Transactions on Software Engineering, SE-3(1), pp. 60-69, 1977.

ANSI/ANS-10.4-1987, *Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, May 13, 1987.

ANSI/IEEE ANS-7-4.3.2-1982, *Application Criteria for Programmable Digital Computer Systems of Nuclear Power Generating Stations*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, July 6, 1982.

ANSI/IEEE 1008-1987, *IEEE Standard for Software Unit Testing*, IEEE Standards Board, New York, New York, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, 1986.

ANSI/ANS-3.5-1985, *American National Standard Nuclear Power Plant Simulators for Use in Operator Training*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, October 25, 1985.

ANSI/IEEE 1012-1986, *Software Verification and Validation Plans*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, November 14, 1986.

ANSI/IEEE 729-1983, *Glossary of Software Engineering Terminology*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, February 18, 1982.

ANSI/IEEE 830-1984, *IEEE Guide to Software Requirements Specifications*, IEEE Standards Board, New York, New York, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, 1984.

ANSI/IEEE 1016-1987, *Recommended Practice for Software Design Description*, IEEE Standards Board, New York, New York, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, 1987.

ANSI/IEEE 828-1983, *Software Configuration Management Plans*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, June 24, 1983.

ANSI/IEEE 1042-1987, *Guide to Software Configuration Management*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, September 12, 1988.

ANSI/IEEE 829-1983, *Software Documentation*, American Nuclear Society, 555 North Kensington Ave., La Grange Park, Illinois, 60525, February 18, 1983.

Ascent Logic Corporation, *Requirements Driven Design*, RDD-100, Ascent Logic Technology, 180 Rose Orchard Way, San Jose, California, 95134, 1991.

ASME/NQA-2a-1990 Part 2.7., *Quality Assurance Requirements of Computer Software for Nuclear Facility Application*, The National Institute of Standards and Technology Computer Systems Laboratory, Gaithersburg, Maryland 20899, 1990.

Barnes, M., P. Bishop, B. Bjarland, G. Dahll, D. Esp., J. Lahti, H. Valisuo, and P. Humphreys, *Software Testing and Evaluation Methods (The STEM Project)*, Technical Report, OECD Halden Reactor Project, HWR-210, The Institutt for Energiteknikk, Halden, Norway, May 1987.

Barnes, M., P. Bishop, B. Bjarland, G. Dahll, D. Hufton, and H. Valisuo, *Software Testing and Evaluation Methods Final Report on the STEM Project*, Technical Report: OECD Halden Reactor Project, HPR-334, The Institutt for Energiteknikk, Halden, Norway, May 1988.

Barnes, M., P. Bishop, M. Brewer, P. Bradley, G. Dahll, F. Ross, and T. Sivertsen, *Safety Assessment of Programs (The SAP Project)*, Technical Report: OECD Halden Reactor Project, HWR-269, Institutt for Energiteknikk, Halden, Norway, January 1990.

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

Beltracchi, L., *Overview of Computer Standards and Tools in the European Nuclear Industry*, Presentation at the JTEC Workshop on Assessment of European Nuclear Controls and Instrumentation, National Science Foundation, Washington, D.C., January 31, 1991.

Berns, G.M., *Assessing Software Maintainability*, Communications of the ACM, Vol. 27, No. 1, pp. 14-23, January 1984.

Bishop, P., et al., *PODS -- A Project on Diverse Software*, IEEE Transactions on Software Engineering, Vol. SE-12 (9), ISBN 0098-5589, 1986.

Bishop, P., et al., *STEM -- A Project on Software Test and Evaluation Methods*, Paper presented at SARS '87 and published in Achieving Safety and Reliability with Computer Systems, B. Daniels (Ed.), ISBN 1-85166-167 0, Elsevier Applied Science,New York, New York, 1987.

Boehm, B.W., *A Spiral Model of Software Development and Enhancement*, IEEE Computer, pp. 61-72, May 1988.

Boehm, B.W., *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

Booher, H. (Ed.), *MANPRINT: An Approach to Systems Integration* Van Nostrand Reinhold, New York, New York, 1990.

Booth, G., *Object Oriented Design, With Applications*, Benjamin/Cummings Publishing Co., Redwood City, California, 1991.

Borgida, A., S. Greenspan, and J. Mylopolous, *Knowledge Representation as the Basis for Requirements Specifications*, Computer, 18(4), pp. 82-91, 1985.

Bryan, W.L., and S.G. Siegal, *Software Product Assurance: Techniques for Reducing Software Risk*, Elsevier, New York, New York, 1988.

BSI, *Standards Guide to Assessments of Reliability of Systems Containing Software: Draft for Development*, Document 89/97714, BSI Standards, 2 Park Street, London W1A 2BS, September 12, 1989.

Carre, B., et al., *SPADE -- the Southampton Program Analysis and Development Environment*, Published in Software Engineering Environment, I. Sommerville (Ed.), IEEE Computing, Series 7, 1986.

Chapanis, A., *Man-Machine Engineering* Brooks/Cole Publishing Company, Monterey, California, 1965.

Chen, R., *The Integration of the Air Force Content Data Monel and MIL-STD-1833-2B*, David Taylor Research Center Report No. DTRCC-90/034, Carderock Division, Naval Surface Warfare Center, Code 3323, Bethesda, Maryland 20084-5000, 1990.

Chisholm, G.H., B.T. Smith, and A.S. Wojcik, *Formal System Specifications - A Case Study of Three Diverse Representations*, ANL-90/43, The Mathematics and Computer Science Division and The Reactor Analysis Division, Argonne National Laboratory, Argonne, Illinois, 60439, December 1990.

Culbert, C., G. Riley, and R.T. Savely, *Approaches to the Verification of Rule-Based Expert Systems*, SOAR '87 First Annual Workshop on Space Operations Automation and Robotics, SCAMC, Inc., August 1987.

Dahll, G., and J.E. Sjoberg, *Software Safety Tools - The SOSAT 2 Project, Technical Report: OECD Halden Reactor Project, HWR-268*, Institutt for Energiteknikk, Halden, Norway, January 1990.

Davis, R., B. Buchanan, and E.H. Shortliffe, *Production rules as a representation for a Knowledge-Base Consultation Program*, Artificial Intelligence, pp. 15-45, August 8, 1977.

Davis, A.M., *Software Requirements: Analysis and Specification*, Prentice-Hall, Inc., New York, New York, 1990.

Department of Defense, *Software Master Plan, Volume 1: Plan of Action, Preliminary Draft*, Department of Defense, Washington, D.C. 20362, February 9, 1990.

Department of Defense, *Military Standard 2167, Defense System Software Development*, Department of Defense, Washington, D.C. 20362, June 4, 1985.

Desimone, R., and J. Rininger, *Expert System Validation and Verification*, SRI International, Contract DAAB07-86-D-A035, SRI Project 3002, Final Report, SRI International, Menlo Park, California 94025, August 1990.

Deutsch, M., *Software Verification and Validation: Realistic Project Approaches*, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1982.

Downs, T., *An Approach to the Modeling of Software Testing With Some Applications*, IEEE Transaction on Software Engineering, SE-11(4), April 1985.

Doyle, J., *Methodological Simplicity in Expert System Construction: The Case of Judgements and Reasoned Assumptions*, The Artificial Intelligence Magazine, 4(2), pp. 39-43, 1983.

Dunn, R., *Software Defect Removal*, McGraw- Hill, New York, New York, 1984.

Eason, K.D., *Dialogue Design Implications of Task Allocation Between Man and Computer*, Ergonomics, 3(9), pp. 881-891, 1990.

Ehrig, H., and B. Mahr, *Fundamentals of Algebraic Specification*, Springer Verlag, New York, New York, 1985.

Electric Power Research Institute, *Verification and Validation of Expert Systems for Nuclear Power Plant Applications, Final Report, NP-5978*, The Electric Power Research Institute, Palo Alto, California 94303, August 1988.

Fagan, M.E., *Advances in Software Inspection*, IEEE Transactions on Software Engineering, SE-12(7), pp. 744-751, July 1986.

Ghezzi, C., D. Mandrioli, S. Morasea, and M. Pezze, *A General Way to Put Time in Petri Nets (as in Kramer)*, ACM Order Department, P.O. Box 64145, Baltimore, Maryland 21264, pp. 60-67.

Gilmore, W.E., *Human Engineering Guidelines for the Evaluation and Assessment of Video Display Units*, NUREG/CR-4227, United States Nuclear Regulatory Commission, July 1985.

Gilmore, W.E., D.I. Gertman, and H.S. Blackman, *The User-Computer Interface in Process Control*, Academic Press, Boston, Massachusetts, 1989.

Goodenough, J., and S. Gerhart, *Toward a Theory of Test Data Selection*, IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, 1975.

Gordon, M., *HOL: A Machine Oriented Formulation of Higher Order Logic*, Technical Report No. 68, University of Cambridge, United Kingdom, 1985.

Gould, J.D., and C. Lewis, *Designing for Usability: Key Principles and What Designers Think*, Communications of the ACM, Volume 28, pp. 300-311, 1985.

Halstead, M., *Elements of Software Science*, Elsevier North-Holland, New York, New York, 1977.

Harel, D., H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, and A. Shtul-Trauring, *STATEMENT: A Working Environment for the Development of Complex Reactive Systems*, Proceedings of the 10th International Conference on Software Engineering, Singapore, IEEE Computer Society Press, 1730 Massachusetts Ave., N.W., Washington, D.C. 20036-1903, pp. 396-406, April 1988.

Harel, D. and S. Rolph, *Modeling and Analyzing Complex Reactive Systems: The Statement Approach*, Logix, Inc., Burlington, Massachusetts, 1989.

Harel, D., *Statecharts: A Visual Formalism For Complex Systems*, Science of Computer Programming 8, North-Holland Elservier, New York, New York, pp. 231-274, 1987.

Hartway, B., J. Young, and D. Thomas, *Simulation Characterization*, Proceedings of Third International Conference on Software for Strategic Systems, 27-28 February 1990, Huntsville, Alabama, pp. 64-85.

Hasling, D.W., *Abstract Explanations of Strategy in a Diagnostic Consultation System*, Proceedings of the National Conference on Artificial Intelligence, AAAI-83, The MIT Press, Cambridge, Massachusetts 02142, 1983.

Hatley, D. and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, New York, 1987.

Hayakawa, H., K. Monta, T. Sato, and M. Tani, *Concepts of Integrated Information and Control Systems for Future Nuclear Plants*, IAEA International Conference on Man-Machine Interface in the Nuclear Industry, Tokyo, The International Atomic Energy Agency, Wagramerstrasse 5, P.O. Box 100, A-1400, Vienna, Austria, February 1988.

Hayes-Roth, F., D.A. Waterman, and D.B. Lenat, *Building Expert Systems*, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts, Chapter 8, pp. 241-280, 1983.

Heninger, K., et al., *Specifying Software Requirements for Complex Systems: New Techniques and Their Application*, IEEE Transactions on Software Engineering, 1980.

Hill, J.V., *Software Development Methods in Practice*, Elsevier Applied Science, New York, New York, 1991.

Hoare, C., *Communicating Sequential Processes*, Prentice-Hall International, New York, New York, 1986.

Hoare, C. and J. Shephardson (Eds.), *Mathematical Logic and Programming Languages*, Prentice-Hall, New York, New York, 1985.

Howden, W.E., *Functional Program Testing*, IEEE Transactions on Software Engineering. SE-6(2), pp. 162-169, March 1980.

Humphrey, W.S., *Managing the Software Process*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990.

IEC 880, *Software for Computers in the Safety Systems of Nuclear Power Stations*, Bureau Central de la Commission Electrotechnique Internationale, 3 rue de Varemoe, Geneve, Suisse, 1986.

Ince, D.C., *The Automatic Generation of Test Data*, Computer Journal, 30(1), pp. 63-69, February 1987.

ISO Draft International Standard, *Information Processing Systems -- Open Systems Inter-Connection -- Lotos -- A Formal Description Technique Based on the Temporal Ordering of Observational Behavior*, ISO/TC 97/SC 21, ISO DIS 8807, The National Institute of Standards and Technology Computer Systems Laboratory, Gaithersburg, Maryland 20899, July 20, 1987.

Jackson, P., and P. Lafrere, *On the Application of Rule-Based Techniques to the Design of Advice-Giving Systems*, International Journal of Man-Machine Studies, 20(1), pp. 63-68, 1984.

Jagodzinski, A.P., *A Theoretical Basis for the Representation of On-Line Computer Systems to Naive Users*, International Journal of Man-Machine Studies, Volume 18, pp. 215-252, 1983,

Jensen, R., and C. Tonies, *Software Engineering*, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

Jensen, H., and K. Vairavan, *An Experimental Study of Software Metrics for Real-Time Software*, IEEE Transactions on Software Engineering, Vol. SE-11(2), pp. 231-234, 1985.

Jones, C., *Systematic Software Development Using VDM*, Prentice-Hall International, New York, New York, 1986.

Jones, T.C., *Programming Productivity*, McGraw- Hill, New York, New York, 1986.

Kidd, A.L., and M.B. Cooper, *Man-Machine Interface Issues in the Construction and Use of an Expert System*, International Journal of Man-Machine Studies, Volume 22, pp. 91-102, 1985.

King, J.C., *Symbolic Execution and Program Testing*, Communications of the ACM, 19(7), pp. 385-394, July 1976.

Knowledge CASE Tool, 1650 Tyson Blvd., Suite 800, McLean, Virginia 22102 (703) 506-0800.

Koch, C.G., *User Interface Design for Maintenance/Troubleshooting Expert System*, Proceedings of the Human Factors Society, 29th Annual Meeting, pp. 367-371, ACM Order Department, P.O. Box 64145, Baltimore, Maryland 21264, 1985.

Kramer, J., J. Magee, and M. Sloman, *Configuration Support for System Description, Construction and Evolution*, Proceedings of the Fifth International Workshop on Software Specification and Design, 19-20 May 1989, Pittsburgh, Pennsylvania, ACM SIGSOFT Engineering Notes, Vol. 14 No.3 pp. 28-33, IEEE Computer Society, Order Department, 10662 Los Vaqueros Circle, Los Alamitos, California 90720-2578, 1983.

Lapassat, A.M., *Real Time Systems Software Validation and Verification*, Commissariat a L'Energie Atomique, France, Cen/Saclay - Irdi/D_leti/Dein/sir, 91191 Gif sur Yvette Cedex, France.

Lehner, P.E., and D.A. Zirk, *Cognitive Factors in User/Expert-System Interaction*, Human Factors, 29(1), pp. 97-109, 1987.

Leveson, N.G., and J.L. Stolzy, *Safety Analysis Using Petri Nets*, IEEE Transactions on Software Engineering, SE-13(3), 1987.

Leveson, N.G., and P.R. Harvey, *Analyzing Software Safety*, IEEE Transactions on Software Engineering, SE-9(5), pp. 569-579, September 1983.

Liverpool Data Research Associates Ltd., *LDRA Software Tested, FORTRAN*, User Documentation, Liverpool, United Kingdom, 1985.

Llinas, J., S. Rizzi, and M. McCown, *The Test and Evaluation Process for Knowledge-Based Systems*, Technical Report of Science Applications International Corporation, San Diego, California, June 1987.

McCabe, T., *A Complexity Measure*, IEEE Transactions on Software Engineering, Vol. SE-2(4), pp. 308-320, 1976.

Miller, E., *Better Software Testing*, Proceedings of Third International Conference on Software for Strategic Systems, February 27-28, 1990,pp. 1-7, Huntsville, Alabama, 1990.

Miller, L.A., *Behavioral Studies of the Programming Process*, IBM Research Report, Rc7367, International Business Machines (IBM), Yorktown Heights, New York, 1978.

Miller, L.A., *Testing and Evaluation of Expert Systems*, Paper distributed at the Fourth IEEE Conference on AI Applications, San Diego, California, The Computer Society of the IEEE, P.O. Box 80452, Worldway Postal Center, Los Angeles, California 90080, March 18, 1988.

Miller, L.A., *Dynamic Testing of Knowledge Bases Using the Heuristic Testing Approach. Expert Systems with Applications: An International Journal*, Special Issue: Verification and Validation of Knowledge-Based Systems, Vol. 1, No. 3, pp. 249-269, 1990.

Miller, L.A., *Tutorial on Validation and Verification of Knowledge-Based Systems*, Proceedings of the Conference on Expert Systems Applications for the Electric Power Industry, Science Applications International Corporation, 1710 Goodridge Drive, McLean, Virginia 22102, June 1989.

Miller, L.A., *Verification and Validation of Expert Systems*, Invited paper presented at the United States Army Test and Evaluation Command Conference on AI, Sierra Vista, Arizona, Science Applications International Corporation, 1710 Goodridge Drive, McLean, Virginia 22102, January 15, 1992.

Miller, L.A., *A Realistic Industrial-Strength Life-cycle Model for Knowledge-Based System Development and Testing*, Knowledge Based Systems Verification and Validation Workshop Proceedings, AAAI-90, The MIT Press, Cambridge, Massachusetts 02142, July 1990.

Mills, H., V. Basili, J. Gannon, and R. Hamlet, *Principles of Computer Programming: A Mathematical Approach*, William C. Brown, New York, New York, 1987.

Milner, R., *A Calculus of Communicating Systems*, Laboratory for the Foundations of Computer Science, Edinburgh University Report No. ECCS-LFCS-86-7, University of Edinburgh, Scotland, 1986.

Montalban, M., *Decision Tables*, Science Research Associates, Inc., Chicago, Ilinois, 1974.

Myers, G.J., *The Art of Software Testing*, Wiley, New Yor, New York, 1979.

Naser, J.A. (Ed.), *Expert System Applications for the Electric Power Industry*, Hemisphere Publishing Corporation, New York, New York, 1991.

National Aeronautical and Space Administration, *Space Station Freedom Program Human-Computer Interface Guidelines*, NASA USE 1000, Version 2.1, NASA, Reston, Virginia, 1989.

NBS 500-93, *Software Validation, Verification, and Testing Technique and Tool Reference Guide*, The National Institute of Standards and Technology Computer Systems Laboratory, Gaithersburg, Maryland 20899, September 1982.

NBS 500-75, *Validation, Verification, and Testing of Computer Software*, The National Institute of Standards and Technology Computer Systems Laboratory, Gaithersburg, Maryland 20899, February 1981.

NBS-500-98, *Planning for Software Validation, Verification and Testing*, The National Institute of Standards and Technology Computer Systems Laboratory, Gaithersburg, Maryland 20899, November 1982.

Nelson, W.R., and H.S. Blackman, *Response Tree Evaluation: Experimental Assessment of an Expert System for Nuclear Reactor Operators*, NUREG/CCR-4272, United States Nuclear Regulatory Commission, September 1985.

Ng, P., and R. Yeh (Eds.), *Modern Software Engineering: Foundations and Current Perspectives*, Van Nostrand Reinhold, New York, New York, 1990.

Nicoud, J., and P. Fah, *Common Assembly Language for Micro-Processors, CAM3*, Draft 3.3, EFPL, Lausanne, Switzerland, 1983.

Norman, D.A., and S.W. Draper (Eds.), *User-Centered System Design: New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1986.

NSAC-39, *Verification and Validation for Safety Parameter Display Systems*, Nuclear Safety Analysis Center, Atlanta, Georgia, December 1981.

NUREG/CR-4640, *PNL-5784, Handbook of Software Quality Assurance Techniques Applicable to the Nuclear Industry*, August 1987.

NUREG/CR-4227, *Human Engineering Guidelines for the Evaluation and Assessment of Video Display Units*, W. Gilmore, July 1985.

NUREG-0653, *Report on Nuclear Industry Quality Assurance Procedures for Safety Analysis Computer Code Development and Use*, August 1980.

O'Leary, D.E., *Verification of Frame-based Knowledge Base*, Knowledge Based Systems Verification, Validation and Testing Workshop Proceedings, AAAI-90, The MIT Press, Cambridge, Massachusetts 02142, July 1990.

Oakes, L., *Transition from Analog to Digital Technologies*, Presentation at the JTEC Workshop on Assessment of European Nuclear Controls and Instrumentation, NSF, Washington, DC, The National Science Foundation, Washington, D.C. 20031, January 31, 1991.

Omar, A., and F. Mohammed, *A Survey of Software Functional Testing Methods*, ACM SIGSOFT Software Engineering Notes, Vol. 16, No. 2, pp. 75-82, 1991.

Ostrand, T.J., and M.J. Balcea, *The Category-Partition Method for Specifying and Generating Functional Tests*, Communications of the ACM, Vol 31, No. 6, pp. 676-686, June 1988.

Oyeleye, O., *Qualitative Modeling of Continuous Chemical Processes and Applications to Fault Diagnosis*, Ph.D Dissertation, Massachusetts Institute of Technology, February 1990.

Parnas, D.L., D.G. Smith, and T. Pearce, *Making Formal Software Documentation More Practical: A Progress Report*, Technical Report #88-236, ISSN 0836-0227, November 1988, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada K7L 3N7.

Parnas, D., and W. Bartussek, *Using Traces to Write Abstract Specifications For Software Modules*, Lecture notes in Computer Science (65), Information System Methodology Proceedings IOS, Springer Verlag, New York, New York, 1978.

Parnas, D.L., and P.C. Clements, *A Rational Design Process: How and Why to Fake It*, IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 251-257, February 1986.

Peters, L., *Timing Extensions to Structured Analysis for Real-Time System*, pp. 83-90, IEEE Computer Society, Order Department, 10662 Los Vaqueros Circle, Los Alamitos, California 90720-2578.

Preece, A.D., and R. Shinghal, *Practical Approach to Knowledge Base Verification*, Ed. M. Trivedi, Proceedings of Applications of Artificial Intelligence IX, pp. 608-619, Concordia University, Montreal, Canada H3G 1M8, April 1991.

Pritsker, A.A.B., *Introduction to Simulation and SLAMII*, John Wiley and Sons, New York, New York, 1986.

Rapps, S., and E.J. Weyuker, *Selecting Software Test Data Using Data Flow Information*, IEEE Transactions on Software Engineering, SE-11(4), pp. 367-375, April 1, 1985.

Rasmussen, J., and K.J. Vicente, *Cognitive Control of Human Activities and Errors: Implications for Ecological Interface Design*, Presented at The Fourth International Conference on Event Perception and Action, Trieste, Italy, August 24-28, 1987, Riso National Laboratory, Roskilde, Denmark, 1987.

Rattray, C. (Ed.), *Specification and Verification of Concurrent Systems*, Springer-Verlag, New York, New York, 1990.

Regulation Guide 1.52 (Task IC 127-5) *Criteria for Programmable Digital Computer System Software in Safety-Related System of Nuclear Power Plants*, United States Nuclear Regulatory Commission, November 1985.

Roe, R., and J. Rowland, *Some Theory Concerning Certification of Mathematical Subroutines by Black Box Testing*, IEEE Transactions Software Engineering, Vol. SE-13, No. 6, 1987.

Ross, D., *Structured Analysis (SA): A Language for Communicating Ideas*, IEEE Transactions on Software Engineering, SE-3(1), pp. 16-33 (SADT), 1977.

Rushby, J., F. von Henke, and S. Owre, *An Introduction to Formal Specification and Verification Using EHDM*, SRI International, SRI-CSL-91-02, CSL Technical Report, SRI International, Menlo Park, California 94025, February 1991.

Rushby, J., *Quality Measures and Assurance for AI Software*, NASA Contractor Report 4187, SRI International, Menlo Park, California 94025, October 1988.

Schnell, D.A., *Usability Testing of Screen Design: Beyond Standards, Principles, and Guidelines*, Proceedings of the Human Factors Society, 30th Annual Meeting, pp. 1212-1215, The Human Factors Society, Box 1369, Santa Monica, California 90406, 1986.

Schulmeyer, G.G. and J.I. McManus, *Handbook of Software Quality Assurance*, Van Nostrand Reinhold, New York, New York, 1992.

Schulmeyer, G., *Zero Defect Software*, McGraw-Hill, Inc., New York, New York, 1990.

Seamster, T.L., S.A. Fleger, and D.R. Eike, *The Prototype Process and User Interface Design*, Science Applications International Corporation, 1710 Goodridge Drive, McLean, Virginia 22102, 1987.

Sivertsen, T., and H. Valisuo, *Algebraic Specification and Theorem Proving Used in Formal Verification of Discrete-Event Control Systems*, Technical Report: OECD Halden Reactor Project, HWR-260, Institutt for Energiteknikk, Halden, Norway, December 1989.

Sizemore, N.L., *Test Techniques for Knowledge-Based Systems*, ITEA Journal, Vol. 11, No. 2, 1990.

Smith, S.L., and J.N. Mosier, *Guidelines for Designing User Interface Software*, ESD-TR-86-278, Electronic Systems Division, Air Force Systems Command, Hanscom Air Force Base, Massachusetts, United States Air Force, 1986.

Software A&E, *SNAP: Strategic Networked Applications Platform, Technical Brief*, Reston, Virginia, 1992.

Spivey, J., *The Z Notation -- A Reference Manual*, Prentice-Hall International, New York, New York 1986.

Stachowitz, R.A., and J.B. Combs, *Validation of Expert Systems*, Proceedings of the Hawaii International Conference on Systems Sciences, Kona, Hawaii, Lockheed Corporation, Palo Alto, California 94304, January 1987.

Stachowitz, R.A., C.L. Chang, T.S. Stock, and J.B. Combs, *Building Validation Tools for Knowledge-Based Systems*, First Annual Workshop on Space Operations Automation and Robotics (SOAR '87), NASA Johnson Space Center, Houston, Texas, August 1987.

Straker, E.A., and N.C. Thomas, *Verification and Validation as an Integral Part of the Development of Digital Systems for Nuclear Applications*, Nuclear Safety, Vol. 24, No. 3, pp. 338-351, May/June 1983.

Sudduth, A., *Diagnostic Reasoning Using Qualitative Causal Models*, Paper presented at the Electric Power Research Institute Conference and Expert System Applications for the Electric Power Industry, Boston, Electric Power Research Institute, Palo Alto, California, September 9-11, 1991)

Sun Microsystems Reference Manual, pp. 169-180, Revision A of March 27, 1990, pp. 169-180, Sun Microsystems, Inc., 2650 Park Tower Drive, Merrifield, Virginia 22116.

Swartout, W.R., *XPLAIN: A System for Creating and Explaining Expert Consulting programs*, Artificial Intelligence, 21(3), pp. 285-325, 1983.

Szygenda, S., D. Yatim, and J. Girardeau, *Fault Modeling for Digital Systems: A State of the Art Review*, Proceedings of Third International Conference on Software for Strategic Systems held February 27-28, 1990, Huntsville, Alabama, pp. 144-153, Addison-Wesley Publishing Company, Order Department, Jacob Way, Reading, Massachusetts 01867.

Teichroew, D. and E. Hershey, III, *PSL/PSA: A Computer-Aided Technique for Structure Documentation and Analysis of Information-Processing Systems*, IEEE Transactions on Software Engineering, SE(3)-1, pp. 41-48 ((PSL/PSA)), 1987.

Thomas, N.C., and E.A. Straker, *Application of Verification and Validation to Safety Parameter Display Systems*, Technical Report, Science Applications International Corporation, Lynchburg, Virginia, 1985.

Thomas, N.C., and C.L. Evans. *Life-cycle Verification, Validation and Testing*, Insights '86 Engineering & Operating Computer Forum, Edison Electric Institute, Washington, D.C., September 1986.

Tung, C., *On Control Flow Error Detection and Path Testing*, Proceedings of Third International Conference on Software for Strategic Systems, Huntsville, Alabama, February 27-28,1990, pp. 144-153, Addison-Wesley Publishing Company, Order Department, Jacob Way, Reading, Massachusetts 01867.

United Kingdom Ministry of Defense Draft Interim Defence Standard 00-55, *Requirements for the Procurement of Safety Critical Software in Defense Equipment*, National Institute of Standards and Technology Computer Systems Laboratory, Gaithersburg, Maryland 20899, May 9, 1989.

Von Mayrhauser, A., *Software Engineering: Methods and Management*, Academic Press, Inc., Boston, Massachusetts, 1990.

Wallace, D.R., and R.U. Fujii, *Software Verification and Validation: An Overview*, From IEEE Software, Vol. 6, No. 3, May 1989.

Wallace, R., J. Stockenberg, and R. Charette, *A Unified Methodology for Developing Systems*, McGraw-Hill, New York, New York, 1987.

Ward, P., *The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing*, IEEE Transactions on Software Engineering, 12(2), pp. 128-210, 1986.

Wasserman, A.I., *Extending State Transition Diagrams for the Specification of Human-Computer Interaction*, IEEE Transactions on Software Engineering, Vol SE-11 (8), pp. 699-713, 1985.

Weyuker, E., and T. Ostrand, *Theories of Program Testing and the Application of Revealing Subdomains*, IEEE Transactions on Software Engineering, Vol SE-6 (3), 1980.

Williges, R.C., B.H. Williges, and J. Elkerton, *Software Interface Design*, In G. Salvendy (Ed.) Handbook of Human Factors, John Wiley & Sons, New York, pp. 1416-1449, 1987.

Winchester, J., and G. Estin, *Requirements Definition and its Interface to the SARA Design Methodology for Computer-based Systems*, AFIPS Conference Proceedings, 51, pp. 369-379, ((RDL)), Addison-Wesley Publishing Company, Order Department, Jacob Way, Reading, Massachusetts 01867, 1982.

Wolfgram, D.D., T.J. Dear, and C.S. Galbraith, *Expert Systems for the Technical Professional*, John Wiley & Sons, New York, New York, 1987.

Wood, D.P., and W.G. Wood, *Comparative Evaluations of Four Specification Methods for Real-time Systems*, Carnegie-Mellon University/Software Engineering Institute Technical Report CMU/SEI 89-TR-36, Carnegie-Mellon University Library, Carnegie-Mellon University, EDSH 109, Pittsburgh, Pennsylvania 15213-3890, 1989.

Wood, W., R. Pethia, L. Gold, and R. Firth, *A Guide to the Assessment of Software Development Methods*, Carnegie Mellon University/Software Engineering Institute Technical Report CMU/SEI 88-TR-8, Carnegie-Mellon University Library, Carnegie-Mellon University, EDSH 109, Pittsburgh, Pennsylvania 15213-3890, April 1988.

Woods, D.D., *Cognitive Technologies: The Design of Joint Human-Machine Cognitive Systems*, The AI Magazine, 6(4), pp. 86-92, 1986.

NRC FORM 335 (2-89) NRCM 1102, 3201, 3202

U.S. NUCLEAR REGULATORY COMMISSION

BIBLIOGRAPHIC DATA SHEET

(See instructions on the reverse)

1. REPORT NUMBER (Assigned by NRC, Add Vol., Supp., Rev., and Addendum Numbers, If any.)

NUREG/CR-6316
SAIC-95/1028
Vol. 2

2. TITLE AND SUBTITLE

Guidelines for Verification and Validation of Expert Systems Software and Conventional Software

Survey and Assessment of Conventional Software Verification and Validation Methods

3. DATE REPORT PUBLISHED

MONTH	YEAR
March	1995

4. FIN OR GRANT NUMBER

L1530

5. AUTHOR(S)

L.A. Miller, E. H. Groundwater, J. E. Hayes, S. M. Mirsky

6. TYPE OF REPORT

7. PERIOD COVERED *(Inclusive Dates)*

8. PERFORMING ORGANIZATION – NAME AND ADDRESS *(If NRC, provide Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address; if contractor, provide name and mailing address.)*

Science Applications International Corporation
1710 Goodridge Drive
McLean, VA 221001

9. SPONSORING ORGANIZATION – NAME AND ADDRESS *(If NRC, type "Same as above"; if contractor, provide NRC Division, Office or Region, U.S. Nuclear Regulatory Commission, and mailing address.)*

Division of Systems Technology
Office of Nuclear Regulatory Research
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001

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

10. SUPPLEMENTARY NOTES

11. ABSTRACT *(200 words or less)*

By means of literature survey, a comprehensive set of methods was identified for the verification and validation of conventional software. The 153 methods so identified were classified according to their appropriateness of various phases of a development lifecycle -- requirements, design, and implementation; the last category was subdivided into two, static testing and dynamic testing methods. The methods were then characterized in terms of eight rating factors, four concerning ease-of-use of the methods and four concerning the methods; power to detect defects. Based on these and an Effectiveness Metric. The Effectiveness Metric was further refined to provide three different estimates for each method, depending on three classes of needed stringency of V&V (determined by ratings of a system's complexity and required integrity). Methods were then rank-ordered for each of the three classes in terms of their overall cost-benefits and effectiveness. The applicability was then assessed of each method for the four identified components of knowledge-based and expert systems, as well as the system as a whole.

12. KEY WORDS/DESCRIPTORS *(List words or phrases that will assist researchers in locating the report.)*

validation, verification, V&V expert systems, knowledge base, guidelines, scenarios, software quality assurance

13. AVAILABILITY STATEMENT

Unlimited

14. SECURITY CLASSIFICATION

(This Page)

Unclassified

(This Report)

Unclassified

15. NUMBER OF PAGES

16. PRICE