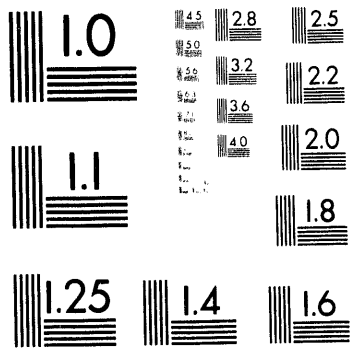

1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

1 of 2

NUREG/CR-6101
UCRL-ID-114839

Software Reliability and Safety in Nuclear Reactor Protection Systems

Manuscript Completed: June 1993
Date Published: November 1993

Prepared by
J. D. Lawrence

Lawrence Livermore National Laboratory
Livermore, CA 94550

Prepared for
Division of Reactor Controls and Human Factors
Office of Nuclear Reactor Regulation
U.S. Nuclear Regulatory Commission
Washington, DC 20555-0001
NRC FIN L1867

ABSTRACT

Planning the development, use and regulation of computer systems in nuclear reactor protection systems in such a way as to enhance reliability and safety is a complex issue. This report is one of a series of reports from the Computer Safety and Reliability Group, Lawrence Livermore National Laboratory, that investigates different aspects of computer software in reactor protection systems. There are two central themes in the report. First, software considerations cannot be fully understood in isolation from computer hardware and application considerations. Second, the process of engineering reliability and safety into a computer system requires activities to be carried out throughout the software life cycle. The report discusses the many activities that can be carried out during the software life cycle to improve the safety and reliability of the resulting product. The viewpoint is primarily that of the assessor, or auditor.

CONTENTS

1. Introduction 1
   1.1. Purpose 1
   1.2. Scope 1
   1.3. Report Organization 2
2. Terminology 3
   2.1. Systems Terminology 3
   2.2. Software Reliability and Safety Terminology 3
      2.2.1. Faults, Errors, and Failures 3
      2.2.2. Reliability and Safety Measures 4
      2.2.3. Safety Terminology 5
   2.3. Life Cycle Models 6
      2.3.1. Waterfall Model 7
      2.3.2. Phased Implementation Model 7
      2.3.3. Spiral Model 7
   2.4. Fault and Failure Classification Schemes 7
      2.4.1. Fault Classifications 12
      2.4.2. Failure Classifications 14
   2.5. Software Qualities 15
3. Life Cycle Software Reliability and Safety Activities 17
   3.1. Planning Activities 17
      3.1.1. Software Project Management Plan 19
      3.1.2 Software Quality Assurance Plan 21
      3.1.3. Software Configuration Management Plan 23
      3.1.4. Software Verification and Validation Plan 26
      3.1.5. Software Safety Plan 30
      3.1.6. Software Development Plan 33
      3.1.7. Software Integration Plan 35
      3.1.8. Software Installation Plan 36
      3.1.9. Software Maintenance Plan 37
      3.1.10. Software Training Plan 38
   3.2. Requirements Activities 38
      3.2.1. Software Requirements Specification 38
      3.2.2. Requirements Safety Analysis 43
   3.3. Design Activities 44
      3.3.1. Hardware and Software Architecture 45
      3.3.2. Software Design Specification 45
      3.3.3. Software Design Safety Analysis 47
   3.4. Implementation Activities 48
      3.4.1. Code Safety Analysis 48
   3.5. Integration Activities 49
      3.5.1. System Build Documents 49
      3.5.2. Integration Safety Analysis 49
   3.6. Validation Activities 49
      3.6.1. Validation Safety Analysis 50
   3.7. Installation Activities 50
      3.7.1. Operations Manual 50

3.7.2. Installation Configuration Tables 50
3.7.3. Training Manuals 50
3.7.4. Maintenance Manuals 50
3.7.5. Installation Safety Analysis 50
3.8. Operations and Maintenance Activities—Change Safety Analysis 51
4. Recommendations, Guidelines, and Assessment 53
4.1. Planning Activities 53
4.1.1. Software Project Management Plan 53
4.1.2. Software Quality Assurance Plan 54
4.1.3. Software Configuration Management Plan 56
4.1.4. Software Verification and Validation Plan 59
4.1.5. Software Safety Plan 65
4.1.6. Software Development Plan 67
4.1.7. Software Integration Plan 68
4.1.8. Software Installation Plan 69
4.1.9. Software Maintenance Plan 70
4.2. Requirements Activities 71
4.2.1. Software Requirements Specification 71
4.2.2. Requirements Safety Analysis 73
4.3. Design Activities 74
4.3.1. Hardware/Software Architecture Specification 74
4.3.2. Software Design Specification 74
4.3.3. Design Safety Analysis 75
4.4. Implementation Activities 76
4.4.1. Code Listings 76
4.4.2. Code Safety Analysis 77
4.5. Integration Activities 78
4.5.1. System Build Documents 78
4.5.2. Integration Safety Analysis 78
4.6. Validation Activities 78
4.6.1. Validation Safety Analysis 78
4.7. Installation Activities 79
4.7.1. Installation Safety Analysis 79
Appendix: Technical Background 81
A.1. Software Fault Tolerance Techniques 81
A.1.1. Fault Tolerance and Redundancy 82
A.1.2. General Aspects of Recovery 82
A.1.3. Software Fault Tolerance Techniques 84
A.2. Reliability and Safety Analysis and Modeling Techniques 87
A.2.1. Reliability Block Diagrams 87
A.2.2. Fault Tree Analysis 88
A.2.3. Event Tree Analysis 93
A.2.4. Failure Modes and Effects Analysis 93
A.2.5. Markov Models 95
A.2.6. Petri Net Models 97
A.3. Reliability Growth Models 101
A.3.1. Duane Model 103
A.3.2. Musa Model 103
A.3.3. Littlewood Model 104
A.3.4. Musa-Okumoto Model 104
References 107
Standards 107
Books, Articles, and Reports 108
Bibliography 113

Figures

Figure 2-1. Documents Produced During Each Life Cycle Stage 8
Figure 2-2. Waterfall Life Cycle Model 10
Figure 2-3. Spiral Life Cycle Model 11
Figure 3-1. Software Planning Activities 18
Figure 3-2. Outline of a Software Project Management Plan 19
Figure 3-3. Outline of a Software Quality Assurance Plan 22
Figure 3-4. Outline of a Software Configuration Management Plan 24
Figure 3-5. Verification and Validation Activities 28
Figure 3-6. Outline of a Software Verification and Validation Plan 30
Figure 3-7. Outline of a Software Safety Plan 30
Figure 3-8. Outline of a Software Development Plan 34
Figure 3-9. Outline of a Software Integration Plan 35
Figure 3-10. Outline of a Software Installation Plan 37
Figure 3-11. Outline of a Software Maintenance Plan 37
Figure 3-12. Outline of a Software Requirements Plan 39
Figure A-1. Reliability Block Diagram of a Simple System 89
Figure A-2. Reliability Block Diagram of Single, Duplex, and Triplex Communication Line 89
Figure A-3. Reliability Block Diagram of Simple System with Duplexed Communication Line 90
Figure A-4. Reliability Block Diagram that Cannot Be Constructed from Serial and Parallel Parts 90
Figure A-5. Simple Fault Tree 90
Figure A-6. AND Node Evaluation in a Fault Tree 91
Figure A-7. OR Node Evaluation in a Fault Tree 91
Figure A-8. Example of a Software Fault Tree 92
Figure A-9. Simple Event Tree 93
Figure A-10. A Simple Markov Model of a System with Three CPUs 95
Figure A-11. Markov Model of a System with CPUs and Memories 96
Figure A-12. Simple Markov Model with Varying Failure Rates 97
Figure A-13. Markov Model of a Simple System with Transient Faults 97
Figure A-14. An Unmarked Petri Net 99
Figure A-15. Example of a Marked Petri Net 99
Figure A-16. The Result of Firing Figure A-15 100
Figure A-17. A Petri Net for the Mutual Exclusion Problem 100
Figure A-18. Petri Net for a Railroad Crossing 101
Figure A-19. Execution Time Between Successive Failures of an Actual System 102

Tables

Table 2-1. Persistence Classes and Fault Sources 12
Table A-1. Failure Rate Calculation 104

ABBREVIATIONS AND ACRONYMS

ANSI	American National Standards Institute
CASE	Computer-Assisted Software Engineering
CCB	Configuration Control Board
CI	Configuration Item
CM	Configuration Management
CPU	Central Processing Unit
ETA	Event Tree Analysis
FBD	Functional Block Diagram
FLBS	Functional Level Breakdown Structure
FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes, Effects and Criticality Analysis
FTA	Fault Tree Analysis
I&C	Instrumentation and Control
I/O	Input/Output
IEEE	Institute of Electrical and Electronic Engineers
MTTF	Mean Time To Failure
PDP	Previously Developed or Purchased
PERT	Program Education and Review Technique
QA	Quality Assurance
RAM	Random Access Memory
ROM	Read Only Memory
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SPMP	Software Project Management Plan
SQA	Software Quality Assurance
SQAP	Software Quality Assurance Plan
SRS	Software Requirements Specification
SSP	Software Safety Plan
TMR	Triple Modular Redundancy
UCLA	University of California at Los Angeles
UPS	Uninterruptable Power Supply
V&V	Verification and Validation
WBS	Work Breakdown Structure

EXECUTIVE SUMMARY

The development, use, and regulation of computer systems in nuclear reactor protection systems to enhance reliability and safety is a complex issue. This report is one of a series of reports from the Computer Safety and Reliability Group, Lawrence Livermore National Laboratory, which investigates different aspects of computer software in reactor protection systems.

There are two central themes in this report. First, software considerations cannot be fully understood in isolation from computer hardware and application considerations. Second, the process of engineering reliability and safety into a computer system requires activities to be carried out throughout the software life cycle. These two themes affect both the structure and the content of this report.

Reliability and safety are concerned with faults, errors, and failures. A *fault* is a triggering event that causes things to go wrong; a software bug is an example. The fault may cause a change of state in the computer, which is termed an *error*. The error remains *latent* until the incorrect state is used; it then is termed *effective*. It may then cause an externally-visible *failure*. Only the failure is visible outside the computer system. Preventing or correcting the failure can be done at any of the levels: preventing or correcting the causative fault, preventing the fault from causing an error, preventing the error from causing a failure, or preventing the failure from causing damage. The techniques for achieving these goals are termed *fault prevention*, *fault correction*, and *fault tolerance*.

Reliability and *safety* are related, but not identical, concepts. Reliability, as defined in this report, is a measure of how long a system will run without failure of any kind, while safety is a measure of how long a system will run without *catastrophic* failure. Thus safety is directly concerned with the consequences of failure, not merely the existence of failure. As a result, safety is a system issue, not simply a software issue, and must be analyzed and discussed as a property of the entire reactor protection system.

Faults and failures can be classified in several different ways. Faults can be described as *design faults*, *operational faults*, or *transient faults*. All software faults are design faults; however, hardware faults may occur in any of the three classes. This is important in a safety-related system since the software may be required to compensate for the operational faults of the hardware. Faults can also be classified by the source of the fault; software and hardware are two of the possible sources discussed in the report. Others are: input data, system state, system topology, people, environment, and unknown. For example, the source of many transient faults is unknown.

Failures are classified by mode and scope. A failure mode may be sudden or gradual; partial or complete. All four combinations of these are possible. The scope of a failure describes the extent within the system of the effects of the failure. This may range from an internal failure, whose effect is confined to a single small portion of the system, to a pervasive failure, which affects much of the system.

Many different life cycle models exist for developing software systems. These differ in the timing of the various activities that must be done in order to produce a high-quality software product, but the actual activities must be done in any case. No particular life cycle is recommended here, but there are extensive comments on the activities that must be carried out. These have been divided into eight categories, termed *sets of activities* in the report. These sets are used merely to group related activities; there is no implication that the activities in any one set must be all carried out at the same time, or that activities in "later" sets must follow those of "earlier" sets. The eight categories are as follows:

- Planning activities result in the creation of a number of documents that are used to control the development process. Eleven are recommended here: a Software Project Management Plan, a Software Quality Assurance Plan, a Software Configuration Management (CM) Plan, a Software Verification and Validation (V&V) Plan, a Software Safety Plan, a Software Development Plan, a Software Integration Plan, a Software Installation Plan, a Software Maintenance Plan, a Software Training Plan, and a Software Operations Plan. Many of these plans are discussed in detail, relying on various ANSI/IEEE standards when these exist for the individual plans.
- The second set of activities relate to documenting the *requirements* for the software system. Four documents are recommended: the Software Requirements Specification, a Requirements Safety Analysis, a V&V

Requirements Analysis, and a CM Requirements Report. These documents will fully capture all the requirements of the software project, and relate these requirements to the overall protection system functional requirements and protection system safety requirements.

- The *design* activities include five recommended documents. The Hardware and Software Architecture will describe the computer system design at a fairly high level, giving hardware devices and mapping software activities to those devices. The Software Design Specification provides the complete design on the software products. Design analyses include the Design Safety Analysis, the V&V Design Analysis, and the CM Design Report.
- *Implementation* activities include writing and analyzing the actual code, using some programming language. Documents include the actual code listings, the Code Safety Analysis, the V&V Implementation Analysis and Test Report, and the CM Implementation Report.
- *Integration* activities are those activities that bring software, hardware, and instrumentation together to form a complete computer system. Documents include the System Build Documents, the Integration Safety Analysis, the V&V Integration Analysis and Test Report, and the CM Integration Report.
- *Validation* is the process of ensuring that the final complete computer system achieves the original goals that were imposed by the protection system design. The final system is matched against the original requirements, and the protection system safety analysis. Documents include the Validation Safety Analysis, the V&V Validation and Test Report, and the CM Validation Report.
- *Installation* is the process of moving the completed computer system from the developer's site to the operational environment, within the actual reactor protection system. The completion of installation provides the operator with a documented operational computer system. Seven documents are recommended: the Operations Manual, the Installation Configuration Tables, Training Manuals, Maintenance Manuals, an Installation Safety Analysis, a V&V Installation Analysis and Test Report, and a CM Installation Report.
- The *operations and maintenance* activities involve the actual use of the computer system in the operating reactor, and making any required changes to it. Changes may be required due to errors in the system that were not found during the development process, changes to hardware or requirements for additional functionality. Safety analyses, V&V analyses, and CM activities are all recommended as part of the maintenance process.

Three general methods exist that may be used to achieve software fault tolerance; n-version programming, recovery block, and exception handling. Each of these attempts to achieve fault tolerance by using more than one algorithm or program module to perform a calculation, with some means of selecting the preferred result. In *n-version programming*, three or more program modules that implement the same function are executed in parallel, and voting is used to select the "correct" one. In *recovery block*, two or more modules are executed in series, with an acceptance algorithm used after each module is executed to decide if the result should be accepted or the next module executed. In *exception handling*, a single module is executed, with corrections made when exceptions are detected. Serious questions exist as to the applicability of the n-version programming and the recovery-block techniques to reactor protection systems, because of the assumptions underlying the techniques, the possibility of common-mode failures in the voting or decision programs, and the cost and time of implementing them.

One means of assessing system reliability or safety is to create a mathematical model of the system and analyze the properties of that model. This can be very effective providing that the model captures all the relevant factors of the reality. Reliability models have been used for many years for electronic and mechanical systems. The use of reliability models for software is fairly new, and their effectiveness has not yet been fully demonstrated. Fault tree models, event tree models, failure modes and effects analysis, Markov models, and Petri net models all have possibilities. Of particular interest are reliability growth models, since software bugs tend to be corrected as they are found. Reliability Growth models can be very useful in understanding the growth of reliability through a testing activity, but cannot be used alone to justify software for use in a safety-related application, since such applications require a much higher level of reliability than can be convincingly demonstrated during a test-correct-test activity.

Software Reliability and Safety in Nuclear Reactor Protection Systems

1. INTRODUCTION

1.1. Purpose

Reliability and safety are related, but not identical, concepts. Reliability can be thought of as the probability that a system fails in any way whatever, while safety is concerned with the consequences of failure. Both are important in reactor protection systems. When a protection system is controlled by a computer, the impact of the computer system on reliability and safety must be considered in the reactor design. Because software is an integral part of a computer system, software reliability and software safety become a matter of concern to the organizations that develop software for protection systems and to the government agencies that regulate the developers. This report is oriented toward the assessment process. The viewpoint is from that of a person who is assessing the reliability and safety of a computer software system that is intended to be used in a reactor protection system.

1.2. Scope

Software is only one portion of a computer system. The other portions are the computer hardware and the instrumentation (sensors and actuators) to which the computer is connected. The combination of software, hardware, and instrumentation is frequently referred to as the Instrumentation and Control (I&C) System. Nuclear reactors have at least two I&C systems—one controls the reactor operation, and the other controls the reactor protection. The latter, termed the Protection Computer System, is the subject of this report.

This report assumes that the computer system as a whole, as well as the hardware and instrumentation subsystems, will be subject to careful development, analysis, and assessment in a manner similar to that given here for the software. That is, it is assumed that there will be appropriate plans, requirements and design specifications, procurement and installation, testing and analysis for the complete computer system, as well as the hardware, software, and instrumentation subsystems. The complete computer system and the hardware and instrumentation subsystems are discussed here only as they relate to the software subsystem.

The report is specifically directed toward enhancing the reliability and safety of computer controlled reactor protection systems. Almost anything can affect safety, so it is difficult to bound the contents of the report. Consequently material is included that may seem tangential to the topic. In these cases the focus is on reliability and safety; other aspects of such material are summarized or ignored. More complete discussions of these secondary issues may be found in the references.

This report is one of a series of reports prepared by the Computer Safety and Reliability Group, Fission Energy and System Safety Program, Lawrence Livermore National Laboratory. Aspects of software reliability and safety engineering that are covered in the other reports are treated briefly in this report, if at all. The reader is referred to the following additional reports:

1. Robert Barter and Lin Zucconi, "Verification and Validation Techniques and Auditing Criteria for Critical System-Control Software," Lawrence Livermore National Laboratory, Livermore, CA (February 1993).
2. George G. Preckshot, "Real-Time Systems Complexity and Scalability," Lawrence Livermore National Laboratory, Livermore, CA (August 1992).

3. George G. Preckshot and Robert H. Wyman, "Communications Systems in Nuclear Power Plants," Lawrence Livermore National Laboratory, Livermore, CA (August 1992).

4. George G. Preckshot, "Real-Time Performance," Lawrence Livermore National Laboratory, Livermore, CA (November 1992).

5. Debra Sparkman, "Techniques, Processes, and Measures for Software Safety and Reliability," Lawrence Livermore National Laboratory, Livermore, CA (April 1992).

6. Lloyd G. Williams, "Formal Methods in the Development of Safety Critical Software Systems," SERM-014-91, Software Engineering Research, Boulder, CO (April 1992).

7. Lloyd G. Williams, "Assessment of Formal Specifications for Safety-Critical Systems," Software Engineering Research, Boulder, CO (February 1993).

8. Lloyd G. Williams, "Considerations for the Use of Formal Methods in Software-Based Safety Systems," Software Engineering Research, Boulder, CO (February 1993).

9. Lin Zucconi and Booker Thomas, "Testing Existing Software for Safety-Related Applications," Lawrence Livermore National Laboratory, Livermore, CA (January 1993).

1.3. Report Organization

Section 2 contains background on several topics relating to software reliability and software safety. Terms are defined, life cycle models are discussed briefly, and two classification schemes are presented.

Section 3 provides detail on the many life cycle activities that can be done to improve reliability and safety. Development activities are divided into eight sets of activities: planning, requirements specification, design specification, software implementation, integration with hardware and instrumentation, validation, installation and operations, and maintenance. Each set of activities includes a number of tasks that can be undertaken to enhance reliability and safety. Because the report is oriented towards assessment, the tasks are discussed in terms of the documents they produce and the actions necessary to create the document contents.

Section 4 discusses specific motivations, recommendations, guidelines, and assessment questions. The motivation sections describe particular concerns of the assessor when examining the safety of software in a reactor protection system. Recommendations consist of actions the developer should or should not do in order to address such concerns. Guidelines consist of suggestions that are considered good engineering practice when developing software. Finally, the assessment sections consist of lists of questions that the assessor may use to guide the assessment of a particular aspect of the software system.

From the viewpoint of the assessor, software development consists of the organization that does the development, the process used in the development, and the products of that development. Each is subject to analysis, assessment and judgment. This report discusses all three aspects in various places within the framework of the life cycle. Process and product are the primary emphasis.

Following the main body of the report, the appendix provides information on software fault tolerance techniques and software reliability models. A bibliography of information relating to software reliability and safety is also included.

2. TERMINOLOGY

This section includes discussions of the basic terminology used in the remainder of the report. The section begins with a description of the terms used to describe systems. Section 2.2 provides careful definitions of the basic terminology for reliability and safety. Section 2.3 contains brief descriptions of several of the life cycle models commonly used in software development, and defines the various activities that must be carried out during any software development project. Section 2.4 describes various classification schemes for failures and faults, and provides the terms used in these schemes. Finally, Section 2.5 discusses the terms used to describe software qualities that are used in following sections.

2.1. Systems Terminology

The word *system* is used in many different ways in computer science. The basic definition, given in IEEE Standard 610.12, is "a collection of components organized to accomplish a specific function or set of functions." In the context of a nuclear reactor, the word could mean, depending on context, the society using the reactor, the entire reactor itself, the portion devoted to protection, the computer hardware and software responsible for protection, or just the software.

In this report the term *system*, without modifiers, will consistently refer to the complete application with which the computer is directly concerned. Thus a "system" should generally be understood as a "reactor protection system." When portions of the protection system are meant, and the meaning isn't clear from context, a modifier will be used. Reference could be made to the computer system (a portion of the protection system), the software system (in the computer system), the hardware system (in the computer system) and so forth. In some cases, the term "application system" is used to emphasize that the entire reactor protection system is meant.

A computer system is itself composed of subsystems. These include the computer hardware, the computer software, operators who are using the computer system, and the instruments to which the computer is connected. The definition of *instrument* is taken from ANSI/ISA Standard S5.1: "a device used directly or indirectly to measure and/or control a variable. The term includes primary elements, final control elements, computing devices and electrical devices such as annunciators, switches, and pushbuttons. The term does not apply to parts that are internal components of an instrument."

Since this report is concerned with computer systems in general, and software systems in particular, instruments are restricted to those that interact with the computer system. There are two types: sensors and actuators. Sensors provide information to the software on the state of the reactor, and actuators provide commands to the rest of the reactor protection system from the software.

2.2. Software Reliability and Safety Terminology

2.2.1. Faults, Errors, and Failures

The words *fault*, *error*, and *failure* have a plethora of definitions in the literature. This report uses the following definitions, specialized to computer systems (Laprie 1985; Randell 1978; Siewiorek 1982).

A *fault* is a deviation of the behavior of a computer system from the authoritative specification of its behavior. A *hardware fault* is a physical change in hardware that causes the computer system to change its behavior in an undesirable way. A *software fault* is a mistake (also called a bug) in the code. A *user fault* consists of a mistake by a person in carrying out some procedure. An *environmental fault* is a deviation from expected behavior of the world outside the computer system; electric power interruption is an example. The classification of faults is discussed further in Subsection 2.4.1.

An *error* is an incorrect state of hardware, software, or data resulting from a fault. An error is, therefore, that part of the computer system state that is liable to lead to failure. Upon occurrence, a fault creates a *latent error*, which becomes *effective* when it is activated, leading to a failure. If never activated, the latent error never becomes effective and no failure occurs.

A *failure* is the external manifestation of an error. That is, a failure is the external effect of the error, as seen by a (human or physical device) user, or by another program.

Some examples may clarify the differences among the three terms. A *fault* may occur in a circuit (a wire breaks) causing a bit in memory to always be a 1 (an

error, since memory is part of the state) resulting in a *failed* calculation.

A programmer's mistake is a *fault;* the consequence is a *latent error* in the written software (erroneous instruction). Upon *activation* of the module where the error resides, the error becomes *effective*. If this effective error causes a divide by zero, a *failure* occurs and the program aborts.

A maintenance or operating manual writer's mistake is a *fault*; the consequence is an *error* in the corresponding manual, which will remain latent as long as the directives are not acted upon.

The view summarized here enables fault pathology to be made precise. The creation and action mechanisms of faults, errors and failures may be summarized as follows.

1. A *fault* creates one or more latent errors in the computer system component where it occurs. Physical faults can directly affect only the physical layer components, whereas other types of faults may affect any component.
2. There is always a time delay between the occurrence of a fault and the occurrence of the resulting latent error(s). This may be measured in nanoseconds or years, depending on the situation. Some faults may not cause errors at all; for example, a bug in a portion of a program that is never executed. It is convenient to consider this to be an extreme case in which an infinite amount of time elapses between fault and latent error.
3. The properties governing *errors* may be stated as follows:
   a. A latent error becomes effective once it is activated.
   b. An error may cycle between its latent and effective states.
   c. An effective error may, and in general does, propagate from one component to another. By propagating, an error creates other (new) errors.

From these properties it may be deduced that an effective error within a component may originate from:

- Activation of a latent error within the same component.
- An effective error propagating within the same component or from another component.

4. A component *failure* occurs when an error affects the service delivered (as a response to requests) by the component. There is always a time delay between the occurrence of the error and the occurrence of the resulting failure. This may vary from nanoseconds to infinity (if the failure never actually occurs).
5. These properties apply to any component of the computer system. In a hierarchical system, failures at one level can usefully be thought of as faults by the next higher level.

Most reliability, availability, and safety analysis and modeling assume that each fault causes at most a single failure. That is, failures are statistically independent. This is not always true. A *common-mode failure* occurs when multiple components of a computer system fail due to a single fault. If common mode failures do occur, an analysis that assumes that they do not will be excessively optimistic. There are a number of reasons for common mode failures (Dhillon 1983):

- Environmental causes, such as dirt, temperature, moisture, and vibrations.
- Equipment failure that results from an unexpected external event, such as fire, flood, earthquake, or tornadoes.
- Design deficiencies, where some failures were not anticipated during design. An example is multiple telephone circuits routed through a single equipment box. Software design errors, where identical software is being run on multiple computers, is of particular concern in this report.
- Operational errors, due to factors such as improper maintenance procedures, carelessness, or improper calibration of equipment.
- Multiple items purchased from the same vendor, where all of the items have the same manufacturing defect.
- Common power supply used for redundant units.
- Functional deficiencies, such as misunderstanding of process variable behavior, inadequately designed protective actions, or inappropriate instrumentation.

2.2.2. Reliability and Safety Measures

Reliability and safety measurements are inherently statistical, so the fundamental quantities are defined

statistically. The four basic terms are reliability, availability, maintainability, and safety. These and other related terms are defined in the following text. Note that the final three definitions are qualitative, not quantitative (Siewiorek 1982; Smith 1972). Most of these definitions apply to arbitrary systems. The exception is safety; since this concept is concerned with the consequences of failure, rather than the simple fact of failure, the definition applies only to a system that can have major impacts on people or equipment. More specifically, safety applies to reactors, not to components of a reactor.

- The *reliability*, $R(t)$, of a system is the conditional probability that the system has survived the interval [0, t], given that it was operating at time 0. Reliability is often given in terms of the *failure rate* (also referred to as the *hazard rate*), $\lambda(t)$, or the *mean time to failure*, $mttf$. If the failure rate is constant, $mttf = 1/\lambda$. Reliability is a measure of the success with which the system conforms to some authoritative specification of its behavior, and cannot be measured without such a specification.
- The *availability*, $A(t)$, of a system is the probability that the system is operational at the instant of time t. For nonrepairable systems, availability and reliability are equal. For repairable systems, they are not. As a general rule, $0 \le R(t) \le A(t) \le 1$.
- The *maintainability*, $M(t)$, of a system is the conditional probability that the system will be restored to operational effectiveness by time t, given that it was not functioning at time 0. Maintainability is often given in terms of the repair rate, $\mu(t)$, or *mean time to repair*, $mttr$. If the repair rate is constant, $mttr = 1/\mu$.
- The *safety*, $S(t)$, of a system is the conditional probability that the system has survived the interval [0, t] without an accident, given that it was operating without catastrophic failure at time 0.
- The *dependability* of a system is a measure of its ability to commence and complete a mission without failure. It is therefore a function of both reliability and maintainability. It can be thought of as the quality of the system that permits the user to rely on it for service.
- The *capability* of a system is a measure of its ability to satisfy the user's requirements.
- *System effectiveness* is the product of capability, availability and dependability. *System cost effectiveness* is the quotient of system effectiveness and cost.

2.2.3. Safety Terminology

Safety engineering has special terminology of its own. The following definitions, based on those developed by the IEEE Draft Standard 1228, are used in this report. They are reasonably standard definitions, but specialized to computer software in a few places.

- An *accident* is an unplanned event or series of events that result in death, injury, illness, environmental damage, or damage to or loss of equipment or property. (The word *mishap* is sometimes used to mean an accident, financial loss or public relations loss.)
- A *system hazard* is an application system condition that is a prerequisite to an accident. That is, the system states can be divided into two sets. No state in the first set (of *nonhazardous states*) can directly cause an accident, while accidents may result from any state in the second set (of *hazardous states*). Note that a system can be in a hazardous state without an accident occurring—it is the potential for causing an accident that creates the hazard, not necessarily the actuality.
- The term *risk* is used to designate a measure that combines the likelihood that a system hazard will occur, the likelihood that the hazard will cause an accident and the severity of the worst plausible accident. The simplest measure is to simply multiply the probability that a hazard occurs, the probability that a hazard will cause an accident (given that the hazard occurs), and the worst-case severity of the accident.
- *Safety-critical software* is software whose inadvertent response to stimuli, failure to respond when required, response out-of-sequence, or response in unplanned combination with others can result in an accident. This includes software whose operation or failure to operate can lead to a hazardous state, software intended to recover from hazardous states, and software intended to mitigate the severity of, or recover from, an accident.
- The term *safety* is used to mean the extent to which a system is free from system hazard. This is a less precise definition than that given in Section 2.2.2, which is generally preferred in this report.

It is also useful to consider the word "critical" when used to describe systems. A *critical system* is a system whose failure may have very unpleasant consequences (mishaps). The results of failure may affect the developers of the system, its direct users, their customers or the general public. The consequences may involve loss of life or property, financial loss, legal liability (such as jail), regulatory threats, or even the loss of good will (if that is extremely important). The term *safety critical* refers to a system whose failure could cause an accident.

A good brief discussion of accidents is found in Leveson 1991:

> *Despite the usual oversimplification of the causes of particular accidents ("human error" is often the identified culprit despite the all-encompassing nature and relative uselessness of such a categorization), accidents are caused almost without exception by multiple factors, and the relative contribution of each is usually not clear. An accident may be thought of as a set of events combining together in random fashion or, alternatively, as a dynamic mechanism that begins with the activation of a hazard and flows through the system as a series of sequential and concurrent events in a logical sequence until the system is out of control and a loss is produced (the "domino theory"). Either way, major incidents often have more than one single cause, and it is usually difficult to place blame on any one event or component of the system. The high frequency of complex, multifactorial accidents may arise from the fact that the simpler potentials have been anticipated and handled. But the very complexity of events leading to an accident implies that there may be many opportunities to intervene or interrupt the sequence.*
>
> *A second characteristic of accidents is that they often involve problems in subsystem interfaces. It appears to be easier to deal with failures of components than failures in the interfaces between components. This should not be a surprise to software engineers, consider the large number of operational software faults that can be traced back to requirements problems. The software requirements are the specific representation of the interface between the software and the processes or devices being controlled.*
>
> *A third important characteristic claimed for accidents is that they are intimately intertwined with complexity and coupling. Perrow has argued that accidents are "normal" in complex and tightly coupled systems. Unless great care is taken, the addition of computers to control these systems is likely to increase both complexity and coupling, which will increase the potential for accidents.*

2.3. Life Cycle Models

Many different software life cycles have been proposed. These have different motivations, strengths, and weaknesses. The life cycle models generally require the same types of tasks to be carried out; they differ in the ordering of these tasks in time. No particular life cycle is assumed here. There is an assumption that the activities that occur during the developer's life cycle yield the products indicated in Figure 2-1. Each of the life cycle activities produces one or more products, mostly documents, that can be assessed. The development process itself is subject to assessment.

The ultimate result of software development, as considered in this report, is a suite of computer programs that run on computers and control the reactor protection system. These programs will have characteristics deemed desirable by the developer or customer, such as reliability, performance, usability, and functionality. This report is only concerned with reliability and safety; however, that concern does "spill over" into other qualities.

The development model used here suggests one or more audits of the products of each set of life cycle activities. The number of audits depends, among other things, on the specific life cycle model used by the developer. The audit will assess the work done that relates to the set of activities being audited. Many reliability, performance, and safety problems can be resolved only by careful design of the software product, so must be addressed early in the life cycle, no matter which life cycle is used. Any errors or oversights can require difficult and expensive retrofits, so are best found as early as possible. Consequently, an incremental audit process is believed to be more cost effective than a single audit at the end of the

development process. In this way, problems can be detected early in the life cycle and corrected before large amounts of resources have been wasted.

Three of the many life cycle models are described briefly in subsections 2.3.1. through 2.3.3. No particular life cycle model is advocated. Instead, a model should be chosen to fit the style of the development organization and the nature of the problem being solved.

2.3.1. Waterfall Model

The classic waterfall model of software development assumes that each phase of the life cycle can be completed before the next phase is begun (Pressman 1987). This is illustrated in Figure 2-2. The actual phases of the waterfall model differ among the various authors who discuss the model; the figure shows phases appropriate to reactor protection systems. Note that the model permits the developer to return to previous phases. However, this is considered to be an exceptional condition to the normal forward flow, included to permit errors in previous stages to be corrected. For example, if a requirements error is discovered during the implementation phase, the developer is expected to halt work, return to the requirements phase, fix the problem, change the design accordingly, and then restart the implementation from the revised design. In practice, one only stops the implementation affected by the newly discovered requirement.

The waterfall model has been severely criticized as not being realistic to many software development situations, and this is frequently justified. It remains an excellent model for those situations where the requirements are known and stable before development begins, and where little change to requirements is anticipated.

2.3.2. Phased Implementation Model

This model assumes that the development will take place as a sequence of versions, with a release after each version is completed. Each version has its own life cycle model. If new requirements are generated during the development of a version, they will generally be delayed until the next version, so a waterfall model may be appropriate to each version. (Marketing pressures may modify such delays.)

This model is appropriate to commercial products that are evolving over long periods of time, or for which external requirements change slowly. Operating systems and language compilers are examples.

2.3.3. Spiral Model

The spiral model was developed at TRW (Boehm 1988) in an attempt to solve some of the perceived difficulties with earlier models. This model assumes that software development can be modeled as a sequence of activities, as shown in Figure 2-3. Each time around the spiral (phase), the product is developed to a more complete degree. Four broad steps are required:

1. Determine the objectives for the phase. Consider alternatives to meeting the objectives.
2. Evaluate the alternatives. Identify risks to completing the phase, and perform a risk analysis. Make a decision to proceed or stop.
3. Develop the product for the particular phase.
4. Plan for the next phase.

The products for each phase may match those of the previous models. In such circumstances, the first loop around the spiral results in a concept of operations; the next, a requirements specification; the next, a design; and so forth. Alternately, each loop may contain a complete development cycle for one phase of the product; here, the spiral model looks somewhat like the phased implementation model. Other possibilities exist.

The spiral model is particularly appropriate when considerable financial, schedule, or technical risk is involved in the product development. This is because an explicit risk analysis is carried out as part of each phase, with an explicit decision to continue or stop.

2.4. Fault and Failure Classification Schemes

Faults and failures can be classified in several different ways. Those that are considered useful in safety-related applications are described briefly here. Faults are classified by persistence and by the source of the fault. There is some interaction between these, in the sense that not all persistence classes may occur for all sources. Table 2-1 provides the interrelationship.

Failures are classified by mode, scope, and the effect on safety. These classification schemes consider the effect of a failure, both on the environment within which the computer system operates, and on the components of the system.

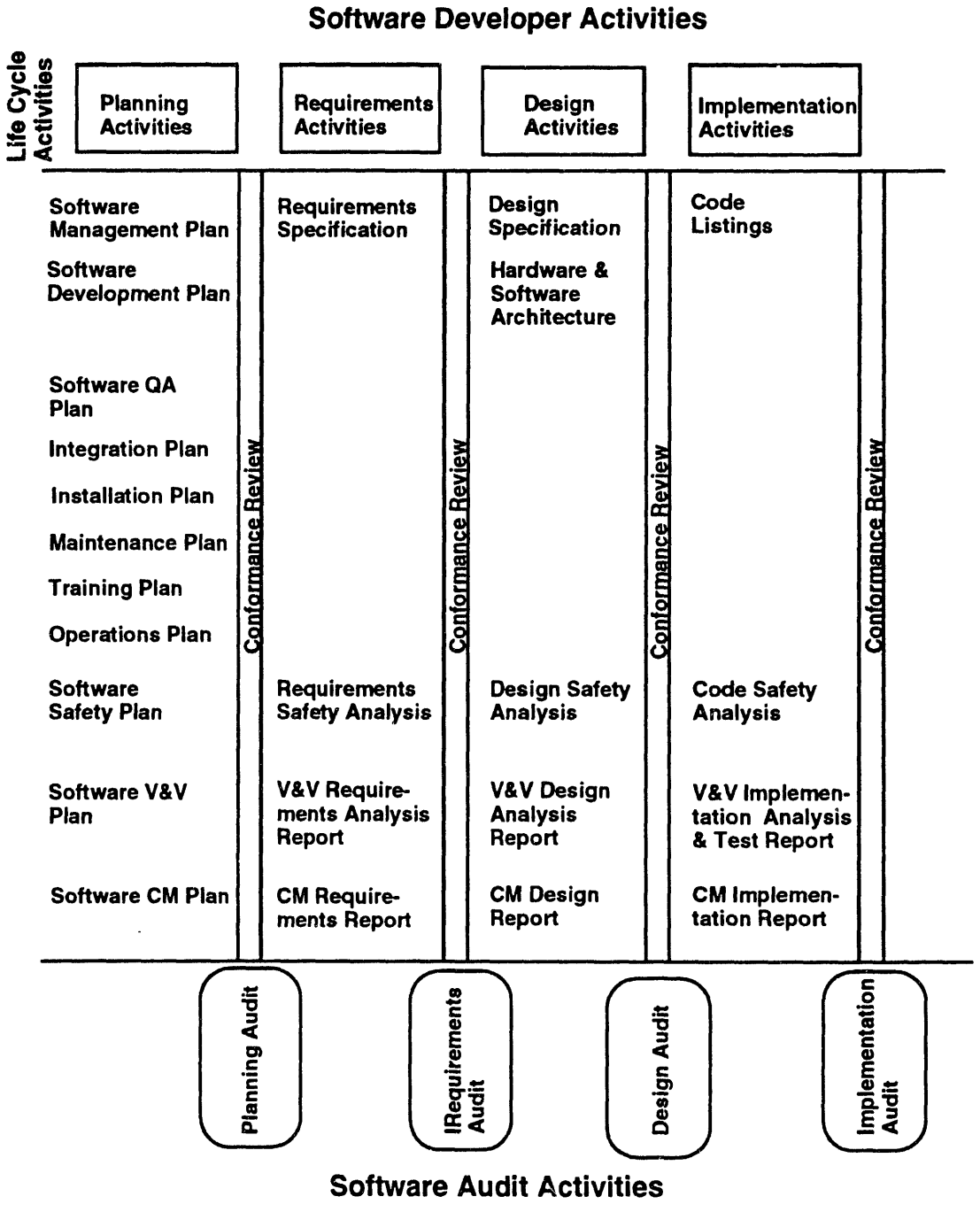


Figure 2-1. Documents Produced During Each Life Cycle Stage

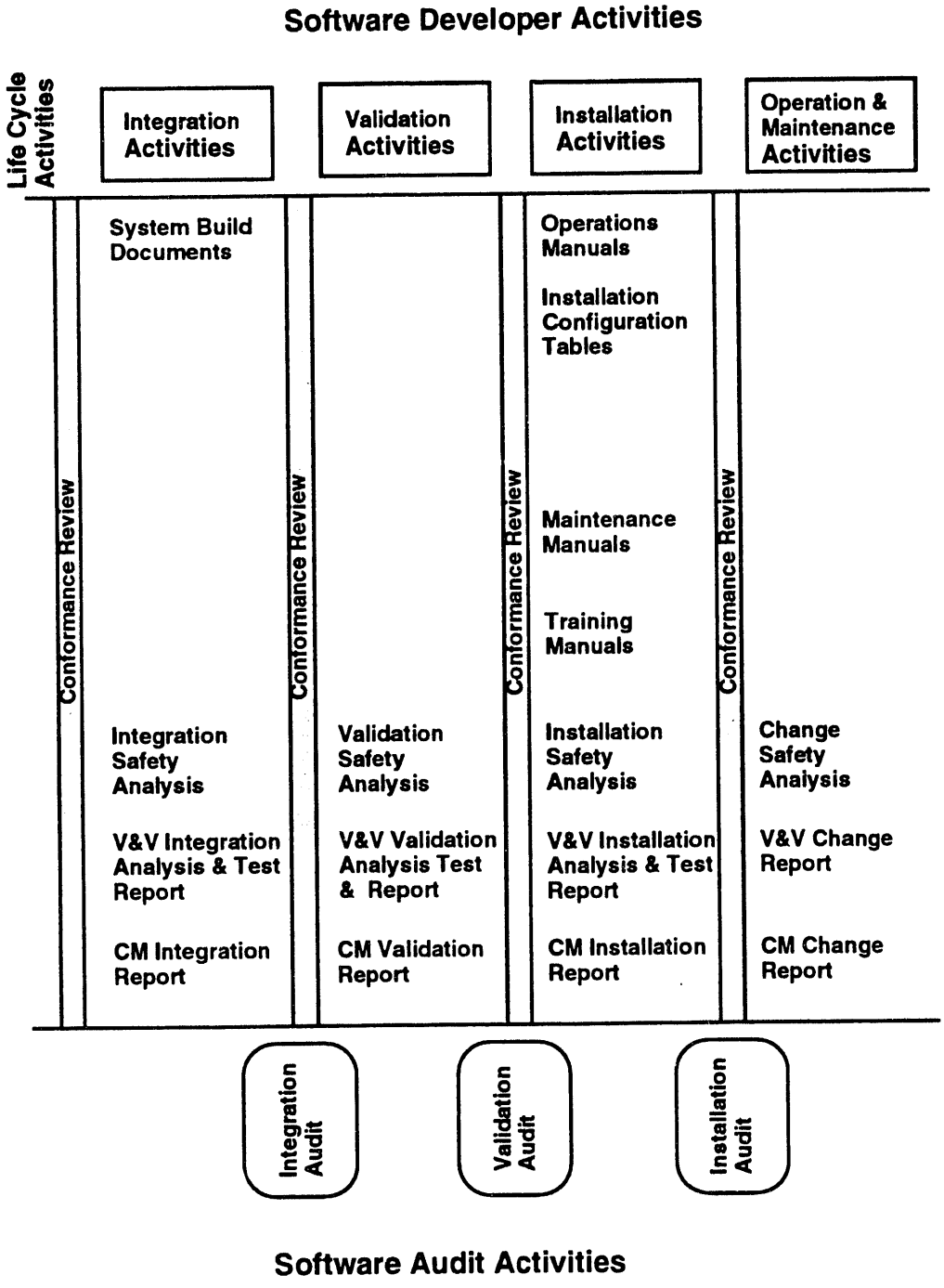


Figure 2-1. Documents Produced During Each Life Cycle Stage (continued)

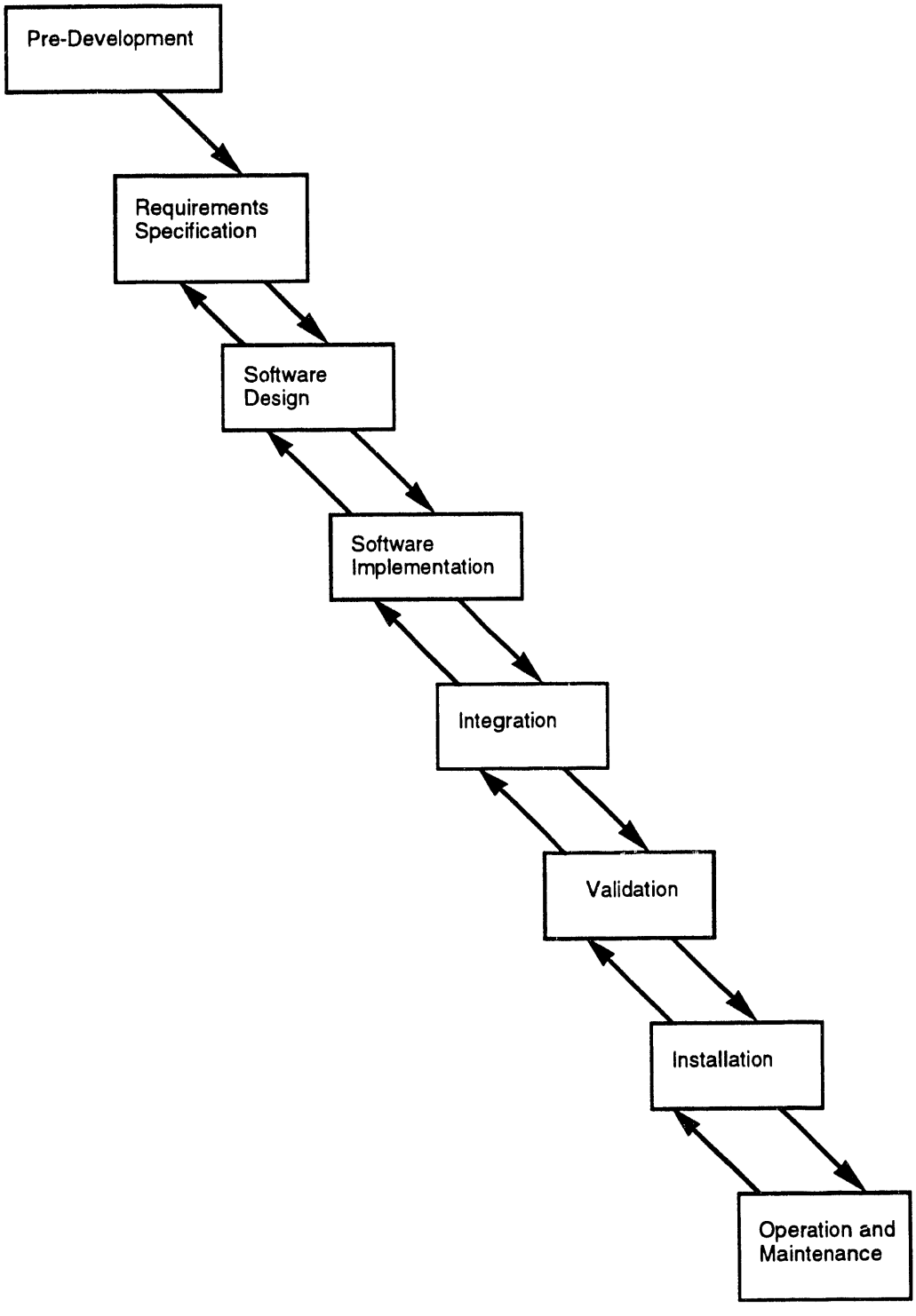
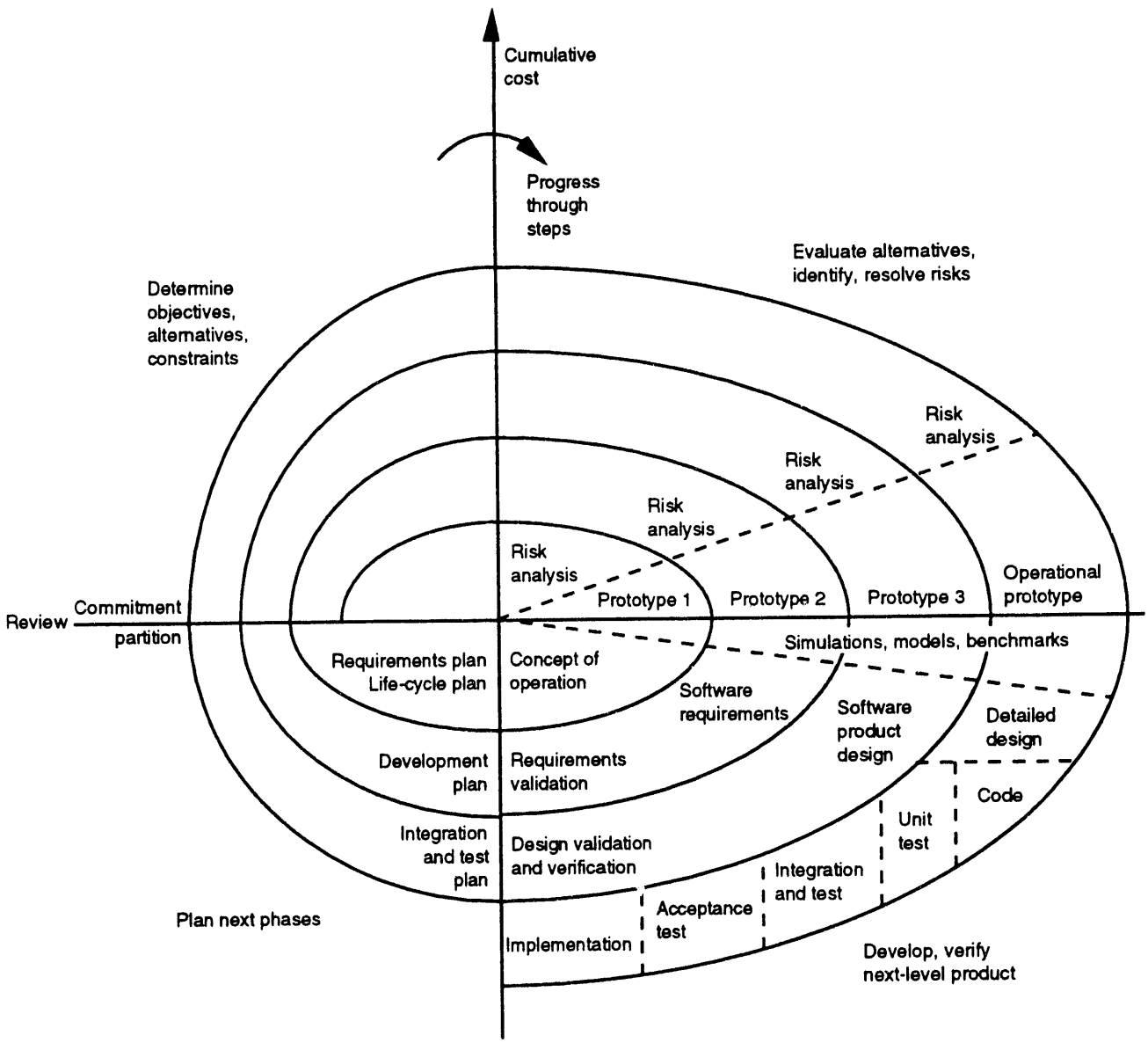


Figure 2-2. Waterfall Life Cycle Model



(Boehm 1988)

Figure 2-3. Spiral Life Cycle Model

Table 2-1. Persistence Classes and Fault Sources

	Design	Operational	Transient
Hardware component	X	X	X
Software component	X		
Input data	X	X	
Permanent state	X	X	
Temporary state	X	X	
Topological	X		
Operator	X	X	X
User	X	X	X
Environmental	X	X	X
Unknown			X

2.4.1. Fault Classifications

Faults and failures can be classified by several more-or-less orthogonal measures. This is important, because the classification may affect the depth and method of analysis and problem resolution, as well as the preferred modeling technique.

Faults can be classified by the persistence and source of the fault. This is described in the two subsections of this section. Terms defined in each subsection are used in other subsections.

2.4.1.1. Fault Persistence

Any fault falls into one of the following three classes (Kopetz 1985):

- A *design fault* is a fault that can be corrected by redesign. Most software and topological faults fall into this class, but relatively few hardware faults do. Design faults are sometimes called *removable* faults, and are generally modeled by reliability growth models (See Appendix A.3.). One design fault can cause many errors and failures before it is diagnosed and corrected. Design faults are usually quite expensive to correct if they are not discovered until the product is in operation.
- An *operational fault* is a fault where some portion of the computer system breaks and must be repaired in order to return the system to a state that meets the design specifications. Examples include electronic and mechanical faults, database corruption, and some operator faults. Operational faults are sometimes called *non-removable* faults. When calculating fault rates for operational faults, it is generally assumed that the entity that has failed is in the steady-state portion of its life, so operational fault rates are constant. As with design faults, an operational fault may cause many errors before being identified and repaired.
- A *transient fault* is a fault that does cause a computer system failure, but is no longer present when the system is restarted. Frequently the basic cause of a transient fault cannot be determined. Redesign or repair has no effect in this case, although redesign can affect the frequency of transient faults. Examples include power supply noise and operating system timing errors. While an underlying problem may actually exist, no action is taken to correct it (or the fault would fall into

one of the other classes). In some computer systems, 50–80% of all faults are transient. The frequency of operating system faults, for example, is typically dependent on system load and composition.

The class of transient faults actually includes two different types of event; they are grouped together here since it is generally impossible to distinguish between them. Some events are truly transient; a classic (though speculative) example is a cosmic ray that flips a single memory bit. The other type is an event that really is a design or operational fault, but this is not known when it occurs. That is, it looks like the first type of transient event. If the cause is never discovered, no real harm is done in placing it in this class. However, if the cause is eventually determined, the event should be classified properly; this may well require recalculation of reliability measures.

A computer system is constructed according to some specification. If the system fails, but still meets the specification, then the specification was wrong. This is a design fault. If, however, the system ceases to meet the specification and fails, then the underlying fault is an operational fault. A broken wire is an example. If the specification is correct, but the system fails momentarily and then recovers on its own, the fault is transient.

Many electronic systems, and some mechanical systems, have a three stage life cycle with respect to fault persistence. When the device is first constructed, it will have a fairly high fault rate due to undetected design faults and "burn-in" operational faults. This fault rate decreases for a period of time, after that the device enters its normal life period. During this (hopefully quite long) period, the failure rate is approximately constant, and is due primarily to operational and transient faults, with perhaps a few remaining design faults. Eventually the device begins to wear out, and enters the terminal stage of its life. Here the fault rate increases rapidly as the probability of an operational fault goes up at an increasing rate. It should be noted that in many cases the end of the product's useful life is defined by this increase in the fault rate.

The behavior described in the last paragraph results in a failure rate curve termed the "bathtub" curve. It was originally designed to model electronic failure rates. There is a somewhat analogous situation for software. When a software product is first released, there may be many failures in the field for some period of time. As the underlying faults are corrected and new releases are sent to the customers, the failure rate should decrease until a more-or-less steady state is reached. Over time, the maintenance and enhancement process may perturb the software structure sufficiently that new faults are introduced faster than old ones are removed. The failure rate may then go up, and a complete redesign is in order.

While this behavior looks similar to that described for electronic systems, the causal factors are quite different. One should be very careful when attempting to extrapolate from one to the other.

2.4.1.2. Source of Faults in Computer Systems

Fault sources can be classified into a number of categories; ten are given here. For each one, the source is described briefly, and the types of persistence that are possible is discussed.

- A *hardware fault* is a fault in a hardware component, and can be of any of the three persistence types. Application systems rarely encounter hardware design faults. Transient hardware faults are very frequent in some systems.
- A *software fault* is a bug in a program. In theory, all such are design faults. Dhillon (1987) classifies software faults into the following eight categories:
  - — Logic faults
  - — Interface faults
  - — Data definition faults
  - — Database faults
  - — Input/output faults
  - — Computational faults
  - — Data handling faults
  - — Miscellaneous faults
- An *input data fault* is a mistake in the input. It could be a design fault (connecting a sensor to the wrong device is an example) or an operational fault (if a user supplies the wrong data).
- A *permanent state fault* is a fault in state data that is recorded on non-volatile storage media (such as disk). Both design and operational faults are possible. The use of a data structure definition that does not accurately reflect the relationships among the data items is an example of a design fault. The failure of a program might cause an erroneous value to be stored in a file, causing an operational fault in the file.

- A *temporary state fault* is a fault in state data that is recorded on volatile media (such as main memory). Both design and operational faults are possible. The primary reason to separate this from permanent state faults is to allow for the possibility of different failure rates.
- A *topological fault* is a fault caused by a mistake in computer system architecture, not with the component parts. All such faults are design faults. Notice that the failure of a cable is considered a hardware operational fault, not a topological fault.
- An *operator fault* is a mistake by the operator. Any of the three types are possible. A design fault occurs if the instructions provided to the operator are incorrect; this is sometimes called a *procedure fault*. An operational fault would occur if the instructions are correct, but the operator misunderstands and doesn't follow them. A transient fault would occur if the operator is attempting to follow the instructions, but makes an unintended mistake. Hitting the wrong key on a keyboard is an example. (One goal of display screen design is to reduce the probability of transient operator errors.)
- A *user fault* differs from an operator fault only because of the different type of person involved; operators and users can be expected to have different fault rates.
- An *environmental fault* is a fault that occurs outside the boundary of the computer system, but that affects the system. Any of the three types is possible. Failure to provide an uninterruptible power supply (UPS) would be a design fault, while failure of the UPS would be an operational fault. A voltage spike on a power line is an example of an environmentally induced transient fault.
- An *unknown fault* is any fault whose source class is never identified. Unfortunately, in some computer systems many faults occur whose source cannot be identified. All such faults are transient (more or less by definition), and this category may well include a plurality of system faults. Another problem is that the underlying problem may be identified at a later time (possibly months later), so there is a certain impermanence about this category. It generally happens that some information is available about the source of the fault, but not sufficient information to allow the source to be completely identified. For example, it might only be known that there is a fault in a communication system.

Table 2-1 shows which persistence classes may occur for each of the ten fault sources.

2.4.2. Failure Classifications

Three aspects of classifying failures are given below; there are others. These are particularly relevant to later discussion in this report.

2.4.2.1. Failure Modes

Different failure modes can have different effects on a computer system. The following definitions apply (Smith 1972).

- A *sudden failure* is a failure that could not be anticipated by prior examination. That is, the failure is unexpected.
- A *gradual failure* is a failure that could be anticipated by prior examination. That is, the system goes into a period of degraded operation before the failure actually occurs.
- A *partial failure* is a failure resulting in deviations in characteristics beyond specified limits but not such as to cause complete lack of the required function.
- A *complete failure* is a failure resulting in deviations in characteristics beyond specified limits such as to cause complete lack of the required function. The limits referred to in this category are special limits specified for this purpose.
- A *catastrophic failure* is a failure that is both sudden and complete.
- A *degradation failure* is a failure that is both gradual and partial.

2.4.2.2. The Scope of Failures

Failures can be assigned to one of three classes, depending on the scope of their effects (Anderson 1983).

- A failure is *internal* if it can be adequately handled by the device or process in which the failure is detected.
- A failure is *limited* if it is not internal, but if the effects are limited to that device or process.
- A failure is *pervasive* if it results in failures of other devices or processes.

2.4.2.3. The Effects of Failures on Safety

Finally, it is possible to classify application systems by the effect of failures on safety.

- A system is intrinsically safe if the system has no hazardous states.
- A system is termed fail safe if a hazardous state may be entered, but the system will prevent an accident from resulting from the hazard. An example would be a facility in a reactor that forces a controlled shutdown in case a hazardous state is entered, so that no radiation escapes.
- A system controls accidents if a hazardous state may be entered and an accident may occur, but the system will mitigate the consequences of the accident. An example is the containment shell of a reactor, designed to preclude a radiation release into the environment if an accident did occur.
- A system gives warning of hazards if a failure may result in a hazardous state, but the system issues a warning that allows trained personnel to apply procedures outside the system to recover from the hazard or mitigate the accident. For example, a reactor computer protection system might notify the operator that a hazardous state has been entered, permitting the operator to "hit the panic button" and force a shutdown in such a way that the computer system is not involved.
- Finally, a system is fail dangerous, or creates an uncontrolled hazard, if system failure can cause an uncontrolled accident.

2.5. Software Qualities

A large number of factors have been identified by various theoreticians and practitioners that affect the quality of software. Many of these are very difficult to quantify. The discussion here is based on IEEE 610.12, Evans 1987, Pressman 1987, and Vincent 1988. The latter two references based their own discussion on McCall 1977. The discussion concentrates on defining those terms that appear important to the design of reactor protection computer systems. Quotations in this section come from the references listed above.

Access Control. The term "access control" relates to "those attributes of the software that provide for control of the access to software and data." In a reactor protection system, this refers to the ability of the utility to prevent unauthorized changes to either software or data within the computer system, incorrect input signals being sent to the computer system by intervention of a human agent, incorrect commands from the operator, and any other forms of tampering. Access control should consider both inadvertent and malicious penetration.

Accuracy. Accuracy refers to "those attributes of the software that provide the required precision in calculations and outputs." In some situations, this can require a careful error analysis of numerical algorithms.

Auditability. Auditability refers to the "ease with which conformance to standards can be checked." The careful development of project plans, adherence to those plans, and proper record keeping can help make audits easier, more thorough and less intrusive. Sections 3 and 4 discuss this topic in great depth.

Completeness. Completeness properties are "those attributes of the software that provide full implementation of the functions required." A software design is complete if all requirements are fulfilled in the design. A software implementation is complete if the code fully implements the design.

Consistency. Consistency is defined as "the degree of uniformity, standardization and freedom from contradictions among the documents or parts of a system or component." Standardized error handling is an example of consistency. Requirements are consistent if they do not require the system to carry out some function, and under the same conditions to carry out its negation. An inconsistent design might cause the system to send incompatible signals to one or more actuators, causing the protection system to attempt contradictory actions. An example would be starting a pump but not opening the intake value.

Correctness. Correctness refers to the "extent to which a program satisfies its specifications and fulfills the user's mission objectives." This is a broader definition than that given for completeness. It is worth noting that some of the documents referenced at the beginning of the section essentially equate correctness with completeness, while others distinguish between them. The IEEE Standard 610.12 gives both forms of definition.

Expandability. Expandability attributes are "those attributes of the software that provide for expansion of data storage requirements or

computational functions." The word "extendibility" is sometimes used as a synonym.

Generality. Generality is "the degree to which a system or component performs a broad range of functions." This is not necessarily a desirable attribute of a reactor protection system if the generality encompasses functionality beyond simply protecting the reactor.

Software Instrumentation. Instrumentation refers to "those attributes of the software that provide for measurement of usage or identification of errors." A well-instrumented system can monitor its own operation, and detect errors in that operation. Software instrumentation can be used to monitor the hardware operation as well as its own operation. A hardware device such as a watch-dog timer can be used to help monitor the software operation. If instrumentation is required for a computer system, it may have a considerable effect on the system design, so must be considered as part of that design.

Modularity. Modularity attributes are "those attributes of the software that provide a structure of highly independent modules." To achieve modularity, the protection computer system should be divided into discrete hardware and software components in such a way that a change to one component has minimal impact on the remaining modules. Modularity is measured by cohesion and coupling (Yourdon 1979).

Operability. Operability refers to "those attributes of the software that determine operation and procedures concerned with the operation of the software." This quality is concerned with the man-machine interface, and measures the ease with which the operators can use the system. This is particularly a concern during off-normal and emergency conditions when confusion may be high and mistakes may be unfortunate.

Robustness. Robustness refers to "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions." This quality is sometimes referred to as "error tolerance" and may be implemented by fault tolerance or design diversity.

Simplicity. Simplicity attributes are "those attributes that provide implementation of functions in the most understandable manner." It can be thought of as the absence of complexity. This is one of the more important design qualities for a reactor computer protection system, and is quite difficult to quantify. See Preckshot 1992 for additional information on complexity and scalability.

A particularly important aspect of complexity is the distinction between functional complexity and structural complexity. The former refers to a system that attempts to carry out many disparate functions, and is controlled by limiting the goals of the system. The latter refers to the method of carrying out the functions, and may be controlled by redesigning the system to carry out the same functions in a simpler way.

Testability. Testability refers to "the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met."

Traceability. Traceability attributes are "those attributes of the software that provide a thread from the requirements to the implementation with respect to the specific development and operational environment."

3. LIFE CYCLE SOFTWARE RELIABILITY AND SAFETY ACTIVITIES

Much has been written about software engineering and how a well-structured development life cycle can help in the production of correct maintainable software systems. Many standard software engineering activities should be performed for any software project, so are not discussed in this report. Instead, the report concentrates on the additional activities required for a software project in which safety is a prime concern. Refer to a general text, such as Macro 1990 or Pressman 1987, for general information on software engineering.

Any software development project can be discussed from a number of different viewpoints. Examples include the customer, the user, the developer, the project manager, the general manager, and the assessor. The viewpoint that is presumed will have a considerable effect on the topics discussed, and particularly on the emphasis placed on different aspects of those topics. The interest here is the viewpoint of the assessor. This is a person (or group of people) who evaluates both the development process and the products of that process for assurance that they meet some externally-imposed standard. In this report, those standards will relate to the reliability of the software products and the safety of the application in which the software is embedded. The assessor may be a person in the development organization charged with the duty of assuring reliability and safety, a person in an independent auditing organization, or an employee of a regulatory agency. The difference among these assessors should be the reporting paths, not the technical activities that are carried out. Consequently no distinction is made here among the different types of assessor.

Since this report is written from the viewpoint of the assessor, the production of documents is emphasized in this report. The documents provide the evidence that required activities have actually taken place. There is some danger that the software developer will concentrate on the creation of the documents rather than the creation of safe reliable software. The assessor must be constantly on guard for this activity. The software runs the protection system, not the documents. There is heavy emphasis below on planning: creating and following the plans that are necessary to the development of software where safety is a particular concern.

The documents that an assessor should expect to have available, and their contents, is the subject of this section of the report. The process of assessing these documents is discussed in Section 4.

3.1. Planning Activities

Fundamental to the effective management of any engineering project is the planning that goes into the project. This is especially true where extreme reliability and safety are of concern. While there are general issues of avoiding cost and schedule overruns, the particular concern here is safety. Unless a management plan exists, and is followed, the probability is high that some safety concerns will be overlooked at some point in the project lifetime, or lack of time or money near the end of the development period will cause safety concerns to be ignored, or testing will be abridged. It should be noted that the time/money/safety tradeoff is a very difficult management issue requiring very wise judgment. No project manager should be allowed to claim "safety" as an excuse for unconscionable cost or schedule overruns. On the other hand, the project manager should also not be allowed to compromise safety in an effort to meet totally artificial schedule and budget constraints.

For a computer-based safety system, a number of documents will result from the planning activity. These are discussed in this section, insofar as safety is an issue. For example, a software management plan will generally involve non-safety aspects of the development project, which go beyond the discussion in Section 3.1.1.

Software project planning cannot take place in isolation from the rest of the reactor development. It is assumed that a number of documents are available to the software project team. At minimum, the following must exist:

- **Hazards analysis**. This identifies hazardous reactor system states, sequences of actions that can cause the reactor to enter a hazardous state, sequences of actions intended to return the reactor from a hazardous state to a nonhazardous state, and actions intended to mitigate the consequences of an accident.

- **High level reactor system design.** This identifies those functions that will be performed by the protection system, and includes a specification of those safety-related actions that will be required of the software in order to prevent the reactor from entering a hazardous state, move the reactor from a hazardous state to a non-hazardous state, or mitigate the consequences of an accident.
- **Interfaces between the protection computer system and the rest of the reactor protection system.** That is, what signals must be obtained from sensors and what signals must be provided to actuators by the computer system. Interfaces also include display devices intended for man-machine interaction.

Planning a software development project can be a complex process involving a hierarchy of activities. The entire process is beyond the scope of this report. Figure 3-1, taken from Evans 1983 (copyright 1983 by Michael Evans, Pamela Piazza, and James Dolkas. Reprinted by permission of John Wiley & Sons, Inc.), gives a hint as to the activities involved. Planning is discussed in detail in Pressman 1987.

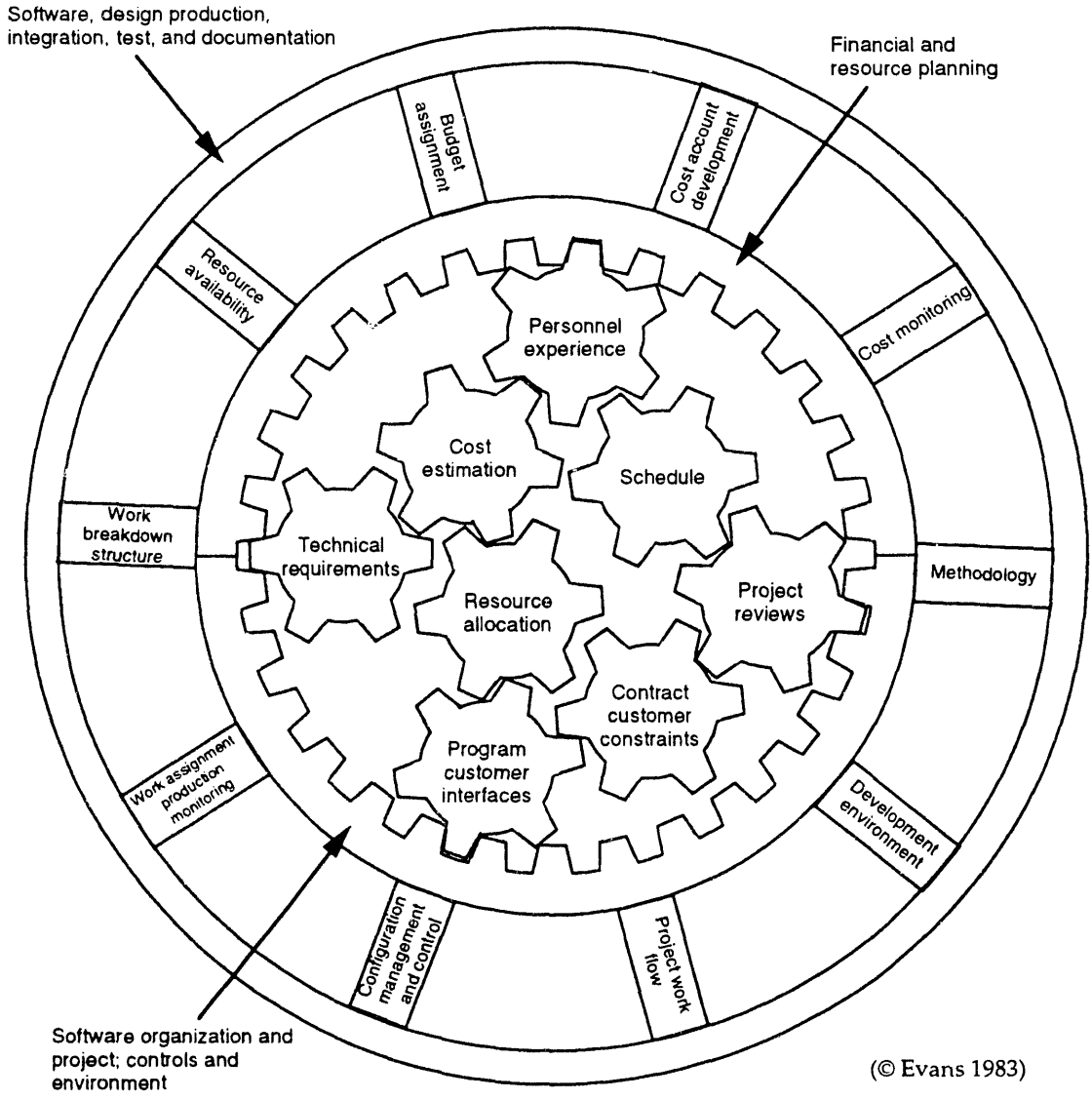


Figure 3-1. Software Planning Activities

The result of the planning activity will be a set of documents that will be used to oversee the development project. These may be packaged as separate documents, combined into a fewer number of documents, or combined with similar documents used by the larger reactor project. For example, the developer might choose to include the software V&V plan in the software project management plan, or to include the software configuration management plan in a project-wide configuration management plan. Such packaging concerns are beyond the scope of this report. Since some method is necessary in order to discuss documents, the report assumes that separate documents will exist. The documents resulting from planning include the following minimum set; additional documents may be required by the development organization as part of their standard business procedures, or by the assessor due to the nature of the particular project.

- Software Project Management Plan
- Software Quality Assurance Plan
- Software Configuration Management Plan
- Software Verification and Validation Plan
- Software Safety Plan
- Software Development Plan
- Software Integration Plan
- Software Installation Plan
- Software Maintenance Plan
- Software Training Plan
- Software Operations Plan

The actual time at which these documents will be produced depends on the life cycle used by the software developer. The Software Project Management Plan will always need to be done early in the life cycle, since the entire management effort is dependent on it. However, documents such as the Software Operations Plan might be delayed until the software system is ready to install.

3.1.1. Software Project Management Plan

The software project management plan (SPMP) is the basic governing document for the entire development effort. Project oversight, control, reporting, review, and assessment are all carried out within the scope of the SPMP.

One method of organizing the SPMP is to use IEEE Standard 1058; this is done here. Other methods are possible, provided that the topics discussed below are addressed. The plan contents can be roughly divided into several categories: introduction and overview, project organization, managerial processes, technical processes, and budgets and schedules. A sample table of contents, based on IEEE 1058, is shown in Figure 3-2. Those aspects of the plan that directly affect safety are discussed next.

1. Introduction
   - 1.1. Project Overview
   - 1.2. Project Deliverables
   - 1.3. Evolution of the SPMP
   - 1.4. Reference Materials
   - 1.5. Definitions and Acronyms
2. Project Organization
   - 2.1. Process Model
   - 2.2. Organizational Structure
   - 2.3. Organizational Boundaries and Interfaces
   - 2.4. Project Responsibilities
3. Managerial Process
   - 3.1. Management Objectives and Priorities
   - 3.2. Assumptions, Dependencies and Constraints
   - 3.3. Risk Management
   - 3.4. Monitoring and Controlling Mechanisms
   - 3.5. Staffing Plan
4. Technical Process
   - 4.1. Methods, Tools and Techniques
   - 4.2. Software Documentation
   - 4.3. Project Support Functions
5. Work Packages, Schedule and Budget
   - 5.1. Work Packages
   - 5.2. Dependencies
   - 5.3. Resource Requirements
   - 5.4. Budget and Resource Allocation
   - 5.5. Schedule
6. Additional Components

Index

Appendices

Figure 3-2. Outline of a Software Project Management Plan

A combination of text and graphics may be used to create and document the SPMP. PERT charts, organization charts, matrix diagrams or other formats are frequently useful.

3.1.1.1. Project Organization

This portion of the SPMP addresses organizational issues; specifically, the process model, organizational structure, boundaries and interfaces, and project responsibilities. The following items should be discussed in this portion of the plan.

- Process Model. Define the relationships among major project functions and activities. The following specifications must be provided:
  - — Timing of major milestones.
  - — Project baselines.
  - — Timing of project reviews and audits.
  - — Work products of the project.
  - — Project deliverables.
- Organization Structure. Describe the internal management structure of the project.
  - — Lines of authority.
  - — Responsibility for the various aspects of the project.
  - — Lines of communication within the project.
  - — The means by which the SPMP will be updated if the project organization changes. Note that the SPMP should be under configuration control; see Section 3.1.3.
- Organization Boundaries. Describe the administrative and managerial boundaries, and interfaces across those boundaries, between the project and the following external entities.
  - — The parent organization.
  - — The customer organization.
  - — Any subcontractor organizations.
  - — The regulatory and auditor organizations.
  - — Support organizations, including quality assurance, verification and validation, and configuration management.
- Project responsibilities. State the nature of each major project function and activity, and identify by name the individuals who are responsible for them. Give the method by which these names can be changed during the life of the project.

3.1.1.2. Project Management Procedures

This section of the SPMP will describe the management procedures that will be followed during the project development life cycle. Topics that can affect safety are listed here; the development organization will normally include additional information in order to completely describe the management procedures. The following aspects of the SPMP fall into the category of management procedures.

- Project Priorities. Describe the priorities for management activities during the project. Topics include:
  - — Relative priorities among safety requirements, functional requirements, schedule and budget.
  - — Frequency and mechanisms for reporting.
- Project Assumptions, Dependencies and Constraints. State:
  - — The assumptions upon which the project is based.
  - — The external events upon which the project is dependent.
  - — The constraints under which the project will be conducted.
- Risk Management. Identify and assess the risk factors associated with the project. All of the items listed here may have an impact on safety; this impact must be described here, with a method for managing that risk.
  - — Financial risks.
  - — Schedule risks.
  - — Contractual risks.
  - — Technology change risks.
  - — Size and complexity risks.
  - — Scale-up risks.
- Monitoring and Controlling Methods. Describe reporting requirements, report formats, information flows, and review and audit mechanisms.
  - — Internal reporting—within the development organization.

— External reporting—to auditors and regulators.

- Staffing. Specify the numbers and types of personnel required in order to achieve a reliable software system that meets safety requirements.
  - — Skill levels required.
  - — Start times and duration of needs.
  - — Training requirements.

3.1.1.3. Project Technical Procedures

This section of the SPMP will describe management aspects of the technical procedures that will be followed during the project development life cycle. Topics that can affect safety are listed here; the development organization will normally include additional information in order to completely describe the technical procedures. In some cases, these procedures may be documented in other documents, and this portion of the SPMP will merely reference those documents. In particular, the technical aspects of the development effort are described in the Software Development Plan; see Section 3.1.5. The difference is one of emphasis: the SPMP is directed at the project management personnel, while the Software Development Plan is directed at the project technical personnel. The following topics should be discussed.

- Methods, Tools, and Techniques. Specify all of the methods, tools, and techniques that will be used to develop the product. The following list is meant to be indicative.
  - — Computing systems to be used for software development.
  - — Development methods.
  - — Programming languages.
  - — Computer-assisted software engineering (CASE) tools.
  - — Technical standards to be followed.
  - — Company development procedures.
  - — Company programming style.
- Software Documentation. Describe all of the technical documentation that will be required for the project. The documents listed below are considered mandatory; additional documents may be included at the option of the development organization or auditing organization. Additional documents may be required by other plans. In particular, the Verification and Validation Plan and the Configuration Management Plan will require documents that describe assessments done for each life cycle phase. Discuss milestones, baselines, reviews, and sign-offs for each document.
  - — Software Development Plan.
  - — Software Requirements Specification.
  - — Requirements Safety Analysis.
  - — Hardware/Software Architecture.
  - — Software Design Specification.
  - — Design Safety Analysis.
  - — Unit Development Folders.
  - — Code Safety Analysis.
  - — System Build Specification.
  - — Integration Safety Analysis.
  - — Validation Safety Analysis.
  - — Installation Procedures.
  - — Operations Manuals.
  - — Installation Configuration Tables.
  - — Installation Safety Analysis.
  - — Change Safety Analysis.
- Project Support Functions. Describe all technical support functions for the project. In many cases, these will have their own plans, and the SPMP may simply refer to those. Describe (either here or in the supporting plans) responsibilities, resource requirements, schedules, and budgets for each supporting activity. Support functions include:
  - — Software quality assurance.
  - — Software configuration management.
  - — Software verification and validation (including testing).
  - — Software safety management.
  - — Software reviews and audits.

3.1.2 Software Quality Assurance Plan

Quality assurance (QA) is defined by IEEE as "a planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements." Software quality assurance (SQA) is the portion of

general quality assurance that applies to a software product. The SQA plan describes how the quality of the software will be assured by the development organization. It may exist as a separate document, or be part of the general reactor QA plan. Here, the first is assumed to provide specificity to the discussion.

There will be considerable overlap between the SQA Plan and the other project plans. The SQA Plan will generally reference such documents, and limit the discussion in the SQA Plan itself to matters of particular concern to SQA activities. For example, the section on code control may reference the Software Configuration Management Plan, and describe the methods by which the SQA organization will ensure that this plan is followed.

The description here is based on ANSI/IEEE standard 730.1, *Software Quality Assurance Plans*. The use of this standard is discussed in a draft IEEE guide, 730.2, *Guide for Software Assurance Planning*. A sample table of contents for a SQA plan is shown in Figure 3-3. It is based on the IEEE standard. The developer need not follow this sample, provided that the requirements listed below are included in his own plan. Concerns that are unlikely to directly affect safety are not discussed in this list of requirements.

1. SQA Organization. Describe the organizational structure of the SQA effort. Major topics to discuss include:
   - — Major SQA organizational elements and linkages between them.
   - — Organizational independence or dependence of the SQA organization from the development organization.
2. SQA Management Tasks. Describe the major tasks that will be required for the SQA activity.
   - — Describe that portion of the software life cycle subject to quality assurance oversight.
   - — Describe the tasks to be performed for quality assurance. These tasks are discussed in detail in Sections 3-14 of the SQA Plan.
   - — Describe the relationships between the SQA tasks and the project review points.
3. SQA Responsibilities. Identify the organizational elements responsible for each SQA task.
   - — Identify the persons responsible for the SQA Plan.
   - — Identify the person responsible for overall software quality, by name.
4. Documentation. List the documents subject to SQA oversight.
   - — List the documents. This list should generally coincide with the list provided in the Software Project Management Plan, as discussed in Section 3.1.1.
   - — Discuss how each document will be reviewed by the SQA organization for adequacy.
5. Standards, Practices, Conventions, and Metrics. Describe all safety-related standards, practices, conventions, and metrics that will be used during the development process.
   - — Identify the life cycle phase to which each standard, practices, conventions, and metrics applies.

1. Introduction
   - 1.1. Purpose
   - 1.2. Scope
   - 1.3. Definitions and Acronyms
   - 1.4. References
2. Management
   - 2.1. Organization
   - 2.2. Tasks
   - 2.3. Responsibilities
3. Documentation
4. Standards, Practices, Conventions and Metrics
5. Reviews and Audits
6. Test
7. Problem Reporting and Corrective Action
8. Tools, Techniques and Methodologies
9. Code Control
10. Media Control
11. Supplier Control
12. Records Collection, Maintenance and Retention
13. Training
14. Risk Management

Figure 3-3. Outline of a Software Quality Assurance Plan

— Specify how compliance with each standard, practices, conventions, and metrics will be assured.

— The following (from IEEE 730.2) lists standards, practices, conventions, and metrics that may apply to the different life cycle phases:

* Documentation standards.
* Logic structure standards.
* Coding standards.
* Commentary standards.
* Testing standards and practices.
* Product and process metrics.

6. Reviews and Audits. Describe the reviews and audits to be carried out during the development process.

— Identify each technical and managerial review and audit.

— Describe how each review and audit will be carried out.

— Describe how follow-up actions will be implemented and verified.

— The following (from IEEE 730.1) lists a minimal set of reviews and audits; the actual set should be determined by the project management and the SQA organization, acting together:

* Software Requirements Review.
* Preliminary Design Review.
* Critical Design Review.
* Software Verification and Validation Plan Review.
* Functional Audit.
* Physical Audit.
* In-Process Audits.
* Managerial Reviews.

7. Test. Describe any safety-related tests that will be required on the software that are not included in the Software Verification and Validation Plan.

— Identify all such tests.

— Describe how the tests will be carried out.

8. Problem Reporting and Corrective Action. Describe how safety-related problems encountered during development will be reported, tracked, and resolved.

— Identify responsibilities for reporting and tracking problems.

— Identify responsibilities for ensuring that all safety-related problems are resolved.

9. Tools, Techniques, and Methodologies. Discuss any special software tools, techniques, and methodologies that will be used to support the SQA activity.

— Identify each tool, technique, and methodology.

— Identify responsibilities for each tool, technique, and methodology.

10. Code Control. Describe how source and object code will be controlled during the project development. (This is discussed further in Section 3.1.3.)

11. Media Control. Describe the methods and facilities used to identify the media for each software product and documentation, including storage, copying, and retrieval.

12. Supplier Control. Describe the provisions used to assure that software provided by suppliers will meet established project requirements.

— Identify the methods to make sure that suppliers receive adequate and complete requirements.

— State the methods used to assure the suitability of previously-developed software for this project.

— Describe procedures to be used to provide assurance that suppliers SQA methods are satisfactory, and consistent with this SQA Plan.

13. Risk Management. Specify the methods and procedures employed to identify, assess, monitor, and control areas of risk, especially those relating to safety.

3.1.3. Software Configuration Management Plan

Software configuration management (SCM) is the process by which changes to the products of the software development effort are controlled. SCM consists of four major parts: the SCM plan (SCMP), the SCM baseline, the configuration control board and the configuration manager. The SCMP may exist as a document of its own, may be part of the SPMP, or may

be part of a larger project configuration management plan. Here, the first is assumed, simply as a vehicle for discussion.

This description is based on ANSI/IEEE standard 828, *Software Configuration Management Plans*. The use of this standard is discussed in another ANSI/IEEE document, 1042, *Guide to Software Configuration Management*. The latter includes, in AppendixA, an example of a SCM plan for a safety-critical embedded application.

See Babich 1986 for a general introduction to the topic of SCM.

The configuration baseline identifies the development products (termed *configuration items*) that will be under configuration control. The configuration control board (CCB) generally contains representatives from both customer and developer organizations, and approves all changes to the baseline. The configuration manager makes sure the changes are documented and oversees the process of making changes to the baseline.

A sample table of contents for a SCM plan is shown in Figure 3-4. It is based on IEEE 828. The developer need not follow this sample, provided that the requirements listed below are included in his own plan. As usual, issues that are unlikely to directly affect safety are not discussed in the list of requirements, which follows:

1. SCM Organization. Describe the organizational structure of the SCM effort. Major topics to discuss include:
   - — Major CM organizational elements and linkages between them.
   - — Organizational relationships involving the CCB.
2. SCM Responsibilities. The various responsibilities for configuration management are described under this heading.
   - — Organizational responsibilities for each safety-related SCM task.
   - — Relationships between the SCM organization and other organizations. For example, the quality assurance organization and the software development organization.
   - — Responsibilities of the CCB.
3. SCM Interface Control. Describe the methods that will be used for each of the following functions involving interfaces. All types of interfaces are included: among organizational elements, among software modules, between hardware and software, and so forth.
   - — Identify all interface specifications and control documents.
   - — Describe the method used to manage changes to interface specifications and related documents.
   - — Describe how it will be ensured that such changes are actually accomplished.
   - — Describe how the status of interface specifications and documents will be maintained.

1. Introduction
   - 1.1. Purpose
   - 1.2. Scope
   - 1.3. Definitions and Acronyms
   - 1.4. References
2. Management
   - 2.1. Organization
   - 2.2. SCM Responsibilities
   - 2.3. Interface Control
   - 2.4. SCM Plan Implementation
   - 2.5. Applicable Policies, Directives and Procedures
3. SCM Activities
   - 3.1. Configuration Identification
   - 3.2. Configuration Control
   - 3.3. Configuration Status Accounting
   - 3.4. Audits and Reviews
   - 3.5. Release Procedures
4. Tools, Techniques and Methodologies
5. Supplier Control
   - 5.1. Subcontractor Software
   - 5.2. Vendor Software
6. Records Collection and Retention

Figure 3-4. Outline of a Software Configuration Management Plan

4. SCM Plan Implementation. Establish the major CM milestones. These include such items as:
   - — Establishment of the CCB.
   - — Establishment of the configuration baseline.
   - — The schedule for configuration reviews and audits.
5. SCM Policies, Directives, and Procedures. Describe all policies, directives and procedures that will be used in configuration control. Many of the examples given in IEEE 828 are shown in the following list.
   - — Identification of levels of software in a hierarchical tree.
   - — Program and module naming conventions.
   - — Version level designations.
   - — Software product identification methods.
   - — Identification of specifications, test plans and procedures, programming manuals, and other documents.
   - — Media identification and file management identification.
   - — Document release process.
   - — Turnover or release of software products to a library function.
   - — Processing of problem reports, change requests, and change orders.
   - — Structure and operation of CCB.
   - — Release and acceptance of software products.
   - — Operation of software library systems, including methods of preparing, storing, and updating modules.
   - — Auditing of SCM activities.
   - — Level of testing required prior to entry of software into configuration management.
   - — File backup and storage procedures, including defense against fires and natural disasters.
6. Configuration Identification. Identify the initial baseline of items under configuration control. These are the initial configuration items, and will generally include many different types of things. The following list is meant to be illustrative, not exhaustive.
   - — Management plans and other management documents.
   - — Specifications, such as requirements and design specifications.
   - — User documentation.
   - — Test designs, test cases, and test procedure specifications.
   - — Test data and test generation procedures.
   - — Support software.
   - — Data dictionaries.
   - — Design graphics, such as CASE designs.
   - — Source, object, and executable code.
   - — Software libraries.
   - — Databases.
   - — Build instructions.
   - — Installation procedures.
   - — Installation configuration tables.
7. Configuration Control. Describe, in detail, the process by which change takes place.
   - — Describe the level of authority required to approve changes. This may vary according to the life cycle phase.
   - — Identify the routing of change requests for each life cycle phase.
   - — Describe procedures for software library control, including access control, read and write protection, CI protection, archive maintenance, change history and disaster recovery.
   - — Define the authority and makeup of the CCB. Identify members by name and position. State how changes to the CCB membership will be made known.
   - — State control procedures for nonreleased software, off-the-shelf software, and other special software products.
8. Configuration Status Accounting. Describe the method by which SCM reporting will take place.
   - — Describe how information on the status of the various configuration items will be collected, verified, stored and reported.
   - — Identify periodic reporting requirements.

9. Audits and Reviews. Define the role of the SCM organization in reviews and audits of the life cycle products. (The development organization may wish to address this topic in the V&V plan instead of here.)
   - — Identify which configuration items will be covered by each review and audit.
   - — State the procedures to be used for identifying and resolving problems that are discovered in reviews and audits.
10. Supplier Control. State how the SCM procedures will apply to purchased and subcontractor-developed software.

Changes to software requirements, design specifications, and code are almost certain to occur. It is necessary to keep track of such changes and their potential impact on safety. The purpose of configuration management is to manage the tracking, to make sure that the version of a configuration item (CI) that is being changed is actually the current version, and to always know the current release of CIs. If this is not done, there are a number of significant dangers. (1) A safety-related change might be lost, and not done. For example, a change to the design might not be carried through to the code. (2) Two or more people might be simultaneously changing the same CI, resulting in inconsistent changes or lost changes. The latter can occur when the second person to finish the change overwrites the change made by the first person. (3) A software release may be issued containing inconsistent versions of the various code modules.

Developing and carefully following an SCMP helps to avoid the aforementioned problems. When a change is desired, the CCB will examine the change and decide whether or not it should be implemented. For example, the change might be required for safety reasons, so should be approved. On the other hand, the suggested change might have a negative impact on some apparently unrelated safety issue; in this case, the change will have to be modified or rejected.

Once the CCB has approved the change, the configuration manager oversees the process of implementing the change. This is a process issue; the configuration manager is not involved in the technical aspects of the change. The technical person who will actually carry out the change will request control of the CI from the configuration manager. The latter makes sure the CI is available; that is, no other person is currently changing it. Only one person is permitted to work on the CI at any one time. Once the CI is available, it is "checked out" to the technical person, who is now responsible for it. He will make the needed changes. Once this is done, and any tests or reviews have taken place, the CI is returned to the configuration manager, who ensures that all procedures have been followed. The new document or module now becomes the new baseline, for use in the future for other changes, or for constructing product releases.

This report requires a SCM plan to be written and followed. At this point, only the plan is at issue. The later stages of the project life cycle will require assessment of the actual process of configuration management.

3.1.4. Software Verification and Validation Plan

Verification is the process that examines the products of each life cycle phase for compliance with the requirements and products of the previous phase. Validation is the process that compares the final software product with the original system requirements and established standards to be sure that the customer's expectations are met. The combination of verification and validation (V&V) processes generally includes both inspections and tests of intermediate and final products of the development effort. Figure 3-5, taken from IEEE Standard 1012, provides an overview of the process. See also ANS 7-4.3.2 and ANS 10.4.

Software V&V is discussed in detail in Barter 1993. That document gives a background discussion of V&V issues, the V&V plan, and V&V activities throughout the life cycle. Consequently, nothing more will be written here about the general issues.

The V&V plan can be based on ANSI/IEEE standard 1012, *Verification and Validation Plans*. A sample table on contents for a software V&V plan, based on this standard, is shown in Figure 3-6. The developer need not follow this sample, provided that all V&V requirements are included in the developer's own plan. The figure assumes that the software life cycle phases match the life cycle stages presented in Figure 2-1; the developer will need to make modifications to match the actual life cycle.

There can be considerable overlap between V&V activities and Quality Assurance and Safety Analysis activities. It is the responsibility of the project management to allocate responsibilities in a way suitable to the actual project at hand.

1. Management of Life Cycle V&V. The major portion of the V&V Plan will be the way in which V&V will be carried out through the life of the development project. If this is done carefully, then the V&V tasks for the remainder of the project consists of carrying out the Plan. In general, the following activities should be required for each phase of the life cycle:
   - Identify the V&V tasks for the life cycle phase.
   - Identify the methods that will be used to perform each task.
   - Specify the source and form for each input item required for each task.
   - Specify the purpose, target and form for each output item required for each task.
   - Specify the schedule for each V&V task.
   - Identify the resources required for each task, and describe how the resources will be made available to the V&V organization.
   - Identify the risks and assumptions associated with each V&V task. Risks include safety, cost, schedule and resources.
   - Identify the organizations or individuals responsible for performing each V&V task.
2. Requirements Phase V&V. The V&V Plan will describe how the various V&V tasks will be carried out during the requirements phase of the life cycle. The following tasks are identified in IEEE 1012 as a minimal set:
   - Software Requirements Traceability Analysis.
   - Software Requirements Evaluation.
   - Software Requirements Interface Analysis.
   - System Test Plan Generation.
   - Acceptance Test Plan Generation.
3. Design Phase V&V. The V&V Plan will describe how the various V&V tasks will be carried out during the design phase of the life cycle. The following tasks are identified in IEEE 1012 as a minimal set:
   - Software Design Traceability Analysis.
   - Software Design Evaluation.
   - Software Design Interface Analysis.
   - Component Test Plan Generation.
   - Integration Test Plan Generation.
   - Component Test Design Generation.
   - Integration Test Design Generation.
   - System Test Design Generation.
   - Acceptance Test Design Generation.
4. Implementation Phase V&V. The V&V Plan will describe how the various V&V tasks will be carried out during the implementation phase of the life cycle. The following tasks are identified in IEEE 1012 as a minimal set:
   - Source Code Traceability Analysis.
   - Source Code Evaluation.
   - Source Code Interface Analysis.
   - Source Code Documentation Analysis.
   - Component Test Case Generation.
   - Integration Test Case Generation.
   - System Test Case Generation.
   - Acceptance Test Case Generation.
   - Component Test Procedure Generation.
   - Integration Test Procedure Generation.
   - System Test Procedure Generation.
   - Acceptance Test Procedure Generation.
5. Integration Phase V&V. The V&V Plan will describe how the various V&V task will be carried out during the integration phase of the life cycle. The following tasks are identified in IEEE 1012 as a minimal set:
   - Integration Test Execution.
6. Validation Phase V&V. The V&V Plan will describe how the various V&V tasks will be carried out during the validation phase of the life cycle. The following tasks are identified in IEEE 1012 as a minimal set:
   - Acceptance Test Procedure Generation.
   - System Test Procedure Execution.
   - Acceptance Test Procedure Execution.
7. Installation Phase V&V. The V&V Plan will describe how the various V&V tasks will be carried out during the installation phase of the life cycle. The following tasks are identified in IEEE 1012 as a minimal set:
   - Installation Configuration Audit.
   - Final V&V Report Generation.

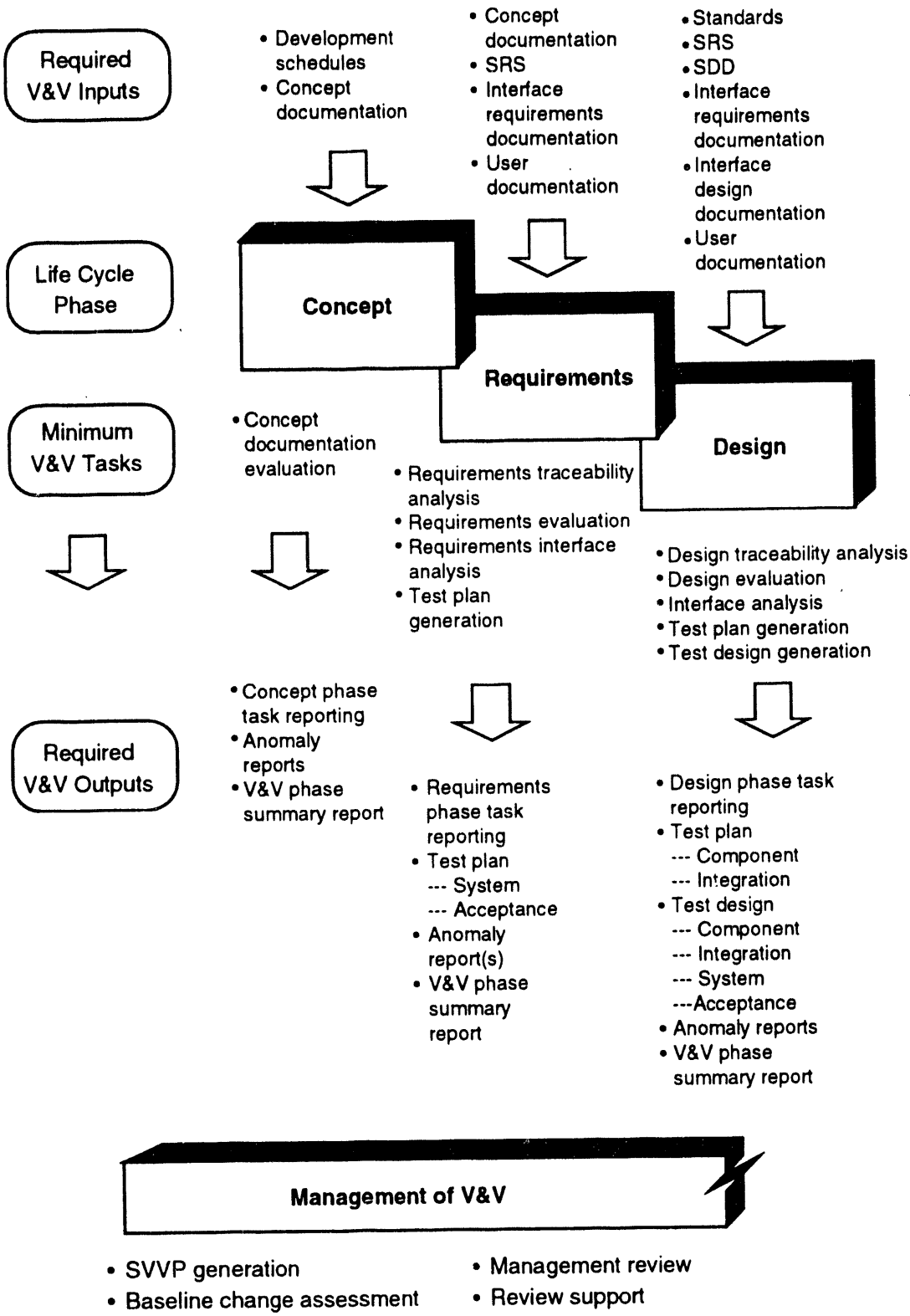


Figure 3-5. Verification and Validation Activities

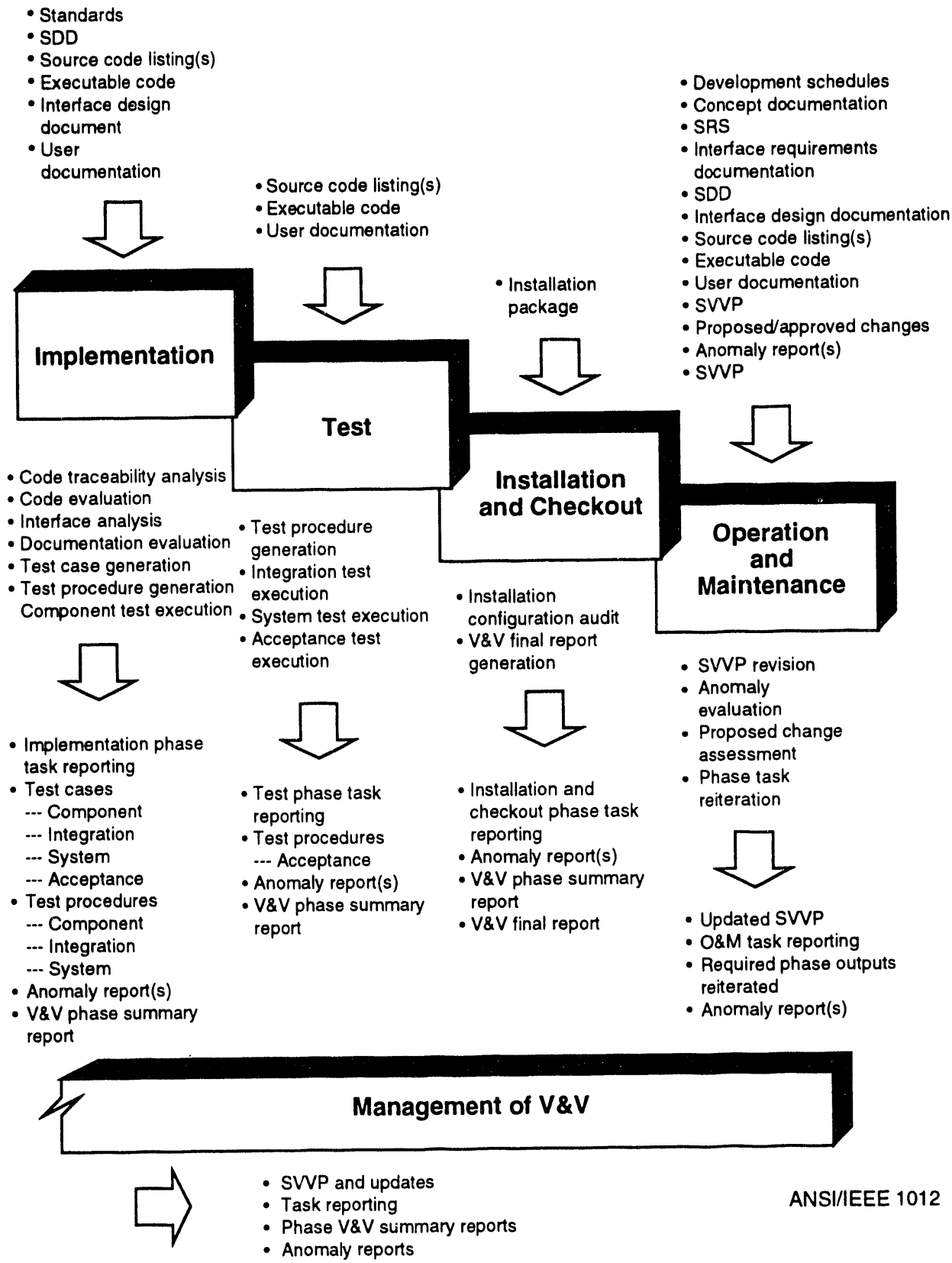


Figure 3-5. Verification and Validation Activities (cont.)

1. Introduction
   - 1.1. Purpose
   - 1.2. Scope
   - 1.3. Definitions and Acronyms
   - 1.4. References
2. Verification and Validation Overview
   - 2.1. Organization
   - 2.2. Master Schedule
   - 2.3. Resource Summary
   - 2.4. Responsibilities
   - 2.5. Tools, Techniques and Methodologies
3. Life Cycle Verification and Validation
   - 3.1. Management of V&V
   - 3.2. Requirements Phase V&V
   - 3.3. Design Phase V&V
   - 3.4. Implementation Phase V&V
   - 3.5. Integration Phase V&V
   - 3.6. Validation Phase V&V
   - 3.7. Installation Phase V&V
   - 3.8. Operation and Maintenance Phase V&V
4. Software Verification and Validation Reporting
5. Verification and Validation Administrative Procedures
   - 5.1. Anomaly Reporting and Resolution
   - 5.2. Task Iteration Policy
   - 5.3. Deviation Policy
   - 5.4. Control Procedures
   - 5.5. Standards, Policies and Conventions

Figure 3-6. Outline of a Software Verification and Validation Plan

3.1.5. Software Safety Plan

The Software Safety Plan (SSP) is required for safety-critical applications, such as reactor protection systems, to make sure that system safety concerns are properly considered during the software development. The discussion here is based on an IEEE Draft Standard, 1228, *Software Safety Plans*. A sample table of contents for an SSP, based on the draft standard, is shown in Figure 3-7. The developer need not follow this sample, provided that the requirements listed below are included the actual plan. In particular, the developer may wish to include many of the Safety Management requirements in other plans. For example, CM activities discussed here could be included in the developer's SCM plan. The entire SSP could be included within a global reactor safety plan. The software safety organization referred to in the list could be part of the system safety organization. The requirements for a software safety plan are listed next.

1. Introduction
   - 1.1. Purpose
   - 1.2. Scope
   - 1.3. Definitions and Acronyms
   - 1.4. References
2. Safety Management
   - 2.1. Organization and Responsibilities
   - 2.2. Resources
   - 2.3. Staff Qualifications and Training
   - 2.4. Software Life Cycle
   - 2.5. Documentation Requirements
   - 2.6. Software Safety Program Records
   - 2.7. Software Configuration Management Activities
   - 2.8. Software Quality Assurance Activities
   - 2.9. Tool Support and Approval
   - 2.10. Previously Developed or Purchased Software
   - 2.11. Subcontract Management
   - 2.12. Process Certification
3. Safety Engineering Practices during Software Development
   - 3.1. Requirements Safety Analysis
   - 3.2. Design Safety Analysis
   - 3.3. Code Safety Analysis
   - 3.4. Integration Test Safety Analysis
   - 3.5. Validation Test Safety Analysis
   - 3.6. Installation Test Safety Analysis
   - 3.7. Change Safety Analysis

Figure 3-7. Outline of a Software Safety Plan

1. Organization and Responsibilities. Describe the way in which software safety activities fit within the overall project safety activities and with the development organization. Major topics to discuss include:
   - — Organizational relationships involving the software safety organization.
   - — Lines of communication between the software safety organization, the system safety organization, and the software development organization.
   - — The oversight, review, and approval authority of the software safety organization.
   - — The authority of the software safety organization to enforce compliance with safety requirements and practices.
   - — The name and title of a single individual with overall responsibility for the conduct of the software safety program.
   - — The responsibilities of the software safety organization. Typical responsibilities include the following:
     * Preparation and update of the SSP.
     * Acquisition and allocation of resources to ensure effective implementation of the SSP.
     * Coordination of safety task planning with other organizational components and functions. This includes software development, system safety, software quality assurance, software configuration management, and software V&V.
     * Coordination of all technical issues related to software safety with other components of the development and support organizations and the regulators.
     * Creating, maintaining, and preserving adequate records to document the conduct of the software safety activities.
     * Participation in audits of the SSP implementation.
     * Training of safety and other personnel in methods, tools, and techniques used in the software safety tasks.
2. Resources. Specify the methods to be used to ensure there are adequate resources to implement the software safety program. Resources include (but are not limited to) financial, schedule, safety personnel, other personnel, computer and other equipment support, and tools.
   - — Specify the methods to be used to identify resource requirements.
   - — Specify the methods to be used to obtain and allocate these resources in the performance of the safety tasks.
   - — Specify the methods to be used to monitor the use of these resources.
3. Staff Qualifications and Training. Specify the qualifications and training required for the software safety personnel.
   - — Specify the personnel qualifications required for each of the following tasks:
     * Defining safety requirements.
     * Designing and implementing safety-critical portions of the protection system.
     * Performing software safety analysis tasks.
     * Testing safety-critical features of the protection system.
     * Auditing and certifying SSP implementation.
     * Performing process certification.
   - — Define training requirements and the methods by which training objectives will be met.
4. Software Life Cycle. Describe the relationship between software safety tasks and the development activities that will occur in the development organization's chosen life cycle.
5. Documentation Requirements. Specify the documents that will be required as part of the software safety program.
   - — Describe the method of documentation control that will be used. (The configuration management organization could be used for this purpose.)
   - — List all safety-specific documents that will be prepared. In particular, there must be documents describing the results of the various safety analyses described below in Sections 3.2.2, 3.3.3, 3.4.2, 3.5.2, 3.6.1, 3.7.5, and 3.8.1.
   - — Describe how other project documents must be augmented to address software safety activities. At minimum, the following topics must be addressed:

    * Software project management.
    * Software safety requirements.
    * Software development standards, practices, and conventions.
    * Software test documentation.
    * Software verification and validation documentation.
    * Software user and operator documentation.

6. Software Safety Program Records. Identify what software safety program records will be generated, maintained and preserved.
    — At minimum, the following records shall be kept.
        * Results of all safety analyses.
        * Information on suspected or verified safety problems that have been detected in pre-release or installed systems.
        * Results of audits performed on software safety program activity.
        * Results of safety tests carried out on the software system.
        * Records on training provided to software safety personnel and software development personnel.
    — Specify the person responsible for preserving software safety program records.
    — Specify what records will be used to ensure that each hazard, the person responsible for its management, and its status can be tracked throughout the software development life cycle.
7. Software Configuration Management Activities. Describe the interactions between the software configuration management organization and the software safety organization.
    — Describe the process by which changes to safety-critical software items are to be authorized and controlled.
    — Describe the role and responsibility of safety personnel in the change evaluation, change approval, and change verification processes.
    — Describe the relationship between the Configuration Control Board and the software safety organization.
    — Describe the methods for ensuring that configuration management of the following software meets the additional requirements necessary for safety-critical software:
        * Software development tools.
        * Previously developed software.
        * Purchased software.
        * Subcontractor-developed software.
8. Software Quality Assurance Activities. Describe the interactions between the software quality assurance organization and the software safety organization.
9. Tool Support and Approval. Specify the process to be used and the criteria to be applied in approving and controlling tool usage. This applies to development tools, and concerns the appropriateness of each tool to developing safety-critical code. The following aspects must be addressed:
    — Tool approval procedures.
    — Installation of upgrades to previously approved tools.
    — Withdrawal of previously approved tools.
    — Limitations imposed on tool use.
10. Previously Developed or Purchased Software. State the actions that will take place to ensure that previously developed or purchased (PDP) software meet the safety-related requirements of the development project.
    — Define the role of the software safety organization in approving PDP software.
    — Describe the approval process. At minimum, the following steps should be performed for PDP software that will be used in safety-critical applications:
        * Determine the interfaces to and functionality of the PDP software.
        * Identify relevant documents that are available to the obtaining organization, and determine their status.
        * Determine the conformance of the PDP software to published specifications.
        * Identify the capabilities and limitations of the PDP software with respect to the safety requirements of the development project.
        * Using an approved test plan, test the safety-critical features of the PDP

software in isolation from any other software.

* Using an approved test plan, test the safety-critical features of the PDP software in conjunction with other software with which it interacts.
* Perform a risk assessment to determine if the use of the PDP software will result in undertaking an acceptable level of risk even if unforeseen hazards result in a failure.

11. Subcontract Management. Specify the method for ensuring that subcontractor software meets the requirements of the software safety program.
    - — Describe how subcontractors will be controlled to ensure that they meet the requirements of the software safety plan.
    - — Describe how the capabilities of the subcontractor to support the software safety program requirements will be determined.
    - — Describe how the subcontractor will be monitored to ensure his adherence to the requirements of the SSP.
    - — Describe the process to be used to assign responsibility for, and track the status of, unresolved hazards identified or impacting the subcontractor.
12. Process Certification. Specify the method to be used (if any) to certify that the software product was produced in accordance with the SSP processes.
13. Safety Analyses. Specify the various safety analyses that will be performed for each stage of the life cycle. These are discussed in detail below in Sections 3.2.2, 3.3.3, 3.4.2, 3.5.2, 3.6.1, 3.7.5, and 3.8.1.
    - — Specify which safety analyses will be carried out for each life cycle stage.
    - — Name the person responsible for each analysis.
    - — Describe the review procedures that will be carried out for each analysis.
    - — Describe the documentation that will be required for each analysis.

3.1.6. Software Development Plan

The Software Development Plan provides necessary information on the technical aspects of the development project, that are required by the development team in order to carry out the project. Some of the topics that should be discussed in this plan were also listed for the Software Project Management Plan discussed in Section 3.1.1. The latter document is directed at the project management personnel, so emphasizes the management aspects of the development effort. The document discussed here emphasizes the technical aspects of the development effort, and is directed at the technical personnel.

A sample table of contents for a Software Development Plan is shown in Figure 3-8. The developer need not follow this sample, provided that the requirements listed below are included in the actual plan. Additional information required for development is discussed in other plans. For example, testing is discussed in the Software V&V Plan.

The reader is referred to IEEE Standard 1074 for information on life cycle processes.

1. Life Cycle Processes. Describe the life cycle that will be used on this project. Discuss the various processes that make up this life cycle. For each process, give the input information required in order to carry out the process, a description of the actions that must take place during the process, and the output information produced by the process. Since the output of one process is likely to be used as input to another, a data flow diagram would be appropriate. The processes suggested here are based on IEEE 1074.
    - — Requirements Processes.
        * Define, Develop, and Document Software Requirements.
        * Define and Document Interface Requirements.
        * Prioritize and Integrate Software Requirements.
        * Verify Requirements.
        * Perform and Document Requirements Safety Analysis.
    - — Design Processes.
        * Define and Document the Software Architecture. This includes how the software architecture will fit into the hardware architecture.

1. Introduction
   1.1. Purpose
   1.2. Scope
   1.3. Definitions and Acronyms
   1.4. References
2. Life Cycle Processes
   2.1. Requirements Processes
   2.2. Design Processes
   2.3. Implementation Processes
   2.4. Integration Processes
   2.5. Validation Processes
   2.6. Installation Processes
   2.7. Operation and Maintenance Processes
3. Methods, Tools and Techniques
   3.1. Requirements Methods, Tools and Techniques
   3.2. Design Methods, Tools and Techniques
   3.3. Implementation Methods, Tools and Techniques
   3.4. Integration Methods, Tools and Techniques
   3.5. Installation Methods, Tools and Techniques
4. Standards
5. Schedule and Milestones
6. Technical Documentation Requirements

Figure 3-8. Outline of a Software Development Plan

* Design and Document the Database.
* Design and Document Interfaces. This includes all interfaces between the software components, and between software, hardware, and instrumentation.
* Select or Develop and Document Algorithms.
* Perform and Document Detailed Design.
* Verify the Software Design.
* Perform and Document Design Safety Analysis.

— Implementation Processes.

* Create Unit Development Folders.
* Create Source Code.
* Create and Document Test Data.
* Generate Object Code.
* Perform and Document Unit Testing.
* Verify the Software Implementation.
* Perform and Document Implementation Safety Analysis.

— Integration Processes.

* Specify and Document System Build Methods.
* Integrate Hardware, Software and Instrumentation.
* Create and Document Integration Test Procedures and Test Cases.
* Perform Integration Testing.
* Verify the Software Integration.
* Perform and Document Integration Safety Analysis.

— Validation Processes.

* Specify and Document Validation Test Procedures and Test Cases.
* Perform and Document Validation Testing.
* Verify the Validation Testing.
* Perform and Document Validation Safety Analysis.

— Installation Processes.

* Specify and Document Installation Procedures.
* Specify and Document Installation Acceptance Procedures.
* Specify and Document Installation Test Procedures and Test Cases.
* Specify and Document Installation Configuration Tables.
* Verify the Installation Procedures.
* Perform and Document Installation Safety Analysis.

— Operation and Maintenance Processes.

* Specify and Document Operation Procedures.
* Specify and Document Regression Test Procedures and Test Cases. This will probably consist of the installation test procedures and test cases, augmented by tests to cover any faults found and repaired during operation.

2. Methods, Tools, and Techniques. Describe the methods and techniques that will be used to develop the software, and the tools that will be used in connection with those methods and techniques.

— Requirements. Describe the requirements methodology and any tools that will be used to implement it. A formal requirements methodology is recommended for reactor protection systems. Specify any requirements tracking tool that will be used.

— Design. Describe the design method and any tools that will be used to implement it. A formal design method is recommended for reactor protection systems. Specify any CASE tools that will be used. Specify what computers will be used to perform and document the software design.

— Implementation. Describe the implementation method and all tools that will be used in implementation. Specify what programming language will be used. Specify the compiler and linker that will be used. Specify what computers will be used for software development and unit testing.

— Integration. Describe the integration method and any tools that will be used to implement the integration. Note that integration validation procedures, methods and tools are described in the Software V&V Plan. This includes integration testing.

— Installation. Describe the installation method and any tools used to assist in installation. Note that installation validation procedures, methods, and tools are described in the Software V&V Plan. This includes installation testing.

3. Standards. List all international, national, and company standards that will be followed in the project.

4. Schedule and Milestones. List all of the technical milestones that must be met. Describe what is expected at each milestone.

5. Technical Documentation. List all of the technical documents that must be produced during the software development. Discuss milestones, baselines, reviews, authors and sign-offs for each document. This list may be all of or a subset of the list given in Section 3.1.1.3.

3.1.7. Software Integration Plan

Software integration actually consists of three major phases: integrating the various software modules together to form single programs, integrating the result of this with the hardware and instrumentation, and testing the resulting integrated product. During the first phase, the various object modules are combined to produce executable programs. These programs are then loaded in the second phase into test systems that are constructed to be as nearly identical as possible to the ultimate target systems, including computers, communications systems and instrumentation. The final phase consists of testing the results, and is discussed in another report (Barter 1993).

A sample table of contents for a Software Integration Plan is shown in Figure 3-9, based on IEC 880. The developer need not follow this sample, provided that the requirements listed below are included in his own plan. These requirements are listed next.

1. Integration Level. Multiple levels of integration may be necessary, depending on the complexity of the software system that is being developed. Several integration steps may be required at some levels.

1. Introduction
   - 1.1. Purpose
   - 1.2. Scope
   - 1.3. Definitions and Acronyms
   - 1.4. References
2. Identification of the Integration Process
   - 2.1. Integration Level
   - 2.2. Integration Objects and Strategies
3. Integration Marginal Conditions
   - 3.1. Integration and Testing Environment
   - 3.2. Priorities
   - 3.3. Risks
   - 3.4. Other Marginal Conditions
4. Organization of Integration
   - 4.1. Integration Network Plan
   - 4.2. Personnel and Responsibilities
5. Integration Procedures
   - 5.1. Required Products
   - 5.2. Integration Instructions
   - 5.3. Special Handling

Figure 3-9. Outline of a Software Integration Plan

- — Describe the different levels of integration and the scope of each integration step at each level.
- — Give a general description of the various objects that will be included in each step at each level.

2. Integration Objects and Strategies.
   - — Give a complete list of all objects, computer hardware, instrumentation, software, and data that will be included in each integration step.
   - — Describe the strategy that will be used for each integration step.
3. Integration and Testing Environment.
   - — Describe the environment that will be used to perform and test each integration step.
   - — List the tools that will be used in each integration step.
4. Integration Priorities. Allocate each integration step a priority, based on schedule, dependence along the integration products, risk, and any other factors deemed important to the development organization.
5. Integration Risks. Risk here refers primarily to budget and schedule. Other forms of risk can be considered, at the option of the development organization.
   - — If any integration step involves significant risk, describe the potential problems and the preventive measures that will be taken to avoid them.
6. Integration Network Plan.
   - — Order the integration steps into a time sequence. This order is determined primarily by the dependencies among the integration steps. Steps at more detailed levels will generally be required to complete successfully before a step at a more general level can be performed. Other factors can influence this order.
7. Integration Personnel and Responsibilities.
   - — List the personnel who will be involved in the integration steps.
   - — Provide a means to keep this list up to date.
8. Integration Products.
   - — List all of the products of each integration step.
9. Integration Instructions. Provide the technical instructions needed to carry out each integration step, as follows.
   - — List the inputs to the integration step.
   - — Describe the procedures for obtaining the input items (hardware, instrumentation, software, and data) for the step. This is expected to involve the CM organization.
   - — Describe the integration process for the step.
   - — List the outputs of the integration step.
   - — Discuss contingency strategies if the integration fails to complete.
   - — Describe the procedures for delivering the completed integration product to the configuration management organization.
   - — Describe the procedures for delivering the completed integration product to the V&V organization for integration testing.

3.1.8. Software Installation Plan

Software installation is the process of installing the finished software products in the production environment. The Installation Plan will describe the general procedures for installing the software product. For any particular installation, modifications, or additions may be required to account for local conditions.

A sample table of contents for a Software Installation Plan is shown in Figure 3-10. The developer need not follow this sample, provided that all of the installation requirements are included in the actual plan. Installation testing may be included in this plan, or in the V&V plan. The latter alternative is presumed here.

1. Installation Environment. Describe the environment within which the software product is expected to perform. This can include the reactor itself, the reactor protection system, and the protection system instrumentation and computer hardware. This description should be limited to those items required for successful installation and operation.

1. Introduction
   1.1. Purpose
   1.2. Scope
   1.3. Definitions and Acronyms
   1.4. References
2. Identification of the Installation Environment
   2.1. Application Environment
   2.2. Computer Hardware
   2.3. Instrumentation
3. Installation Package
   3.1. Installation Software
   3.2. Installation Documents
4. Installation Procedures

Figure 3-10. Outline of a Software Installation Plan

2. Installation Package. Describe all of the materials that will be included in the installation package. This will include the software products, the media that contain them, and associated documents. If alternatives are available, describe each. For example, several different installation media might be provided.

3. Installation Procedures. Describe completely the procedure for installing the software in the operational environment. This should be a step-by-step procedure, written for the anticipated customer. Anticipated error conditions should be described, with the appropriate recovery procedures.

3.1.9. Software Maintenance Plan

Software maintenance is the process of correcting faults in the software product that led to failures during operation. There is a related activity, sometimes termed "enhancement," which is the process of adding functionality to a software product. That is not considered here. Enhancement of a reactor protection system should repeat all of the development steps described in this report.

The software maintenance plan describes three primary activities: reporting of failures that were detected during operation, correction of the faults that caused those failures, and release of new versions of the software product. A sample table of contents for this plan is shown in Figure 3-11. The developer need not follow this sample, provided that all the necessary activities are included in his own plan.

The maintenance activity should include use of a configuration management system to track the failure reports, fault corrections, and new releases of code and documents.

1. Failure Reporting. A well-designed method must exist for collecting operational failures and making them known to the software maintenance organization. The essential points are that no failures be overlooked by the customer and that no failures be lost by the maintenance organization. All failures must be tracked by the maintenance organization. This includes software failures, misunderstandings on the part of operators, mistakes in documents, bad human factors, and anything else that causes the protection to fail or potentially to fail.

   — Failure detection includes all procedures by which the existence of the failure is recorded by the customer. This will generally include reporting forms.

   — Failure reporting includes the procedures used to inform the maintenance organization of the failures. It includes transmission of the failure reports from the customer to the maintenance

1. Introduction
   1.1. Purpose
   1.2. Scope
   1.3. Definitions and Acronyms
   1.4. References
2. Failure Reporting Procedures
   2.1. Failure Detection
   2.2. Failure Reporting
   2.3. Failure Tracking
3. Fault Correction Procedures
   3.1. Fault Detection
   3.2. Fault Repair
   3.3. Testing Correction
   3.4. Regression Testing
4. Re-release Procedures

Figure 3-11. Outline of a Software Maintenance Plan

organization, and entry of these reports into a failure tracking system. The latter should be under configuration control.

— Failure tracking consists of the procedures used to make sure that the failure is assigned to some person or group for analysis and fault detection. The failure tracking system should permit management to always know who is responsible for handling the failure and any causative faults.

2. Fault Correction. Every failure is caused by one or more faults. These may be an incorrect requirements specification, a design error, or a coding error. The fault must be found and corrected. This includes correction of any related documentation if that is necessary. The plan will describe the method to be used in finding and correcting the fault

— Fault detection includes all activities required to find the fault or faults that caused the failure.

— Fault repair consists of all activities and procedures involved in correcting the fault.

— Because the failure was not discovered during any of the previous test activities, the acceptance test will need to be modified to take the new failure into account.

— Regression testing is the process of testing the newly modified software product to make sure that no new faults have been placed into the software

3. Re-release. Procedures must be defined to create new versions of the software product, releasing these to the customers and ensuring that the newly corrected software product is correctly installed.

3.1.10. Software Training Plan

The training plan will describe the procedures that will be used to train the operators of the software system. In this case, reactor operators will need to be trained in use of the protection system software. It is also possible that training will be required for managers and for maintenance personnel.

The actual training requirements depend to a great extent on the actual software product, development organization, maintenance organization, and customer (utility). Consequently, no attempt is made here to outline the contents of the training plan.

3.2. Requirements Activities

Perhaps the most critical technical tasks in any software development project are those tasks that relate to the understanding and documentation of the software requirements. This is especially true for a project, such as a reactor protection system, in which safety is a vital concern. There are several risks if the software requirements are not fully documented. Some requirements may be omitted, others may be misunderstood, still others may be interpreted differently by different developers. Any of these can create a hazard in a safety-critical application.

Software requirements are concerned with what the software must do in the context of the entire application, and how the software will interact with the remainder of the application. These two aspects are captured here in the *Software Requirements Specification*. In actual usage, the aspects may be split into two documents, if so desired by the development organization. It is also possible to record the requirements specifications in some type of computer database, using a requirements tracking system or a CASE tool.

In the discussion here, four documents are described. Taken together, they span the documentation, analysis, and review of the various requirements activities. The development organization may choose to combine some of the documents, or to have additional documents. This is not of great importance as long as the information discussed below is included in some document. The documents are:

- Software Requirements Specification.
- Requirements Safety Analysis.
- V&V Requirements Analysis Report.
- CM Requirements Report.

The V&V and CM reports are described in detail in the V&V and CM plans, so are not discussed here.

3.2.1. Software Requirements Specification

The SRS is required for a safety-critical application, to make sure that all safety-related system requirements are made known to the software developers. These requirements come from the overall application system design, and reflect the requirements placed on the software by the application system. In a reactor protection system, this means that the protection

system design must be known and documented, and the demands that the system makes on the computer system must be known. A hazard analysis of the protection system should be available.

The discussion here is based on IEEE 830, *Software Requirements Specification*. Topics have been added that relate specifically to real-time safety-critical systems. A sample table of contents of an SRS is shown in Figure 3-12. The developer need not follow this outline, provided that the requirements for an SRS that are listed below are included. In particular, the developer is encouraged to use some form of automated (or semi-automated) requirements tracking system, or to use a CASE system.

1. Project Perspective. Describe the way in which the software system fits within the larger reactor protection system. This section can be thought of as an overview of the software project, showing how it fits within the larger protection system.
   - — Describe briefly the functions of each component of the protection system, to the extent required for the software developers to understand the context of the software requirements.
   - — Identify the principal external interfaces of the software system. A context diagram, showing the connections between the protection system and the software system, can be quite helpful.
   - — Describe the computer hardware and instrumentation that will be used, if this is known and imposed by outside authority. Be sure that no unnecessary constraints are imposed on the software design.
2. Project Functions. Describe briefly the functions of the various parts of the software system. Do not go into great detail here—that is reserved for Section 3 of the SRS. Instead, think of this section as a self-contained management overview.
   - — For the sake of clarity, the functions listed here should be organized so that they can be understood to the customer, assessor, and regulator.
   - — Block diagrams, data flow diagrams, finite state diagrams, and other graphical techniques can be helpful, but are not mandatory.
3. User Characteristics. Describe the general characteristics of the users of the software system. These will include the reactor operators and

1. Introduction
   - 1.1. Purpose
   - 1.2. Scope
   - 1.3. Definitions and Acronyms
   - 1.4. References
2. General Description
   - 2.1. Project Perspective
   - 2.2. Project Functions
   - 2.3. User Characteristics
   - 2.4. General Constraints
   - 2.5. Assumptions and Dependencies
   - 2.6. Impacts
     - 2.6.1. Equipment Impacts
     - 2.6.2. Software Impacts
     - 2.6.3. Organizational Impacts
     - 2.6.4. Operational Impacts
3. Specific Requirements
   - 3.1. Functional Requirements for Software Components
   - 3.2. Performance Requirements
     - 3.2.1. Modes of Operation
     - 3.2.2. Timing Requirements
     - 3.2.3. Flexibility Requirements
   - 3.3. Interface Requirements
     - 3.3.1. Operator Interfaces
     - 3.3.2. Instrumentation Interfaces
     - 3.3.3. Software Interfaces
     - 3.3.4. Hardware Interfaces
   - 3.4. Reliability Requirements
     - 3.4.1. Availability
     - 3.4.2. Reliability
     - 3.4.3. Safety.
     - 3.4.4. Maintainability
     - 3.4.5. Backup
     - 3.4.6. Recovery and Restart
     - 3.4.7. Software and Hardware Diagnostic Capabilities
     - 3.4 8. Fault Tolerance
   - 3.5. Design Constraints
     - 3.5.1. Standards Compliance
     - 3.5.2. Hardware Limitations
     - 3.5.3. Software Limitations
   - 3.6. Security Requirements
   - 3.7. Database Requirements
   - 3.8. Operations Requirements

Figure 3-12. Outline of a Software Requirements Specification

software maintainers, and may include reactor management personnel, depending on the specific requirements imposed by the overall protection system.

- For each class of user, give those characteristics that will be required in order to design the software system. This can include educational level, experience with nuclear reactors, experience with real-time computer systems, and general technical ability.

4. General Constraints. Give a general description of outside factors that will limit the designer's options. The following list is typical, not exhaustive:
   - Regulatory and other legal policies.
   - Hardware limitations.
   - Interfaces to other applications.
   - Audit functions.
   - Use of specific operating systems, compilers, programming languages, and database products.
   - Use of specific communications protocols.
   - Critical safety considerations.
   - Critical security considerations.
5. Assumptions and Dependencies. List each of the factors that can affect the requirements or design if they change. These are not design constraints, but more general assumptions. For example:
   - Business conditions.
   - Anticipated changes in laws or regulations.
   - Availability of specific hardware or software.
   - Possible changes in the computer industry.
6. Impacts. Provide warnings about anticipated impacts of the new software system. For a new reactor, this section of the SRS is probably not applicable. However, if an existing protection system is being replaced by a new computer-based protection system, impacts should be described. The purpose is to provide ample time to prepare for the new system.
   - Summarize any changes that will be required to existing equipment, new equipment that must be installed, and building modifications that may need to be made.
   - Discuss any additions or modifications that will be needed to existing applications and support software in order to adapt them to the new software system.
   - Describe any organizational impacts required by the new software system. This can include reorganizations, increase or decrease in staff levels, upgrade of staff skills, and changes to interfaces with regulators.
   - Summarize any operational changes that will be required, such as operational procedures, relationships between the reactor protection system and the reactor operators, staff procedures, data retention and retrieval procedures, reporting channels and methods, failure consequences, and recovery procedures and processing time requirements.
7. Functional Requirements. This large section of the SRS describes what the software system must do. The section can be subdivided in a manner that makes the functional requirements clear. For example, functions that relate to particular protection system activities could be grouped together, or functions that relate to particular protection system components.
   - Each functional requirement should be individually numbered, for reference by designers and assessors.
   - Each functional requirement should be stated in a single positive sentence. Additional sentences may be included if necessary to fully describe the requirement.
   - It is frequently useful to state what is not required, so that the designers do not provide unwanted functionality.
   - It is desirable to use a formal language to describe the functions. This can be the only language used, or English can be used as a supplement. The formal language may, for example, be a language such as Z, which is based on mathematical logic (Stepney 1987). An English description may be required in addition to the formal specification in order to ensure that the latter is comprehensible to developers, managers, users, and auditors.
8. General Interface Requirements. The Software Interface Specification should identify the reliability and safety requirements associated with each system interface. The list given next, from Redmill 1988, applies to all interfaces. Separate lists are given below for instrumentation and operator interfaces.

- — Functions provided and required by the system and its processes.
- — Sequences of interactions.
- — Input/output files for batch transfer to and from other systems.
- — Nature of the information transferred.
- — Definition of printed or microfilm outputs.
- — Formats.
- — Available options.
- — Timing.
- — Color.
- — Frequency of demands.
- — Accuracy.
- — Message error rates and types.
- — Electrical and mechanical interface requirements.
- — Input checking.
- — Presentation formats.
- — Information density.
- — Alerting signals.
- — Backup and error correction.
- — Security requirements.

9. Operator Interface Requirements. Define all of the requirements for communication between the software system and the operators.
   - — Be careful to define only the requirement, not the design. In particular, do not specify the appearance of operator screens unless that appearance is really a requirement.
   - — The following aspects of operator interfaces should be considered if they apply to the application (Redmill 1988):
     * Positioning and layout of controls and displays
     * Human reaction and decision times
     * Use of colors, bold face, underlining and blinking on displays
     * Menu techniques
     * Default values
     * Response times
     * Help facility
     * Comfort signals
10. Instrumentation Interface Requirements. In a real-time system, instrumentation can be divided into two classes: sensors and actuators.
    - — For each sensor, describe the values that may be received from the sensor. Give ranges, units, precision, error bounds (including whether the bounds are a function of range), meaning, calibration, and any other significant facts about the sensor. If an analog to digital converter is used, give its resolution.
    - — For each actuator, describe the values that may be sent to the actuator. Give ranges, units, digital to analog resolution, precision, meaning, calibration, and any other significant facts about the actuator. (Many actuators will be simple on/off actuators, and most of the foregoing list will not apply.)
11. Software Interface Requirements. If the software system will communicate with any other application software system, define all the interfaces between the systems. This communication may be in terms of subroutine calls, remote procedure calls, communication messages, or some other means. All such are referred to as "messages" here.
    - — For each message, describe the source and destination of the message, the message contents and format, the meaning of the message, expected return messages, transmission method and medium, error conditions, expected frequency and size, and a reasonable upper limit to frequency and size. An upper limit to frequency, for example, could be a frequency that is exceeded less than 0.01% of the time.
    - — Interactions between the application program and the operating system are not usually considered to be interfaces that must be defined here. There may be rare exceptions, however, in particular cases.
    - — The following aspects of communication system interfaces should be considered if they apply to the application (Redmill 1988):
      * Handshaking
      * Error checks
      * Input and output communication ports
      * Communication protocols and procedures
      * Interrupts
      * Exception handling and error recovery
      * Message formats
      * Message throughput

See Preckshot 1992a for more information on communication systems.

12. Hardware Interface Requirements. If the software system must communicate directly with the computer hardware, define all the interfaces between the software and hardware systems. As above, the term "message" is used.
    - For each message, describe the software and hardware elements of the message, the method of message transmittal, the reason such message is required, transmission medium, error conditions, and expected and upper limit to frequency and size of the message.
13. Performance Requirements. Specify both static and dynamic numerical performance requirements.
    - Static requirements include the number of terminals to support and number of simultaneous users to support.
    - Dynamic requirements include numbers of transactions per unit of time, task switching time, amount of data to be processed in certain time periods, and amount of time that may have elapsed between a signal being present at a particular sensor and a resulting signal arriving at an actuator.
    - Modes of operation refer to the basic configurations in which the system must operate and in which the system would exhibit differing performance characteristics. Examples include: fully operational test and the partial loss of a particular protection system component.
    - Timing requirements must be specified numerically, and are usually given in terms of response time. This is the elapsed time between the receipt of an input signal from an operator or sensor and the arrival of a resulting output signal to an operator or actuator.
    - Timing requirements may depend on the mode of operation. If so, individual timing requirements for each mode of operation must be given.
    - Describe the required adaptability (flexibility) of the software system. This can include the ability to switch between modes of operation, operating environments, planned changes or improvements, and the use of new hardware. The intent is to provide information to the designers as to where flexibility will be important so as to minimize future design changes.
14. Reliability Requirements. A number of topics relating to the general subject of reliability are collected together here.
    - Availability is the percent of time that the software system will be available for use. It should be stated numerically. Different availability requirements may exist for the different modes of operation. State any assumptions that are used in deriving the availability number.
    - Reliability is a measure of the length of time the software system can be expected to operate before failing, under the assumption that the computer hardware does not fail. The usual measure is mean time to fail, or failure rate. Different reliability requirements may exist for the different modes of operation. State any assumptions that are used in deriving the reliability numbers.
    - Safety (in this section) is a measure of the length of time the software system can be expected to run before failing in such a way as to cause a system hazard. The usual measure is the probability that the system will fail catastrophically per demand. Different safety requirements may exist for the different modes of operation. See also the requirements discussed for a software safety plan, Section 3.1.5, and the later discussions on safety analyses in Sections 3.2.2, 3.3.3, 3.4.2, 3.5.2, 3.6.1, 3.7.5, and 3.8.1.
    - Maintainability is a measure of the length of time it will take to restore a failed application to usable status. This may be merely a restart of the software system; the measure does not usually include the amount of time required to identify and fix bugs. The usual measure is mean time to repair. In this context, the requirement is the amount of time the system is permitted to be not working before it is restored to operation, and may differ for the different modes of operation.
    - Specify any backup, recovery, and restart requirements. Any special requirements, such as reinitialization or reconfiguration, should be described.
    - If the application requires the ability to diagnose hardware or software failures,

identify these requirements. Acceptable responses to undesired events should be characterized. Undesired events, such as hardware failure and operator errors, should be described and the required software system responses should be stated.

— If the application must not only detect failures, but also recover from them and continue to operate, the fault tolerance requirements should be given. Note that no system can recover from all errors.

— Identify the reliability and safety requirements associated with each mode of operation. The following list identifies potential modes of operation that should be considered (Redmill 1988); not all will necessarily apply to reactors:

  * System generation.
  * Installation of new software.
  * Normal operation. There may be several different normal modes.
  * Degraded operation.
  * Emergency operation.
  * Automatic operation.
  * Semi-automatic operation.
  * Manual operation.
  * Periodic processing (such as daily or hourly).
  * Maintenance.
  * Housekeeping.
  * Shutdown.
  * Error isolation.
  * Error recovery.
  * Backup.

— All aspects of fault, error and failure handling should be covered (Redmill 1988):

  * Hardware/software failure.
    + Failure detection.
    + Failure identification.
    + Failure isolation.
    + Remedial procedures.
    + Redundancy.
    + Fail-safe operation.
  * System recovery procedures.
    + Fallback procedures.
    + Reconstruction of obliterated or incorrectly altered data.
    + Repair of redundant channels.
  * Information collection.
    + Separate sources.
    + Separate channels.
  * Information processing.
    + Weak signal levels.
    + Data storage.
    + Analysis techniques (correlation).
    + Noise reduction.
    + Error handling.
  * Human errors or violations.
    + Security standards.
    + Control aspects.
    + Procedural aspects.
    + Management aspects.

15. Design Constraints. List all guides, standards, and regulations that apply to the design.

    — List any laws and regulations, company policies, departmental standards, and national and international standards that may affect the design of the software system.

    — If the application must run on certain hardware configurations, or in conjunction with particular software products, list them here.

16. Security Requirements. Specify the requirements for protecting the software system from accidental or malicious access, use, modification, destruction, or disclosure.

    — This may include access restrictions, cryptographic techniques, the use of history logs, restrictions on the use of terminals or communications lines, physical isolation, and similar matters.

17. Database Requirements. Specify the requirements for any database that is to be developed as part of the software system. Be careful not to overspecify; list only items for which there is an identified external requirement.

18. Operational Requirements. Specify the normal and special operations required by the user. This can include the different modes of operation listed earlier.

3.2.2. Requirements Safety Analysis

A safety analysis should be performed for any real-time system for which safety is a consideration. The

purpose of the analysis is to identify any errors or deficiencies that could contribute to a hazard and to identify system safety considerations that are not addressed in the software requirements specification. Four analyses are recommended here; additional analyses may be required, depending on the nature and sensitivity of the application. The safety analysis and the reporting of the results of the analysis could be included in the project's V&V activities, or within the system safety activities. The recommended analyses are as follows:

- Criticality Analysis determines which software requirements are safety critical.
- System Analysis determines that all system requirements have been correctly allocated.
- Specification Analysis determines that each software safety function and requirement is correctly and consistently implemented with the other functions and requirements in the requirements documentation.
- Timing and Sizing Analysis determines that there will be sufficient resources to accomplish safety-critical requirements.

One viewpoint of requirements safety analysis is given in Lee 1992. This is not the only possible approach. The discussion here is based on that document, and is limited to the objectives of the various analyses.

Criticality Analysis will identify those software requirements that have safety implications so that analysis efforts can be focused on the most critical requirements. The analysis must consider each software requirement and determine its potential impact on safety. For a reactor protection system, requirements will be classified as critical or non-critical, depending on their impact on safety. Requirements that are determined to be critical must be delineated in some way to be sure that they receive sufficient attention during software design, verification and validation, testing, and later safety analyses.

System Analysis matches the software requirements to the overall system requirements. Software is a component of a system that is made up of hardware, instrumentation, and other software. The objective of system analysis is to ensure that the software component requirements are complete and traceable to the system level requirements and are complementary to the requirements for other parts of the protection system. This analysis ensures that the software component requirements include all necessary functions and constraints and no unintended ones. This same criteria should be applied to the software component interface specifications. The result of the system analysis is an identification of incomplete, missing, and incorrectly allocated requirements.

Specification Analysis evaluates the completeness, correctness, consistency, and testability of software safety requirements. Well-defined requirements are a strong standard by which to evaluate a software component. Specification analysis should evaluate each requirement singularly and all requirements as a set. Among the techniques used to perform specification analysis are hierarchy analysis, control flow analysis, information flow analysis, and functional simulation. Hierarchy analysis identifies missing and incomplete requirements and requirements for functions that are never used. Control flow analysis examines the order in which software functions will be performed and identifies missing and inconsistently specified functions. Information flow analysis examines the relationship between functions and data to identify incorrect, missing, and inconsistent input/output specifications. Functional simulation models the characteristics of a software component to predict performance, check human understanding of system characteristics, and assess feasibility.

Timing and Sizing Analysis evaluates software requirements that relate to execution time and memory allocation. Timing and sizing analysis focuses on program constraints. The types of constraint requirements are maximum execution time, time to execute critical functions, and maximum memory usage. The analysis should evaluate the adequacy and feasibility of timing and sizing requirements, particularly under stress conditions.

3.3. Design Activities

The next logical activity after the software requirements have been determined is to design the software system. The life cycle being used by the development organization will determine whether or not all requirements activities are really completed before the design is begun. However, there are certain activities that can be considered design, as opposed to requirements and implementation, no matter what life cycle is used. These activities are the topic of this section.

In the discussion here, five documents are described. Together, they span the documentation, analysis, and review of the various software design activities. The

development organization may choose to combine some of the documents, or to have additional documents. This is not of great importance as long as the information discussed below is included in some document. The documents are:

- Hardware and Software Architecture.
- Software Design Specification.
- Software Design Safety Analysis.
- V&V Design Analysis Report.
- CM Design Report.

The V&V and CM reports are described in the V&V and CM plans, so are not discussed here.

3.3.1. Hardware and Software Architecture

The design architecture consists of a description of the hardware and software elements, and the mapping of the software into the hardware. The hardware architecture will show the various hardware items—computers, file systems, I/O devices (such as sensors, actuators, and terminals) and communication networks. This provides a physical view of where computation can take place, how information is physically stored and displayed, and how information is physically routed from one device to another.

The software architecture will show the various software processes, databases, files, messages, and screen designs. This architecture shows the logical view of where computations take place, where information is logically stored and displayed, and how information moves logically from one process, data store, or input device to another process, data store, or output device.

Finally, the mapping shows the connections between the software and the hardware. Each process must run on a computer, each data store must reside on one or more physical storage media, each screen display must be shown on a real terminal, and information must move over physical communication lines in order to carry out logical communication between processes and I/O devices.

The architecture description will probably consist of a set of drawings, possibly within a CASE tool. For this reason, no suggestion is made here for a document table of contents.

3.3.2. Software Design Specification

The software design specification shows exactly how the software requirements will be implemented in software modules and processes. The term "module" is used here to mean a collection of programming language statements, possibly including subroutines, which is compiled as a unit. An (executable) load module is a collection of one or more modules, including both application modules and operating system library modules, that is capable of independent execution in a computer. The term "process" is used to mean the execution of a load module on a computer. Multiple copies of a load module might be running on the same or different computers at the same time, each as an individual identifiable process.

Many different design techniques are available. While a formal design technique is preferred for reactor protection systems, other alternatives do exist. Some are described in Pressman 1987. One technique for real-time systems is given in Hatley 1987. Object-oriented design techniques are popular in some quarters (Rumbaugh 1991). An automated or semi-automated design system is recommended; a CASE tool is an example. It is best if this design system is directly linked to the requirements tracking system.

Because of the many design techniques that might be used, plus the desire to use automated design systems, no attempt is made here to describe a table of contents for a design document. Certain requirements can be placed on the design; these are listed next. This list is derived from IEEE Standard 1016, *Software Design Descriptions*.

1. Whatever technique is used, the software design should result in a hierarchical decomposition into layers of design elements. IEEE 1016 defines a design element as a component "that is structurally and functionally distinct from other elements and that is separately named and referenced."
2. A design element may be a software system, subsystem, or module; database, file, data structure, or other data store; message; program or process.
3. Each design element should be placed in the configuration management system as a configuration item.
4. Each design element must have a number of attributes. The following list is meant to be representative. The designer may have additional

attributes. The description of the attributes should be as clear as possible. This may, for example, imply that attributes for many elements are described together. For example, an entity-relationship diagram may be used to describe the structure of several design elements at once. A state-transition diagram can be used to explain the interactions of several process design elements. The intent is to include all necessary design information in a way that can be implemented correctly and understood by the assessor.

- The name of the design element.
- The type of the design element, which could include the system, subsystem, module, database, file, data structure, screen display, message, program, and process.
- The purpose of the design element. That is, why the element exists in the design.
- The function of the design element. That is, what it does.
- If the element has subordinate elements in the hierarchical decomposition, list them. A diagram can be quite helpful for this purpose.
- List all elements with which this element interacts. This could be done using an entity-relationship diagram (for databases and data structures), data flow diagram (for processes), finite state diagram (for understanding control), structure chart (for a program), a transaction diagram (for messages), or whatever best shows the interactions. The interaction description should include timing, triggering events, order of execution, data sharing, and any other factor that affects interaction.
- Provide detailed descriptions of the ways in which the design element interacts with other design elements. This description includes both the methods of interaction and the rules governing the interactions. Methods include the mechanisms for invoking the design element, input and output parameters or messages, and shared data. Rules include communications protocols, data format, range of values of parameters, and the meaning of each parameter or message.
- Describe all resources required by the design element for it to perform its function. This can include CPUs, printers, terminals, disk space, communication lines, math libraries, operating system services, memory, and buffer space.
- If the element does any processing, describe the method by which the element carries out its function. This can include a description of algorithms, sequencing of events or processes, triggering events for process initiation, priority of events, actual process steps, process termination conditions, contingency processing, and error handling. A formal description of the design processing is strongly encouraged. Diagrams are encouraged; this can include data flow diagrams, control flow diagrams, Warnier-Orr diagrams, Jackson diagrams, decision tables, finite state diagrams, and others. See Pressman 1987.
- If the element is a data store, message, or screen display, describe its structure. This can include the method of representation, initial values, permitted ranges of values, use, semantics, format, and appearance. The description should discuss whether the element is static or dynamic, whether it is shared by several processes or transactions, whether it is used for control or value, and how it is validated. A formal description of the structure of the design element is strongly encouraged. An entity-relationship diagram can be helpful to understanding.
- Include a list of all requirements implemented by this element. This is used to verify that the design implements the requirements, and only the requirements. Some elements may implement more than one requirements, while some requirements may need several elements for a successful implementation. A cross reference table can be used here.
- List all implementation concerns that are determined as part of the design. This can include programming language, use of operating system privileges, hardware requirements, error lists, error code lists, standards and regulations that must be followed, accuracy, performance considerations, and reliability considerations.
- List all hazards that may be affected by this design element, or that may affect the way in which the element is implemented.

3.3.3. Software Design Safety Analysis

A safety analysis should be performed on the software design of any computer-controlled reactor protection system. The purpose of the analysis is to verify that the design correctly and consistently incorporates the system safety requirements and identifies safety-critical software design elements and detects errors that might result in violations of the system safety requirements. Four new analyses are recommended here, and one requirement safety analysis should be reviewed. Additional analyses may be required by the developer or the assessor, depending on the nature and sensitivity of the application. The results of the design safety analysis should be documented. The recommended analyses are as follows:

- Design Logic Analysis determines whether the software design equations, algorithms, and control logic correctly implement the safety requirements.
- Design Data Analysis determines whether the data-related design elements are consistent with the software requirements and do not violate system safety requirements.
- Design Interface Analysis determines that the interfaces among the design elements have been properly designed, and do not create a safety hazard.
- Design Constraint Analysis evaluates any restrictions imposed on the software requirements by real-world limitations and the design of the software system, and determines that no new safety hazards have been created.
- The Timing and Sizing Analysis performed as part of the Requirements Safety Analysis (Section 3.2.2) should be reviewed. If the results of that analysis have changed due to the completion of the software design, the analysis should be revised. New information on timing and sizing generally becomes available during the design activities, and may change previous conclusions.

Design Logic Analysis evaluates the equations, algorithms, and control logic of the software design. Logic analysis examines the safety-critical areas of each software module. This is done by determining whether the module implements any of the safety-critical requirements. The interaction between safety critical and non-safety critical components should be identified. Components that generate outputs used by critical components should also be considered critical. The control logic used to invoke the program tasks should be considered critical.

The design must be traceable to the requirements. The analysis should ensure that all safety critical requirements have been included in the design. It should also ensure that no new design features are developed that have no basis in the requirements.

During logic analysis, the design descriptions, control flows, and detail design are analyzed to ensure they completely and correctly implement the requirements. Special emphasis should be placed on logic to handle error conditions and high data rate conditions. The analysis should identify any conditions that would prevent safety-critical processing from being accomplished.

Design Data Analysis evaluates the description and intended use of each data item in the software. Data analysis ensures that the structure and intended usage of data will not violate a safety requirement. The analysis ensures that all critical data item definitions are consistent with the software requirements. This includes the range and accuracy of the data item, and the insertion of the proper values into constants. The analysis should determine that the precision of intermediate variables is sufficient to support the final output accuracy requirements. The use of each data item in the design logic is also evaluated. This includes the order of use and the correct use. The usage must be consistent with the structure, the accuracy, and the range of the data. The analysis should also focus on unintentional use or setting of the safety-critical data by non-safety-critical logic.

Design Interface Analysis verifies the proper design of a safety-critical software component's interfaces with other components of the system. This includes other software components and interfacing software programs, and hardware components. This analysis will verify that the software component's interfaces have been properly designed and do not introduce a hazard. Design interface analysis will verify that control and data linkages between interfacing components have been properly designed. This includes evaluation of the description of parameters passed, data items passed and returned, direction of control, and the form of data. The definition and typing of parameters must be consistent and compatible. The evaluation includes the interfaces for safety-critical components to both critical and non-safety-critical components.

Design Constraint Analysis evaluates restrictions imposed by requirements, by real-world limitations, and by the design solution. The design materials describe any known or anticipated restrictions on the software components. These restrictions may include timing and sizing constraints, equation and algorithm limitations, input and output data limitations, and design solution limitations. Design constraint analysis evaluates the feasibility of the safety-critical software based on these constraints.

The design safety analysis should also identify any additional risks that may arise due to the use of particular tools, methods, programming languages, or design approaches. For example, errors in compilers can create new and unexpected hazards.

3.4. Implementation Activities

Implementation consists of the translation of the software design into actual code. This code will exist in some form, such as a programming language, a database an implementation language, or a screen design language. Many such languages exist, varying from assembler (second generation) languages through procedure-oriented programming (third generation) languages to high-level block (fourth generation) languages.

The discussion here calls for four documents, where the code listings are considered a single document. Taken together, they cover the documentation, analysis and review of the various software implementation activities. The development organization may choose to have additional documents.

- Code Listings.
- Code Safety Analysis.
- V&V Implementation Analysis and Test Report.
- CM Implementation Analysis.

The V&V and CM reports and described in the V&V and CM plans, so are not discussed here. There is little to say about code listings, other than they must exist, so that is not discussed either.

3.4.1. Code Safety Analysis

A safety analysis should be performed on the actual software that is developed for any computer-controlled reactor protection system. The purpose of the analysis is to verify that the implementation correctly and consistently incorporates the system safety requirements, identifies safety-critical software modules and data structures, and detects errors that might result in violations of the system safety requirements. Four new analyses are recommended here, and one requirement safety analysis should be reviewed; additional analyses may be required by the developer or the assessor, depending on the nature and sensitivity of the application. The results of the design safety analysis should be documented. The recommended analyses are as follows:

- Code Logic Analysis determines whether the software correctly implements the software design.
- Code Data Analysis determines whether the data structures correctly implement the data structure design.
- Code Interface Analysis verifies the compatibility of internal and external interfaces between software components and other system components.
- Code Constraint Analysis ensures that the program operates within the constraints imposed by the requirements, the design and the target computer system.
- The Timing and Sizing Analysis performed as part of the design safety analysis should be reviewed. If the results of that analysis have changed due to the completion of the software implementation, the analysis should be revised. New information on timing and sizing generally becomes available during the implementation activities, and may change previous conclusions.

Code Logic Analysis evaluates the sequence of operations represented by the coded program. The logic analysis will detect logic errors in the coded software. This analysis evaluates the branching, looping, and interrupt processing of the software components. The analysis also should ensure that code that has no basis in the design is not implemented. Logic reconstruction entails the preparation of flowcharts or other graphical representations from the code and comparing them to the design material descriptions. This analysis verifies the consistency and correctness of the code with respect to the detailed design. As part of this process, equations are reconstructed and compared to the requirements and design.

Code Data Analysis concentrates on data structure and usage in the coded software. Data analysis focuses on how data items are defined and organized to be sure

the design is correctly implemented. The data analysis compares the usage and value of all data items in the code with the descriptions provided in the design materials to ensure consistency with the design. This analysis verifies the correct type has been used for each data item, such as floating point, integer, or array. This analysis will ensure that data structures are not used in such a way as to create a potential hazard. Special attention is applied to accessing arrays to ensure that code will not access arrays outside their bounds and destroy safety-critical data.

Code Interface Analysis verifies the compatibility of internal and external interfaces of a software component. Interface analysis is designed to verify that the interfaces have been implemented properly. The analysis will ensure that the interfaces are consistent and do not create a potential hazard. At least four types of interfaces are evaluated: subroutine calls to other software components, parameters passed through common or global data, messages passed through communication systems, and external hardware interfaces.

Code Constraint Analysis ensures that the program operates within the constraints imposed on it by the requirements, the design, and the target computer system. The constraints imposed include physical, mathematical, accuracy, speed, and size.

The code safety analysis should also identify any additional risks that may arise due to the use of particular tools, methods, programming languages or implementation approaches. This is in addition to the similar analysis performed as part of the design safety analysis.

3.5. Integration Activities

Integration consists of the activities that are required in order to combine the various software programs and hardware items into a single system. The various hardware modules must be assembled and wired together according to the hardware design specifications. The various software modules are linked together to form executable programs. The software is then loaded into the hardware. Finally, the entire combination is tested to be sure that all internal and external interface specifications have been satisfied. and that the software will actually operate on that particular hardware.

The integration activities are governed by the Integration Plan, discussed above in Section 3.1.7. Integration testing is described in a separate testing report (Barter 1993), and follows the Software Verification and Validation Plan described in Section 3.1.4. The Integration Safety Analysis is carried out according to the Software Safety Plan described in Section 3.1.5.

3.5.1. System Build Documents

The Integration Plan describes the various steps that will be carried out during the integration process. One of these is the actual construction of the software programs from modules and libraries. The exact procedure for doing this is documented in the System Build Specification. There will be one such specification for each program that must be created. The developer may have a separate build document for each program, or combine the specifications into a single document.

The System Build Specification provides the exact steps taken to build the program. This includes names of modules and files, names of libraries, and job control language used to build the program. This specification must be in sufficient detail to permit the build to be carried out without ambiguity.

No attempt is made here to provide an outline for the System Build Specification. It must be tailored to the particular development process being used, the nature of the operating system and programming language being used, and the nature of the hardware upon which the program will run.

3.5.2. Integration Safety Analysis

The Integration Safety Analysis will ensure that no hazards have been introduced during the integration activities. The method of doing this is not specified here. It is the responsibility of the developer, the V&V organization, and the software safety organization to make sure that all safety concerns have been addressed during the integration process.

3.6. Validation Activities

Validation is the set of activities that ensure that the protection system, as actually implemented, satisfied the original externally-imposed requirements. In particular, it is necessary to guarantee that the system safety requirements are all met. Validation consists of a mixture of inspections, analyses and tests. The inspection and test aspects are discussed in Barter 1993. Safety analysis is described here.

Validation is carried out according to the Software Verification and Validation Plan described in Section 3.1.4. The Validation Safety Analysis is carried out according to the Software Safety Plan described in Section 3.1.5.

3.6.1. Validation Safety Analysis

The Validation Safety Analysis will examine the entire system and the process of developing the system, to ensure that all system safety considerations have been correctly implemented in the protection system and that no new system hazards have been created due to any actions of the protection system. This analysis will review all previous analyses and ensure that no actions have taken place to invalidate their results.

The method of performing this analysis is not specified here. It is the responsibility of the software safety organization, possibly with the assistance of the V&V organization and the system safety organization, to ensure that the Validation Safety Analysis is properly carried out.

3.7. Installation Activities

Installation is the process of moving the complete system from the developer's site to the operational site. The nature of reactor construction is such that there may be considerable time delays between the completion of the protection computer system by the developer and the installation of that system in an actual reactor. The documents discussed here should provide sufficient information to permit the installation to take place correctly, and for the protection system to operate correctly.

Installation is carried out according to the Software Installation Plan described in Section 3.1.8. The Installation Safety Analysis is carried out according to the Software Safety Plan described in Section 3.1.5.

3.7.1. Operations Manual

The Operations Manual provides all of the information necessary for the correct operation of the reactor protection system. This includes normal operation, off-normal operation, and emergency operation. Start-up and shut-down of the computer system should be discussed. All communications between the computer system and the operator should be described, including the time sequencing of any extended conversations. All error messages should be listed, together with their meaning and corrective action by the operator.

The Operations Manual structure is dependent on the actual characteristics of the particular computer system. No suggestion is given here as to a possible table of contents.

3.7.2. Installation Configuration Tables

Real-time systems frequently require tables of information that tailor the system to the operational environment. These tables indicate I/O channel numbers, sensor and actuator connections and names, and other installation-specific quantities. If this particular protection system requires such a table, the developer should prepare a document that describes all the configuration information that must be provided and how the system is to be informed of the configuration information. The actual configuration tables should created as part of the installation activity. They should be fully documented.

3.7.3. Training Manuals

An operator training program should be required so that the operators may learn the correct way to use the protection system. The Training Manual is an important part of the training program. It may be provided by the system developer or the customer (utility).

No further information on training is provided here.

3.7.4. Maintenance Manuals

The Maintenance Manual will describe the procedures to be followed when the operational software must be changed. The manual may be prepared by the development organization or by the maintenance organization. The manual will completely describe all of the steps that must be carried out to change the program, validate the changes, prepare new releases, install the new releases, and validate the installation.

3.7.5. Installation Safety Analysis

Once the computer system has been installed in the operational setting, a final safety analysis will be performed. This will verify that all system safety requirements are implemented in the installation, that no safety-related errors have occurred during installation, and that no hazards have been introduced during installation.

3.8. Operations and Maintenance Activities—Change Safety Analysis

Changes may be categorized in three separate areas: software requirement changes; implementation changes to change the design and code to be compliant with software or safety requirements; and constraint changes, such as changes in equipment, assumptions, or operating procedures.

When software requirement changes are recommended, the safety activity should analyze those changes for safety impact. As in requirements analysis, the analyst should identify safety-critical requirements and determine their criticality. The analyst should also identify any safety impacts on system operation, operating procedures, and the safety analysis activity. Impacts to the safety activity include the ability to verify or test the change. The analysis should also ensure that the change does not make any existing hazards more severe. Once the requirements change has been approved, the safety activity should analyze and test the changes using the methods defined in the previous sections of this document.

Implementation changes to design or code are analyzed to identify any safety-critical software components that are changed and to ensure that only the required components are changed. Changes to non-critical code should be analyzed to ensure they do not affect safety-critical software. The design and code should be analyzed and tested using the methods defined in the previous sections of this document.

For constraint changes, the analysis must evaluate the impact on software safety. The operational changes are evaluated for changes to operator interfaces or additional administrative procedures that may result in a hazard. The change may also affect planned safety test procedures. Hardware changes are evaluated for new fault paths that may be introduced or for deletion of required interlocks. All changes to assumptions should be evaluated for their impact. The safety activity may recommend additional software requirements or design changes based on this analysis.

A safety change database should be developed to track the status of all changes. The database should include a tracking number for each change, the level of software affected (e.g., requirements, design, or code), the identification and version of the affected component, safety impact (e.g., none, high, medium, or low), the development status of the change (e.g., requirements, design, code, or test), and the safety analysis and approval status.

4. RECOMMENDATIONS, GUIDELINES, AND ASSESSMENT

This section is directed especially at the internal or external assessor. The life cycle tasks described in Section 3 are revisited, the reliability and safety risks associated with each task are discussed, recommendations for preventing and decreasing these risks are explored, and guidelines for using best engineering judgment to implement the tasks are described. Finally, this section presents a list of questions that can be used by the assessor when assessing the work products, processes, and the development organization's ability to produce a safe reactor protection system.

A recommendation is a suggestion that is important to the safety of the system. A guideline is a good engineering practice that should be followed to improve the overall quality, reliability, or comprehensibility of the system. An assessment question suggests factors that an assessor should investigate in order to verify the acceptability of the task solution.

The assessor may wish to ask additional questions; nothing written here is meant to imply that assessors should be restricted to the questions listed here. The questions are generally phrased in such a way that an affirmative answer is the preferred answer; a negative answer may be acceptable, but requires justification. Many questions ask about the existence of some item; these should be read as containing an implied question that the underlying concept is satisfactory to the assessor. For example, the question "Are general report formats known?" should be read as implying that the formats are sufficient to provide the information needed.

4.1. Planning Activities

Planning activities are basic to the entire development effort. There will be at least one plan; the question is how many plan or plans will there be, who will creates the plan(s), and who follows the plan(s). If the project management team does not create the plans, or at least oversee and coordinate their creation, someone else will. In the worst case, each member of the development team acts according to the developer's own plan. Different team members will be "marching to different drummers." Such chaotic activity is generally not conducive to safety.

4.1.1. Software Project Management Plan

The Software Project Management Plan (SPMP) is the basic governing document for the entire development effort. Project oversight, control, reporting, review, and assessment are all carried out within the scope of this plan.

Without an SPMP, the probability is high that some safety concerns will be overlooked at some point in the project development period, that misassignment of resources will cause safety concerns to be ignored as deadlines approach and funds expire, and that testing will be inadequate. Confusion among project development team members can lead to a confused, complex, inconsistent software product whose safety cannot be assured.

4.1.1.1. Recommendation

A safety-related software project should have an SPMP. The size, scope, and contents of this plan should be appropriate to the size, complexity, and safety-critical nature of the project. Detailed requirements for a SPMP are provided in Section 3.1.1. The plan should be under configuration control.

4.1.1.2. Guideline

The SPMP may be organized according to IEEE Standard 1058, as shown in Figure 3-2.

4.1.1.3. Assessment Questions

1. Process Model Questions.
   a. Is the timing of project milestones realistic?
   b. Is there sufficient time between milestones to accomplish the needed work?
   c. Is sufficient time allotted for review and audit?
   d. Is there time to integrate the software into the complete protection computer system, and to integrate that into the reactor protection system?
   e. Is there time to recover from unanticipated problems?
   f. Are project work products and deliverables well defined?
   g. Is it known who will be responsible for each product and deliverable?

   h. Do adequate resources exist to produce the products and deliverables?
2. Organizational Structure Questions.
   a. Is the project organization structure well defined?
   b. Are responsibilities known and documented?
   c. Does a management structure exist to keep the SPMP up to date?
   d. Is the SPMP under configuration control?
3. Organizational Boundary and Interface Questions.
   a. Are the boundaries of the development organization well defined?
   b. Are reporting channels clear?
   c. Does a formal communication channel exist between the software development organization and the regulator or assessor?
4. Project Responsibility Questions.
   a. Does the SPMP state that safety is the primary concern, over budget and schedule?
   b. Do management mechanisms exist to enforce this?
5. Project Priorities Questions.
   a. Does the SPMP require that safety is the top priority, over budget and schedule?
   b. Does a mechanism exist for ensuring this?
6. Assumptions, Dependencies, and Constraint Questions.
   a. Are the assumptions that may have an impact on safety documented in the SPMP?
   b. Are external events upon which the project depends documented?
   c. Are project constraints that may have an impact on safety identified and documented in the SPMP?
7. Risk Management Questions.
   a. Are known risk factors identified?
   b. Is the potential impact of each risk factor on safety described?
   c. Does a method exist for managing each risk that may impact safety?
8. Monitoring and Controlling Mechanism Questions.
   a. Are required reports identified?
   b. Are general formats known?
   c. Do the formats provide the information required by the recipient of the report?
9. Staffing Questions.
   a. Are necessary special skill needs identified?
   b. Do management mechanisms exist in the SPMP for obtaining people with the required skills in a timely manner?
   c. Are training requirements known and documented?
10. Technical Methods, Tools, and Techniques Questions.
   a. Are the development computer systems identified?
   b. Do these systems exist?
   c. Do they have sufficient resources for the development work?
   d. Are the development methods identified?
   e. Are they few in number?
   f. Are they sufficiently formal to permit correct specification and implementation of the software system?
11. Software Documentation Questions.
   a. Are required technical documents identified?
   b. Are production dates given?
   c. Are these realistic?
   d. Are internal review and audit processes identified?
   e. Are sufficient time and other resources allocated to perform the reviews and audits?
   f. Is a specific person identified as responsible for each document?
12. Project Support Function Questions.
   a. Are the referenced support functions defined in other documents, or defined here? (In the latter case, see the relevant assessment checklists below.)
13. Follow-Through Questions.
   a. Does evidence exist at each audit that the SPMP is being followed?

4.1.2. Software Quality Assurance Plan

Software quality assurance (SQA) is the process by which the overall quality of the software products is assessed.

Many aspects of software quality are described in the various Plans recommended in this report. This includes the Configuration Management Plan, the Software Safety Plan, the Software Verification and Validation Plan, and others. Without a single Software Quality Assurance Plan governing these various

individual plans, it is possible that the various individual plans many not be mutually consistent, and that some aspects of software quality that are important to safety may be overlooked.

4.1.2.1. Recommendation

A safety-related software project should have a Software Quality Assurance Plan. The size, scope, and contents of this plan should be appropriate to the size and complexity of the software system and the risk that can arise if the software system fails. Detailed requirements for a Software Quality Assurance Plan are given above, in Sections 3.1.2. The plan should be under configuration control.

4.1.2.2. Guideline

The SQA Plan (SQAP) may be organized according to IEEE Standard 730.1 as shown in Figure 3-3.

4.1.2.3. Assessment Questions

Many of the assessment questions that relate to the SQAP are given later, in the sections that discuss assessment of the other plans. In particular, see Section 4.1.4.

1. General Questions.
   a. Does the SQAP specify which software products are covered by the Plan?
   b. Does the SQAP explain why it was written? That is, what need does the SQAP satisfy?
   c. Does the SQAP explain the standard that was used to create the SQAP?
2. Management Questions.
   a. Is each project element that interacts with the SQA organization listed?
   b. Is the SQA organization independent of the development organization? If not, is each dependency clearly justified?
   c. Are the life cycle development phases that will be subject to SQA oversight listed?
   d. Are required SQA tasks listed and described?
   e. Is the relationship between the SQAP and other assurance plans described? Does a method exist for delineating overlapping responsibilities? Other plans include, but are not limited to, the Configuration Management Plan, the Software Safety Plan, and the V&V Plan.
   f. Is the relationship between the SQA organization and other assurance organizations described? Other organizations include, but are not necessarily limited to, the CM organization, the Safety organization and the V&V organization.
   g. Is the person responsible for the SQAP identified, by name and position?
   h. Is the person responsible for overall software quality assurance identified, by name and position?
   i. Does the plan explain how conflicts between the SQA organization and the development organization will be resolved?
3. Document Questions.
   a. Are required software documents listed?
   b. Is it known how each document will be reviewed by the SQA organization for adequacy?
4. General Review and Audit Questions.
   a. Are required reviews and audits listed?
   b. Are the methods by which each review and audit will be carried out described?
5. Requirements Review Questions. Does the SQAP require the following items:
   a. Can each requirement be traced to the next higher level specification? Example of such specifications are system specifications and user requirements specifications.
   b. Can each derived requirement be justified?
   c. Are algorithms and equations described adequately and completely?
   d. Are logic descriptions correct?
   e. Are hardware/software external interfaces compatible?
   f. Is the description of and the approach to the man-machine interface adequate?
   g. Are symbols used consistently in the SRS?
   h. Is each requirement testable?
   i. Are verification and acceptance requirements adequate and complete?
   j. Are interface specifications complete and compatible?
   k. Is the SRS free from unwarranted design detail?
6. Preliminary Design Review Questions. Does the SQAP require the following items:

a. Are detailed functional interfaces between the software system under development and other software, hardware, and people fully defined?
b. Is the software design, taken as a whole, complete, consistent, and simple?
c. Can the design be shown to be compatible with critical system timing requirements, estimated running times and any other performance requirements?
d. Is the design testable?
e. Can each element of the preliminary design be traced to one or more specific requirements?
f. Can each requirement be traced to one or more specific design elements?

7. Detailed Design Review Questions. Does the SQAP require the following items:
   a. Is the design compatible with the SRS? That is, can each requirement be traced to the design, and can each design element be traced to one or more requirements?
   b. Are all logic diagrams, algorithms, storage allocation charts, and detailed design representations fully described?
   c. Are the interfaces compatible?
   d. Is the design testable?
   e. Does the final design include function flow, timing, sizing, storage requirements, memory maps, databases and files, and screen formats?
8. Test Questions.
   a. If the SQAP includes test requirements that are not in the V&V Plan, are all such requirements fully justified?
9. Problem Reporting and Corrective Action Questions.
   a. Does the SQAP include provisions to assure that problems will be documented, corrected, and not forgotten?
   b. Does the SQAP require that problem reports be assessed for validity?
   c. Does the SQAP provide for feedback to the developer and the user regarding problem status?
   d. Does the SQAP provide for the collection, analysis and reporting of data that can be used to measure and predict software quality and reliability?

4.1.3. Software Configuration Management Plan

Software configuration management (SCM) is the process by which changes to the products of the software development effort are controlled. This includes determining the configuration baseline and controlling change to the baseline.

Without a Software Configuration Management Plan (SCMP), it is difficult or impossible to manage configuration baseline change, or for software developers to know which versions of the various configuration items are current. Software modules that call other modules may be created using an incorrect version of the latter; in the worst case, this might not be discovered until operation under circumstances when correct operation is absolutely necessary to prevent an accident. This can occur if some functions are rarely needed, so are inadequately tested or linked into the final software product.

It is also possible that several people will have different understandings as to what changes have been approved or implemented, resulting in an incorrect final product.

4.1.3.1. Recommendation

A safety-related software project should have an SCMP. The plan should provide for baseline definition, change authorization, and change control. Detailed requirements for an SCMP are provided in Sections 3.1.3. The plan should be under configuration control.

4.1.3.2. Guideline

The SCMP may be organized according to IEEE Standard 828 and IEEE Guide 1042, as shown in Figure 3-4.

4.1.3.3. Assessment Questions

1. Organizational Questions.
   a. Are product interfaces that have to be supported within the project itself identified? Software-software? Software-hardware? Software maintained at multiple sites? Software developed at different sites? Dependence on support software?
   b. Does the SCMP define the required capabilities of the staff needed to perform SCM activities?

c. Does the plan specify what organizational responsibilities are likely to change during the life of the SCMP?

d. Does the plan state who has the authority to capture data and information and who has authority to direct implementation of changes?

e. Does the plan define the level of management support that is needed to implement the SCM process?

f. Does the plan define responsibilities for processing baseline changes?

- Responsibility for originating changes.
- Responsibility for reviewing changes.
- Responsibility for approving changes.
- Responsibility for administrating the change process.
- Responsibility for validating the changes.
- Responsibility for verifying change completion.

g. Does the plan specify who has the authority to release any software, data and documents?

h. Does the plan specify who is responsible for each SCM activity?

- Ensuring the integrity of the software system.
- Maintaining physical custody of the product baselines.
- Performing product audits.
- Library management.
- Developing and maintaining specialized SCM tools.

i. Does the plan identify the person or persons with authority to override normal SCM procedures during exceptional situations?

j. Does the plan explain how any such overrides will be reconciled with the product baselines, so that inconsistencies and lost updates do not occur?

2. SCM Responsibility Questions.

a. If the developer plans to use an existing CM organization, are required special procedures identified?

b. Does the plan delineate the assumptions made by the SCM group?

3. SCM Interface Control Questions.

a. Does the plan identify organizational interfaces that affect the SCM process, or are affected by the SCM process?

b. Does the plan identify the important interfaces between adjacent phases of the life cycle?

c. Does the plan identify interfaces between different software modules?

d. Does the plan identify interfaces between computer hardware and software modules? Between instrumentation and software?

e. Does the plan identify documents used in interface control? Where are these documents defined? How are they maintained?

4. SCMP Implementation Questions.

a. Are the resources planned for SCM sufficient to carry out the defined tasks? Do they take into account the size, complexity, and criticality of the software project?

b. Does the plan describe how SCM activities will be coordinated with other project activities?

c. Does the plan describe how phase-specific SCM activities will be managed during the different life cycle phases?

5. SCM Policy Questions.

a. Does the plan specify standard identification procedures? Actual needs in this area are specific to the project; audit procedures should ensure that the plan is adequate to prevent confusion. Unnecessary procedures can actually interfere with understanding, so naming policies should be adequately justified. Policies can include:

- Standard labels for products.
- Identification of the hierarchical structure of computer programs.
- Component and unit naming conventions.
- Numbering or version level designations.
- Media identification methods.
- Database identification methods.
- Documentation labeling and identification methods.

b. Do specific procedures exist for interacting with dynamic libraries? These procedures may include the following:

- Promoting a software module from one type of library to another. For example, from a development library to a production library.
- Documentation releases.
- Computer program product releases.
- Firmware releases.

c. Do specific procedures exist to manage the change process? This includes:
   — The handling of change requests.
   — Provision for accepting changes into a controlled library.
   — Processing problem reports.
   — Membership in the CCB.
   — Operation of the CCB.
   — Capturing the audit trail of important changes to product baselines.

d. Do standard reporting procedures exist? These include:
   — Summarizing problem reports.
   — Standard CM reports to management and assessors.

e. Are audit procedures defined in the CM plan?
   — Are procedures defined for performing audits of the CM process?
   — Are procedures defined for performing physical audits of configuration items?

f. Do procedures exist for accessing and controlling libraries? This includes:
   — Security provisions.
   — Change processing.
   — Backups.
   — Long-term storage.

6. Configuration Identification Questions.

a. Does the configuration identification scheme match the structure of the software product?

b. Does the plan specify when CIs will be given identification numbers?

c. Does the plan specify which items will be placed under configuration control (thus becoming configuration items)?

d. Is a separate identification scheme required for third-party software?

e. Does the plan explain how hardware and software identification schemes are related when the software is embedded in the hardware? This applies to such things as firmware, ROM code, and loadable RAM image code.

f. Does the plan explain if a special scheme is required for reusable software?

g. Does the plan explain how support software will be identified? This includes:
   — Language translators.
   — Operating systems.
   — Loaders.
   — Debuggers.
   — Other support software.

h. Does the plan explain how test data will be identified?

i. Does the plan explain how databases (such as installation configuration tables) will be identified?

j. Does the plan explain how baselines are verified?

k. Does the identification scheme provide for identifying different versions and different releases of the CIs?

l. Does the plan explain how physical media will be identified?

m. Are naming conventions available for each modifiable configuration item?

7. Configuration Control Questions.

a. Is the level of authority described in the plan consistent with the CIs identified in the plan?

b. Does the plan require that each significant change be under configuration control?

c. Does the plan fully describe the information needed to approve a change request?

d. Does the plan fully describe CCB procedures for approving change requests?

e. Does the plan require that safety-related change requests be so identified, and made known to the assessors during the next audit?

f. If different change procedures are required during different life cycle phases, are these differences fully described in the plan?

g. Does the plan fully describe procedures for accessing software libraries and controlling library interfaces? For ensuring that only one person at a time is able to change software modules?

h. Does the plan provide a method for maintaining a change history for each CI?

i. Does the plan provide for library backup and disaster recovery procedures? Are these procedures sufficient to enable a change history for each CI to be recovered if a library is lost?

j. Does the plan provide a method of associating source code modules with their derived object code modules and executable modules?

k. Does the plan provide procedures for keeping data files synchronized with the programs that use them?

l. Does the plan fully describe the authority of the CCB? Is this authority sufficient to control safety-related changes to the CI baseline?

m. Does the plan require the CCB to assess the safety impact of change requests?

n. Does the plan describe fully the procedures to be used by the configuration manager in order to oversee changes authorized by the CCB?

o. Does the plan fully describe the authority of the configuration manager? Is this authority sufficient to ensure that unauthorized changes do not take place? That authorized changes have been fully tested, reviewed, or analyzed for safety impact?

p. Is there a clearly-stated method for recovering an old version in the event that a newer version has problems in execution?

8. Configuration Status Accounting Questions.

   a. Does the plan describe what information must be made available for status reports?

   b. Does the plan describe each safety-related status report, including audience, content, and format?

   c. Does the plan provide a means of tracking problem reports that relate to safety, and making sure that each problem reported has been correctly resolved?

9. Audit and Review Questions.

   a. Does the plan provide for a single, separate audit trail for each CI and for the personnel working on each CI?

   b. Does the plan make provisions for auditing the SCM process?

   c. Does the plan provide for periodic reviews and audits of the configuration baseline, including physical audits of the baseline?

   d. Does the plan provide for audits of suppliers and subcontractors, if such are used?

   e. Does the plan make provisions to protect records needed in order to audit and assess the development process and development products?

10. Supplier Control Questions.

    a. Does the plan require suppliers and subcontractors to use a configuration management system consistent with, and equivalent to, that described for the development organization itself?

    b. Does the plan provide for periodic reviews of subcontractor CIs, including physical audits?

    c. Does the plan explain who is responsible for performing subcontractor reviews and audits?

11. Follow-Through Questions.

    a. Does evidence exist at each audit that the SCM plan is being followed?

4.1.4. Software Verification and Validation Plan

Software V&V is discussed in a separate report (Barter 1993). The following recommendations, guidelines, and assessment questions are taken from that report.

The software V&V plan is an essential element of the V&V process because it allows the developer, with regulatory approval, to define the exact nature of the process. Once defined, the V&V plan should be viewed as a "contract" between the developing organization and the regulating organization.

Without a Software V&V Plan, it will be difficult or impossible to be sure that the products of each phase of the software life cycle have been adequately verified, and that the final software system is a correct implementation of the requirements imposed upon it by the original system specifications.

4.1.4.1. Recommendation

A safety-related software project should have a Software V&V Plan. The size, scope, and contents of this plan should be appropriate to the size and complexity of the software system and the risk that can arise if the software system fails. Detailed requirements for a Software V&V Plan are provided in Section 3.1.4. The plan should be under configuration control.

4.1.4.2. Guideline

The Software V&V Plan may be organized according to IEEE Standard 1012, taking into account the contents of ANS standards 7-4.3.2 (Appendix E) and 10.4.

4.1.4.3. Assessment Questions

The assessment questions listed below are from Barter 1993.

1. General Questions.

   a. Does the V&V plan reference a management plan or a quality assurance plan?

b. Are specific elements of the higher-level plans addressed in the V&V plan?

c. Does the V&V plan identify the software that is being placed under V&V?

d. Is the purpose of the protection system clearly identified?

e. Is the scope of the V&V effort defined?

f. Is a clear set of objectives defined and is there a sense that the objectives will support the required level of safety?

2. V&V Overview Questions.

a. Is the V&V organization defined, along with its relationship to the development program?

* Does the plan call for a V&V organization that is independent from the development organization?
* Is the relationship between the V&V organization and other project elements (project management, quality assurance, configuration and data management) defined?
* Are the lines of communication clearly defined within the V&V organization?

b. Is a schedule defined that provides enough time for V&V activities to be effectively carried out?

* Does the schedule define the expected receipt dates for development products?
* Does the schedule define the time allotted to perform V&V activities?
* Does the schedule define the expected receipt dates for development products?
* Are realistic delivery dates set for V&V reports?
* Are the performing organizations defined for each activity?
* Are dependencies on other events clearly defined?

c. Are the resources needed to perform the V&V activities in the time allotted defined?

* Are the staffing levels defined and are they realistic?
* Are resource issues such as facilities, tools, finances, security, access rights, and documentation adequate addressed?

d. Are the tools, techniques, and methods to be used in the V&V process defined? Adequate consideration should be given to acquisition, training, support, and qualification of each tool, technique, and methodology.

* Has each tool been identified by name along with a description, identification number, qualification status, version, and purpose in each V&V activity?
* Has a distinction been made between existing tools and those that will have to be developed (if any)?

e. For those tools that have to be developed, is there an estimate of the time and resources needed to develop and qualify the tools?

* Have tool development activities been factored into the schedule?
* For existing tools, have the tools been adequately qualified?
* For existing tools that have not been adequately qualified, is there an estimate of the time and resources needed to qualify the tool and have the qualification activities been factored into the schedule?

f. Are techniques and methods defined with the same level of detail as tools?

* Has each technique and methodology been identified by name along with a description, qualification status and purpose in each V&V activity?
* Have the techniques and methods been adequately qualified?
* For those techniques and methods that have not been adequately qualified, is there an estimate of the time and resources needed to qualify them, and have the qualification activities been factored into the schedule?
* Is there a requirement that the V&V staff be trained in the techniques and methodologies used as part of the development effort?
* Is there a requirement that at least one member of the V&V staff be experienced in using the techniques or methodologies?

3. Software Life Cycle Management V&V Questions.

a. Is each task identified and tied into the project V&V goals? There should be a sufficient mix of tasks so as to completely support the project V&V goals and only those tasks that support the goals should be included.

* Have the tasks been clearly identified as to development materials to be evaluated, activities to be performed, tools and

techniques to be used, security and control procedures to be followed?

* Are all of the tasks identified in Table 1 of IEEE 1012 included?
* If any of the tasks listed in Table 2 of IEEE 1012 is identified, is there a clear justification for their use?

b. Does each task identify the methods to be used and the criteria to be applied to those methods?
   * Are the methods identified for each task consistent with the V&V overview with respect to resources, qualification, and schedule?

c. Is the input required by each task and the output from each task identified?
   * Are the development materials to be evaluated adequately identified?
   * In addition to specific reports identified for each phase of testing, is output identified for a summary of positive findings, summary of discrepancies, conclusions, and recommendations?

d. Is the method of handling anomalies encountered during each activity identified?
   * Does the output tie into another activity in such a way as to make the output meaningful? (i.e., are discrepancies reported to discrepancy tracking and resolution activities or are they only reported and then dropped?)
   * Is the content of a discrepancy report defined?
   * Will discrepancy reports include the name of the document or program in which the discrepancy was found?
   * Will discrepancy reports include a description of the discrepancy in sufficient detail so as to be understandable by a person not familiar with the original problem?
   * Will discrepancy reports include assessments as to the severity of the discrepancy and the impact of the discrepancy?

e. Are V&V schedule and resource requirements described in detail?
   * Have schedule and resources been adequately defined so as to give a feeling of confidence that the V&V effort will not be unduly rushed in its activities?

f. Are the planning assumptions for each V&V task described? Assumptions about the state of the development process may include completion of prior activities, status of previously identified discrepancies, availability of resources, and scheduling of tasks.
   * Have the assumptions been identified and are the assumptions consistent with the project plan?

g. Does the V&V plan include a contingency plan to identify risk factors that may cause the V&V activity to fail to perform its functions?
   * Have the risks been identified?
   * Is there a contingency plan identified for each risk for each task?
   * Are corrective procedures specified to minimize disruption to the V&V process?

h. Does the V&V plan identify the responsibilities of the V&V participants?
   * Have organizational elements for the entire project been identified in the project plan?
   * Have those organizational elements that interface to the V&V effort been identified in the project plan and in the V&V plan in a consistent manner?
   * Have specific responsibilities for each task been assigned to an organizational unit?
   * Have the interfaces between the V&V organization and the development organization and regulatory agency been defined?

i. Does the V&V plan establish a method of performing base line change assessments?
   * Is there a defined procedure for evaluating proposed software changes against V&V activities that are in progress or have been previously completed?
   * Is there a defined procedure for updating the V&V plan in the event that the software changes require a change in schedule and/or resources?

j. Does the V&V plan describe the means by which management overview of the V&V effort will be conducted?
   * Is the management structure defined?
   * Is there a defined procedure for management review of the V&V process?

* Does the V&V plan state that management is responsible for the technical quality of the V&V process?
* Will management receive summary reports of V&V process at each phase of the life cycle?
* Is there a procedure for the periodic assessment and updating of the V&V procedures, methods, and tools?

k. Does the V&V effort feed its information back into the overall development effort through review support?
* Is there a defined procedure for correlating V&V results with management and technical review documents?

4. System Level V&V Questions.

a. Is the V&V plan coordinated with project planning documents to ensure that early concept documents are available to the V&V effort?
* Does the project plan call for the identification of initiating documentation (statement of need, project initiation memo, or task statement), feasibility studies, performance goals, preliminary hazards analysis, and system definition documentation prior to beginning the V&V effort?
* Does the V&V plan require the generation and dissemination of anomaly reports?

b. Does the V&V plan explicitly define the activities required before the requirements development activities begin?
* Does the V&V plan require an evaluation of the system-level documentation to determine if the proposed concept will satisfy user needs and project objectives?
* Does the V&V plan require the identification of interfaces to other hardware and software systems along with any constraints imposed by those systems?
* Does the V&V plan require the identification of any constraints or limitation of the proposed approach?
* Does the V&V plan require an assessment of the hardware and software allocations?
* Does the V&V plan require the assessment of the criticality of each software item?

5. Requirements Activities V&V Questions.

a. Is the V&V plan coordinated with other project activities, especially those involving safety?
* Does the project plan call for the generation of concept documentation, SRS, interface requirements, hazards analysis, and user documentation prior to beginning the V&V requirements analysis?
* Does the V&V plan require the generation and dissemination of anomaly reports?
* Does the V&V plan define the method of resolving anomalies?

b. Does the V&V plan explicitly define the activities required during the requirements analysis?
* Does the V&V plan require the performance of a software requirements traceability analysis that traces elements of the SRS to elements of the system requirements?
* Does the trace go both from the SRS to the system requirements and from the system requirements to the SRS?

c. Does the V&V plan require a software requirements evaluation to help ensure that the SRS is both internally consistent and consistent with system objectives?
* Does the V&V plan require that the SRS be evaluated for safety, correctness, consistency, completeness, accuracy, readability, and testability?
* Is there a SRS standard and does the V&V plan require that the SRS conform to that standard?
* Does the V&V plan require that the SRS be evaluated for how well the specifications meet system objectives, software system objectives and address issues identified in the hazards analysis?
* Does the V&V plan require that the SRS be evaluated for performance issues?

d. Does the V&V plan require a software requirements interface analysis to help ensure that the software requirements correctly

define the interfaces to the software (both hardware and software)?

* Does the V&V plan require that the SRS be evaluated against hardware requirements, user requirements, operator requirements, and software requirements documentation?

e. Does the V&V plan require a system test plan and an acceptance test plan that will be used for later testing?

* Does the V&V plan require that a system test plan and an acceptance test plan be generated during the requirements phase?
* Does the V&V plan require that the plans be defined in enough detail to support the testing required?

6. Design Activities V&V Questions.

a. Is the V&V plan coordinated with other project activities, especially those involving safety?

* Does the project plan call for the generation of an SRS, software design documents, interface requirements, interface designs, and user documentation prior to beginning the design V&V analysis?
* Does the V&V plan require the generation and dissemination of anomaly reports?
* Does the V&V plan define the method of resolving anomalies?

b. Does the V&V plan explicitly define the activities required during the design analysis?

* Does the V&V plan require the performance of a design traceability analysis that traces elements of the software design document (SDD) to elements of the software requirements?
* Does the trace go both from the SDD to the SRS and from the SRS to the SDD?

c. Does the V&V plan require a design evaluation to help ensure that the software design document is internally consistent, testable, and meets established standards, practices, and conventions?

* Does the V&V plan require that the software design document be evaluated for correctness, consistency, completeness, accuracy, readability, and testability?
* Does the V&V plan require that the software design document be assessed as to the quality of the design?
* Is there a software design documentation standard, and does the V&V plan require that the software design documents conform to that standard?

d. Does the V&V plan require a design interface analysis to help ensure that the software design correctly meets the hardware, operator, and software interface requirements?

* Does the V&V plan require that the software design document be evaluated against hardware requirements, operator requirements, and software interface requirements documentation?

e. Does the V&V plan require a software component test plan, an integration test plan, and a test design be generated for use in later testing?

* Does the V&V plan require that a software component test plan, an integration test plan, and a test design be generated during the design phase?
* Does the V&V plan require that the plans be defined in enough detail to support the testing required?

7. Implementation Activities V&V Questions.

a. Is the V&V plan coordinated with other project activities, especially those involving safety?

* Does the project plan call for the generation of software design documents, interface design documents, source code listings, executable code at the software unit level, and user documentation prior to beginning the implementation V&V analysis and testing?
* Does the V&V plan require the generation and dissemination of anomaly reports?
* Does the V&V plan define the method of resolving anomalies?

b. Does the V&V plan explicitly define the activities required during the implementation phase?

* Does the V&V plan require the performance of an implementation traceability analysis that traces source code to elements of the software design?

* Does the trace go both from the code to the design and from the design to the code?

c. Does the V&V plan require a source code evaluation to help ensure that the source code is internally consistent, testable, and meets established standards, practices, and conventions?
* Does the V&V plan require that the source code be evaluated for correctness, consistency, completeness, accuracy, readability, safety, and testability?
* Does the V&V plan require that the source code be assessed as to the quality of the code?
* Is there a software coding standard and does the V&V plan require that the source code conform to that standard?
* Does the V&V plan require that the source code be evaluated for adherence to project coding standards?

d. Does the V&V plan require a source code interface analysis to help ensure that the source code correctly meets the hardware, operator, and software design documentation?
* Does the V&V plan require that the source code be evaluated against the hardware design, operator interface design, and software design documentation?

e. Does the V&V plan require generation and use of test cases to help ensure the adequacy of test coverage? The test cases bridge the gap between the test design and software design documents and the actual test procedures.
* Does the V&V plan require the generation of test cases for software component, integration, system, and acceptance testing?
* Does the V&V plan require the generation of test procedures for software unit, integration and system testing?
* Does the V&V plan require the execution of the test procedures for software components?

8. Integration and Validation Activities V&V Questions.

a. Is the V&V plan coordinated with other project activities, especially those involving safety?
* Does the project plan call for the generation of software design documents, interface design documents, source code listings, executable code at the software component level, and user documentation prior to beginning the integration and validation V&V analysis and testing?
* Does the V&V plan require the generation and dissemination of anomaly reports?
* Does the V&V plan define the method of resolving anomalies?

b. Does the V&V plan explicitly define the activities required during the integration and validation analysis and testing?
* Does the V&V plan require the performance of integration, system, and acceptance testing?
* Are the testing requirements sufficiently detailed so as to ensure that there is a very low probability of error during operation?

9. Installation Activities V&V Questions.

a. Is the V&V plan coordinated with other project activities, especially those involving safety?
* Does the project plan call for the generation of an installation package and previous phase summary reports prior to beginning the installation and checkout V&V analysis and testing?
* Does the V&V plan require the generation and dissemination of anomaly reports?
* Does the V&V plan define the method of resolving anomalies?

b. Does the V&V plan explicitly define the activities required during the installation analysis and testing?
* Does the V&V plan require the performance of an installation configuration audit?
* Does the V&V plan require the generation of a final report?

10. Operation and Maintenance Activities V&V Questions.

a. Is the V&V plan coordinated with other project activities, especially those involving safety?

* Does the project plan require that development schedules, concept documentation, SRSs, interface documents, software design documents, source code listings, user documentation, the installation package, and proposed changes be available prior to beginning the operation and maintenance V&V analysis and testing?
* Does the V&V plan require the generation and dissemination of anomaly reports?
* Does the V&V plan require that the software V&V plan be updated in response to changes?
* Does the V&V plan require the establishment of a system for evaluating anomalies, assessing proposed changes, feeding information into the configuration management process and iterating the V&V process as necessary?
* Does the V&V plan define the method of resolving anomalies?

4.1.5. Software Safety Plan

The Software Safety Plan is the basic document used to make sure that system safety concerns are properly considered during the software development.

Without a Software Safety Plan (SSP), it will be difficult or impossible to be sure that safety concerns have been sufficiently considered and resolved. Some matters are likely to be resolved by different people in different inconsistent ways. Other matters are likely to be overlooked, perhaps because people may assume that others have accepted those responsibilities.

4.1.5.1. Recommendation

A safety-related software project should have a Software Safety Plan. The size, scope, and contents of this plan should be appropriate to the size and complexity of the software system and the potential risk should the software system fail. Detailed requirements for a Software Safety Plan are provided in Section 3.1.5. The plan should be under configuration control.

4.1.5.2. Guideline

The SSP may be organized according to IEEE Draft Standard 1228, as shown in Figure 3-7. The software safety organization may be a separate organization, or may be part of the system safety or quality assurance organizations. The important issue is its independence from the development organization.

4.1.5.3. Assessment Questions

1. Organization and Responsibility Questions.
   a. Is the software safety program organization described? Is the organization structure practical? Can the organization successfully manage the software safety program?
   b. Are the lines of communication between the software safety organization, the project management organization, the software development team, and the regulators clear?
   c. Is the authority of the software safety organization defined? Is it sufficient to enforce compliance with safety requirements and practices?
   d. Is a single individual named as having overall responsibility for the conduct of the software safety program? Does this person have adequate authority, training in management, conflict resolution, and safety and software engineering to actually carry out this job?
   e. Does the manager of the software safety organization, and the organization itself, have sufficient autonomy from the development organization to ensure proper conduct of the software safety program?
   f. Does a mechanism exist for any person involved with the development project to communicate safety concerns directly to the software safety organization? Does the mechanism protect such a person from management reprisals?
2. Resource Questions.
   a. Does the SSP identify the resources that will be required to implement the plan? These include:
      — Financial resources.
      — Schedule resources.
      — Safety personnel.
      — Other personnel.
      — Computer and other equipment support.
      — Tools.
   b. Are the resources adequate?
3. Staff Qualification and Training Questions.

a. Does the SSP specify personnel qualifications for personnel performing the following safety-related tasks?
   — Defining safety requirements.
   — Performing software safety analysis tasks.
   — Testing safety-critical features of the protection system.
   — Auditing and certifying SSP implementation.
   — Performing process certification.
b. Are the qualifications sufficient to ensure the tasks are carried out correctly and that the safety concerns are adequately addressed?
c. Does the SSP define on-going training requirements for personnel with safety-related responsibilities?
d. Does the SSP specify methods for accomplishing these training objectives?
e. Are the training requirements sufficient to ensure that these people have the knowledge and ability to carry out their defined safety-related activities?

4. Software Life Cycle Questions.
   a. Does the SSP relate safety activities to the software development life cycle? (Note that this question may be addressed in the Software Development Plan rather than in the SSP.)
   b. Does the SSP provide a mechanism to ensure that known safety concerns are adequately addressed during the various life cycle activities?
5. Documentation Questions.
   a. Does the SSP describe what safety-related documents will be produced during the development life cycle?
   b. Are the contents of these documents described, either here or in some other development plan?
   c. Are the contents sufficient to ensure that known safety concerns are addressed in the appropriate places within the development life cycle?
   d. Is a means of document control described in the SSP?
   e. Is the document control system sufficient to ensure that required documents are preserved for development assessors?
6. Software Safety Program Records Questions.
   a. Does the SSP identify the safety-related records that will be generated, maintained, and preserved?
   b. Are these records sufficient to provide adequate evidence that the software safety program has been properly carried out during each phase of the software life cycle?
   c. Does the SSP identify a person responsible for preserving software safety program records?
   d. Does the SSP specify the tracking system to be used to monitor the status of safety-related documents?
7. Software Configuration Management Questions. (These questions could be addressed in the Software Configuration Management Plan instead of here.)
   a. Does the SSP describe the process by which changes to safety-critical software items will be authorized and controlled?
   b. Does the SSP describe the role and responsibility of the safety personnel in change evaluation, change approval and change verification?
   c. Is this sufficient to ensure that no new hazards are introduced into the protection system through changes to the protection software?
   d. Does the SSP describe how configuration management requirements will be met for software development tools, previously developed software, purchased software, and subcontractor-developed software?
8. Software Quality Assurance Questions.
   a. Does the SSP describe the interactions between the software safety organization and the quality assurance organization?
   b. Does the SSP require that the software safety organization have primary responsibility for ensuring safety, not the quality assurance organization?
9. Tool Support and Approval Questions.
   a. Does the SSP specify the process of approving and controlling software tool use?
   b. Does this process provide a means to ensure that tool use is appropriate during the different development life cycle activities?
   c. Does the SSP specify the process of obtaining approval for tool use, for installing approved tools, and for withdrawing tools?
   d. Does the person or persons who have authority to approve tool use, install tools, and

withdraw tools have adequate knowledge to make approval decisions in such a way that safety will not be compromised?

e. Does the person with this authority have an enforcement mechanism to ensure that limitations imposed on tool use are followed by the development team?

10. Previously Developed or Purchased (PDP) Software Questions.

    a. Does the SSP define the role of the software safety organization in approving PDP software?

    b. Does the software safety organization have authority to approve or disapprove the acquisition or use of PDP software?

    c. Does the SSP define an approval process for obtaining PDP software?

    d. Does this approval process include the following steps?

        — Determine the interfaces to and functionality of the PDP software.

        — Identify relevant documents that are available to the obtaining organization, and determine their status.

        — Determine the conformance of the PDP software to published specifications.

        — Identify the capabilities and limitations of the PDP software with respect to the safety requirements of the development project.

        — Using an approved test plan, test the safety-critical features of the PDP software in isolation from any other software.

        — Using an approved test plan, test the safety-critical features of the PDP software in conjunction with other software with which it interacts.

        — Perform a risk assessment to determine if the use of the PDP software will result in undertaking an acceptable level of risk even if unforeseen hazards result in a failure.

    e. Does the SSP provide a means to ensure that PDP software will not be used in a safety-critical product if (1) it cannot be adequately tested, (2) it presents significant risk of hazardous failure, (3) it can become unsafe in the context of its planned use or (4) it represents significant adverse consequence in the event of failure?

    f. If the answer to the previous question is "no," do equivalent analyses, test, and demonstrations by the vendor of the PDP software exist that show its adequacy for use in a safety-critical application?

11. Subcontract Management Questions.

    a. Does the SSP provide a means to ensure that safety-critical software developed by a subcontractor meets the requirements of the software safety program?

    b. Does the SSP provide a means to ensure that the subcontractor is capable of developing safety-critical software?

    c. Does the SSP provide a means to monitor the adherence of the subcontractor to the requirements of the SSP?

    d. Does the SSP provide a process for assigning responsibility for, and tracking the status of, unresolved hazards identified or impacting the subcontractor.

    e. Is the subcontractor required to prepare and implement a SSP that is consistent with this SSP, and obey it?

12. Process Certification Questions. (This applies only if the software product is to be certified.)

    a. Does the SSP provide a method for certifying that the software product was produced in accordance with the processes specified in the SSP?

13. Follow-Through Questions.

    a. Does evidence exist at each audit that the SSP is being followed?

14. Safety Analysis Questions. Assessment questions relating to safety analyses are discussed below, in sections 4.2.2, 4.3.3, 4.4.2, 4.5.2, 4.6.1, and 4.7.1.

4.1.6. Software Development Plan

The Software Development Plan is the plan that guides the technical aspects of the development project. It will specify the life cycle that will be used, and the various technical activities that take place during that life cycle. Methods, tools, and techniques that are required in order to perform the technical activities will be identified.

Without a development plan, there is likely to be confusion about when the various technical development activities will take place and how they will be connected to other development activities. The probability is high that the different team members will

make different assumptions about the life cycle that is being used, about what is required for each life cycle phase, and about what methods, tools, and techniques are permitted, required, or forbidden.

The differences among the members of the project technical team can result in a confused, inconsistent, and incomplete software product whose safety cannot be assured, and may not be determinable.

4.1.6.1. Recommendation

A safety-related project should have a Software Development Plan. The plan should describe the processes to take place during the development life cycle, and should describe the methods, tools, and techniques that will be used to carry out the development processes. Detailed requirements for a Software Development Plan are provided in Section 3.1.6. The plan should be under configuration control.

4.1.6.2. Guideline

The Software Development Plan may be organized as shown in Figure 3-8.

4.1.6.3. Assessment Questions

1. Life Cycle Process Questions.
   a. Is a software life cycle defined?
   b. Are the defined life cycle processes sufficient to provide confidence that a safe and adequate product will be produced?
   c. Are the inputs and outputs defined for each life cycle process?
   d. Is the source of each life cycle process input specified?
   e. Is the destination of each life cycle process output specified?
   f. Does each life cycle phase require a safety analysis?
   g. Does each life cycle phase include a requirement for an audit at the end of the phase?
2. Methods, Tools, and Techniques Questions.
   a. Are methods specified for each life cycle phase?
   b. Is an automated or semi-automated requirements tracking tool specified?
   c. Are formal requirements and design and implementation methods required?
   d. Is one specific programming language required?
   e. If more than one language is permitted, does the plan specify a method for choosing which language to use for each program module? Does the plan give a technical justification for permitting more than one programming language to be used?
   f. Does the plan specify what computers, compilers, libraries, and linkers will be used in the software development?
   g. Is a programming style guide specified?
3. Standards Questions.
   a. Are the technical standards that will be followed listed in the plan?
4. Schedule Questions.
   a. Are the technical milestones listed in the plan?
   b. Are the milestones consistent with the schedule given in the SPMP?
5. Technical Documentation Questions.
   a. Are the technical documents that must be produced listed?
   b. Are these documents consistent with those listed in the SPMP?
   c. Is a principal author listed for each document?
   d. Are milestones, baselines, reviews, and sign-offs listed for each document?
6. Follow-Through Questions.
   a. Does evidence exist at each audit that the Software Development Plan is being followed?

4.1.7. Software Integration Plan

The Software Integration Plan describes the general strategy for integrating the software modules together into one or more programs, and integrating those programs with the hardware.

Without a Software Integration Plan, it is possible that the complete computer system will lack important elements, or that some integration steps will be omitted.

4.1.7.1. Recommendation

A safety-related software project should have a Software Integration Plan. The size, scope, and contents of this plan should be appropriate to the size,

complexity, and safety-critical nature of the project. Detailed requirements for a Software Integration Plan are provided in Section 3.1.7. The plan should be under configuration control.

4.1.7.2. Guideline

The Software Integration Plan may be organized as shown in Figure 3.9.

4.1.7.3. Assessment Questions

1. Integration Process Questions.
   a. Does the Integration Plan specify the levels of integration required? Is this consistent with the software design specification?
   b. Does the Integration Plan specify what objects will be included at each level? These may include:
      — Hardware.
      — Software.
      — Instrumentation.
      — Data.
   c. Does the Integration Plan describe each step of the integration process?
   d. Does the Integration Plan describe the integration strategy to be used for each integration step?
2. Marginal Conditions Questions.
   a. Does the Integration Plan describe the environment that will be used to perform and test each integration step?
   b. Are software and hardware tools that will be used to integrate the computer system listed and described?
   c. Is there a priority-based list of the various integration steps?
   d. Was a risk analysis performed?
   e. If risks were identified, are preventive measures identified to avoid or lessen the risks?
3. Integration Organization Questions.
   a. Are the integration steps ordered in time?
   b. Are personnel who will be involved in the integration activity listed?
   c. Is this list up to date?
   d. Does a mechanism exist to keep the list up-to-date?
4. Integration Procedure Questions.
   a. Are the products of each integration step known?
   b. Are there complete instructions on how to carry out each integration step?
   c. Is there a contingency plan in case the integration fails?
   d. Is there a requirement that the completed product be placed under configuration control?
   e. Is there a procedure for delivering the product to the configuration management organization?
   f. Is there a procedure for delivering the product to the V&V organization for integration testing?
5. Follow-Through Questions.
   a. Was the Integration Plan followed?

4.1.8. Software Installation Plan

The Software Installation Plan governs the process of installing the completed software product into the production environment. There may be a considerable delay between the time the software product is finished and the time it is delivered to the utility for installation.

Without an Installation Plan, the installation may be performed incorrectly, which may remain undetected until an emergency is encountered. If there is a long delay between the completion of the development and the delivery of the software to the utility, the development people who know how to install the software may no longer be available.

4.1.8.1. Recommendation

A safety-related software project should have a Software Installation Plan. The size, scope, and contents of this plan should be appropriate to the size, complexity, and safety-critical nature of the project. Detailed requirements for a Software Installation Plan are provided in Section 3.1.8. The plan should be under configuration control.

4.1.8.2. Guideline

The Software Installation Plan may be organized as shown in Figure 3-10.

4.1.8.3. Assessment Questions

1. Installation Environment Questions.

   a. Is the environment within which the software will operate fully described?
2. Installation Package Questions.
   a. Are materials that are required for a successful integration listed?
3. Installation Procedures Questions.
   a. Does a step-by-step procedure exist for installing the computer system in the operational environment?
   b. Is this procedure complete?
   c. Does each step describe what installation items are required, and what is to be done with each installation item?
   d. Is the expected results from each installation step described? That is, how can the installer know that a step has been successfully completed?
   e. Are known installation error conditions described, and are recovery procedures fully described?
4. Follow-Through Questions.
   a. Was the Installation Plan fully tested?

4.1.9. Software Maintenance Plan

The Software Maintenance Plan controls the process of making changes to the completed software product. There may be a considerable delay between the completion of the development project and changing the product. An organization other than the development organization, termed the maintenance organization here, may actually do the maintenance.

Without a Maintenance Plan, it is not easy to know how the product may be changed, and what procedures are required in order to make changes. Inconsistencies and faults may be inserted into the product during maintenance changes, and this may not become known until the software needs to react to an emergency. In the worst case, maintenance that is carried out in order to improve the reliability of the software product may actually lessen its reliability.

4.1.9.1. Recommendation

A safety-related software project should have a Software Maintenance Plan. The size, scope, and contents of this plan should be appropriate to the size, complexity, and safety-critical nature of the project. The Plan should assume that maintenance will be carried out by some organization other than the development organization, and that development personnel will not be available to answer questions. Detailed requirements for a Software Maintenance Plan are provided in Section 3.1.9. The plan should be under configuration control.

4.1.9.2. Guideline

The Software Maintenance Plan may be organized as shown in Figure 3-11.

4.1.9.3. Assessment Questions

1. Failure Reporting Questions.
   a. Does a procedure exist for collecting operational failure data from the utilities that are using the software product?
   b. Does this procedure ensure that operational failures are documented?
   c. Does this procedure ensure that failure reports are delivered to the maintenance organization?
   d. Does the maintenance organization have a procedure to ensure that failure reports are maintained under configuration control?
   e. Does the maintenance organization have a procedure to ensure that each failure report is assigned to one individual who is responsible for analyzing the failure and determining the underlying fault that caused the failure?
   f. Can the maintenance organization management and the assessors always discover the status of each failure report?
2. Fault Correction Questions.
   a. Does the maintenance organization have a procedure in place to ensure that faults are corrected, or determined not to require correction?
   b. Does this procedure ensure that the documentation related to the fault will be corrected, if this is necessary?
   c. If the fault does not require correction, are the reasons for this fully documented?
   d. If the fault does not require correction, was a risk analysis performed to be sure that the fault cannot affect safety in any way?
   e. Does the procedure require that acceptance test cases be created to test for the previously-undetected fault?
   f. Does the procedure require that regression testing take place before a new release of the software is created?

3. Re-Release Questions.
   a. Do procedures exist to build and test new releases of the software?
   b. Do these procedures identify the events that may trigger the creation of a new release?
   c. Do these procedures require that the new release be fully tested before release to the utilities?
   d. Is the re-installation procedure fully documented?
4. Follow-Through Questions.
   a. Is there evidence, during periodic operational audits, that maintenance procedures are being followed by the utility and the maintenance organization?

4.2. Requirements Activities

The activities associated with the requirements stage result in a complete description of what the software system must accomplish as part of the reactor protection system.

There are a number of risks to not documenting the software requirements. Some requirements might be omitted from the design and implementation. Some requirements might be misunderstood, or interpreted differently by different members of the development team. Some hazards might not be covered by the requirements.

If a particular requirement is omitted, but is necessary to the design, then the designers or programmers are likely to explicitly or implicitly invent a requirement. An example of this is a missing timing requirement—say, that a pump must be turned on within two seconds of a particular signal being received from a sensor. If this requirement is not specified, the programmer might implicitly assume that there is no real timing issue here, and write the code in such a way that it takes five seconds to start the pump. This would be unacceptable, but would still meet the written requirement.

4.2.1. Software Requirements Specification

The Software Requirements Specification (SRS) documents the software requirements. These come from the protection system design and the protection system hazard analysis.

4.2.1.1. Recommendation

An SRS should be written for a reactor protection computer system. It should be correct, consistent, unambiguous, verifiable, and auditable. Each requirement should be separately identified. Each requirement should be traceable to the overall system design. Detailed requirements for a SRS are provided in Section 3.2.1.

4.2.1.2. Guideline

The SRS may be organized as shown in Figure 3-12. If available, an automated or semi-automated requirements tracking system should be used so that the software requirements can be traced through the design, implementation, integration, and validation stages of the development project. The use of a CASE tool to document the requirements is recommended. The use of a formal mathematics-based requirements specification language is recommended for the functional, performance, reliability, and security requirements.

4.2.1.3. Assessment Questions

In addition to the list of questions given here, the assessor may wish to consult other lists. In particular, IEC 880, Appendix A, contains a list of requirements characteristics, and Redmill 1989 contains assessment questions.

1. User Characteristics Questions.
   a. Is each category of user identified in the SRS?
   b. Is the expected experience level of each category of user defined?
   c. Are the training requirements for each category of user defined?
2. General Constraint Questions.
   a. Are known legal restrictions placed on the software development either described fully in the SRS, or referenced in the SRS?
   b. Are known hardware limitations described in the SRS?
   c. Are the SRS audit requirements specified?
   d. If special support software is required (such as operating systems, compilers, programming languages, or libraries), is the support software fully described?
   e. Are any required communications protocols defined?
   f. Are the critical safety considerations listed?

g. Are the critical security considerations listed?

3. Assumptions and Dependency Questions.
   a. Do any assumptions exist that are not listed in the SRS?
4. Impact Questions.
   a. If changes are required to existing hardware or buildings, are these documented in the SRS?
   b. If the system described in the SRS must be used with an existing system, are any required changes to that existing system fully documented?
   c. If the system will be deployed in an existing reactor, are known organizational and operational impacts described?
5. Functional Requirement Questions.
   a. Are the functional requirements individually identified?
   b. Is each requirement unambiguously stated?
   c. Do the functional requirements, taken as a whole, completely specify what the software must do?
   d. Do the functional requirements specify what the software must not do?
   e. Are the functional requirements, taken as a whole, mutually consistent?
   f. Is each functional requirement verifiable, either through inspection or testing of the completed software product?
   g. Can each requirement imposed by the protection system design be traced to a software requirement?
   h. Are the functional requirements complete?
6. Operator Interface Questions.
   a. Is every interaction between an operator and the software system fully defined?
   b. Are requirements for control panel and display layouts described?
   c. Are requirements for human reactions to software-generated messages described, including the amount of time available for making decisions?
   d. Are error messages described, with corrective actions that should be taken?
7. Instrumentation Interface Questions.
   a. Is the possible input from each sensor fully described? This can include:
      — Type of sensor (analog, digital).
      — Possible range of values.
      — Units of measurement.
      — Resolution of measurement.
      — Error bounds on measurements for the range of measurement.
      — Instrument calibration.
      — Conversion algorithms—analog/digital or physical units.
   b. Is the possible output to each actuator fully described? This can include:
      — Type of actuator (analog, digital).
      — Possible range of values and units.
      — Units of measurement.
      — Resolution of measurement, if analog.
      — Calibration requirements.
      — Conversion algorithms.
      — Any error responses.
8. Computer System Interface Questions. (Applies only if the protection system software must communicate with other software systems.)
   a. Are the interfaces between software systems fully defined?
   b. Is the form of each interface described—subroutine call, remote procedure call, communication channel?
   c. Is each interface message format and content defined?
   d. Is the transmission method and medium defined for each message?
   e. Are error detection methods defined for communication lines?
   f. Are communication protocols defined?
9. Performance Requirements Questions.
   a. Are the static performance requirements fully described?
   b. Are the protection system timing requirements included in the SRS?
   c. Are the timing requirements specified numerically?
   d. Are timing requirements expressed for each mode of operation?
   e. Are the performance requirements individually identified?
   f. Are the performance requirements, taken as a whole, mutually consistent?
   g. Is each performance requirement testable?
10. Reliability and Safety Questions.

a. Is each reliability and safety requirement individually identified?
b. Can each hazard identified in the system hazard analysis be traced to one or more software requirements that will prevent, contain, or mitigate the hazard?
c. Are backup, restart, and recovery requirements fully defined?
d. If the software must continue to operate in the presence of faults, are the fault tolerance requirements fully defined?
e. Are reliability and safety requirements specified for each possible mode of operation?
f. Are reliability requirements specified numerically?
g. If the software is required to diagnose hardware or software failures, are the classes of failures that will be detected identified?
h. Is the required response to any identified hardware or software failure described?
i. If the software is required to recover from hardware or software failures, are the recovery requirements fully described?
j. Are the reliability and safety requirements, taken as a whole, mutually consistent?
k. Is each reliability and safety requirement verifiable, either through inspection, analysis or testing of the completed software product?

11. Security Questions.
    a. Are the access restrictions imposed on operators, managers, and other personnel fully defined?
    b. Do requirements exist to prevent unauthorized personnel from interacting with the software system?
    c. Do requirements exist to prevent unauthorized changes to the software system?
    d. Are the security requirements individually identified?
    e. Can each security requirement be verified, either through inspection, analysis or test of the completed software product?
    f. Are the security requirements, taken as a whole, mutually consistent?

4.2.2. Requirements Safety Analysis

The purpose of the safety analysis is to identify any errors or deficiencies that could contribute to a hazard and to identify system safety considerations not addressed in the SRS.

The risk of not performing a safety analysis is that some hazards may be overlooked in the SRS, and that additional hazards may be added.

4.2.2.1. Recommendation

A Requirements Safety Analysis should be performed and documented. The analysis should determine which software requirements are critical to system safety, that all safety requirements imposed by the protection system design have been correctly addressed in the SRS, and that no additional hazards have been created by the requirements specification.

4.2.2.2. Guideline

The four analyses recommended in Section 3.2.2 may be performed.

4.2.2.3. Assessment Questions

1. General Question.
    a. Does the safety analysis present a convincing argument that the system safety requirements are correctly included in the SRS, and that no new hazards have been introduced?
2. Criticality Questions.
    a. Have the requirements that can affect safety been identified?
    b. Is there convincing evidence that the remaining requirements (if any) have no effect on safety?
3. Requirements Tracing Questions.
    a. Can each system safety requirement be traced to one or more software requirements?
    b. Can each software requirement be traced to one or more system requirements?
    c. Is there a requirement that the software not execute any unintended function? (Note: this may be very difficult to verify.)
4. Specification Questions.
    a. Is there convincing evidence that there are no missing or inconsistently specified functions?
    b. Is there convincing evidence that there are no incorrect, missing, or inconsistent input or output specifications?

c. Can timing and sizing requirements be met under normal, off-normal, and emergency operating conditions?

4.3. Design Activities

The software design activities translate the software requirements specifications into a hardware/software architecture specification and a software design specification.

The primary risks of not creating and documenting a formal software design are that it may be impossible to be sure that all requirements have been implemented in the design, and that no design elements exist that are not required. Either of these cases can create a hazard.

4.3.1. Hardware/Software Architecture Specification

The design architecture will show a hardware architecture, a software architecture, and a mapping between them. The hardware architecture shows the various hardware devices and the ways in which they are connected. The software architecture shows the executable software processes and logical communication paths between them. The mapping shows which processes operate in which hardware devices, and how the logical communication paths are implemented in the hardware communication paths.

It may happen that the design architecture cannot be completed until the software design, hardware design, and system design have been completed. The relative timing and overlap among these design descriptions is not specified here; that is the developer's responsibility.

4.3.1.1. Recommendation

A hardware/software architecture description should be prepared. It should be correct, consistent, unambiguous, verifiable, testable, and implementable. All major hardware devices and all major software processes should be included in the description. A mapping of software to hardware should be provided. A mapping of logical communication paths to physical communication paths should be provided.

4.3.1.2. Assessment Questions

1. Hardware Questions.
   a. Are known major hardware elements shown in the design architecture? This includes:
      — Computers.
      — File systems.
      — Sensors and actuators.
      — Terminals.
      — Communication networks.
   b. Does the design architecture show how the various hardware elements are connected together?
2. Software Questions.
   a. Are known independent software elements shown in the design architecture? This includes:
      — Processes, which perform computations.
      — Files and databases, which store information.
      — Input and output messages, which receive and transmit information.
      — Screen, which display information.
      — Communication, which moves information among processes, files and databases, input and output channels, and screens.
   b. Does the design architecture show how the various software elements are logically connected together?
3. Software to Hardware Mapping Questions.
   a. Does the design architecture show how each software element is mapped to a hardware element?

4.3.2. Software Design Specification

The Software Design Specification shows exactly how the software requirements are implemented in the software modules and programs.

4.3.2.1. Recommendation

A formal software design specification should be developed and documented. It should be correct, complete, internally consistent, consistent with the software requirements, unambiguous, verifiable, testable, and implementable. Each design element should be traceable to one or more specific requirements, and each software requirement should be traceable to one or more design elements.

4.3.2.2. Guideline

The use of a CASE system to document the design and the use of a formal mathematics-based design

specification language is recommended. The use of an automated or semi-automated requirements tracking system is also recommended so that the software requirements can be traced through the design to the implementation stage of the development project.

4.3.2.3. Assessment Questions

In addition to the list of questions given here, the assessor may wish to consult other lists. In particular, IEC 880, Appendix B, contains a list of design characteristics, and Redmill 1989 contains assessment questions.

Some of the assessment questions listed here should also be asked of the implemented system (code and data).

1. General Questions.
   a. Can every requirement given in the SRS be traced to one or more specific design elements that implement the requirement?
   b. Can every design element be traced to one or more specific requirements that the design element implements?
   c. Is there sufficient evidence to demonstrate that there are no unintended functions in the design?
   d. Is the design complete, consistent, correct, unambiguous, and simple?
2. Software Structure Questions.
   a. Are the static and dynamic structures simple, with minimal connections between design elements?
   b. Is the software structure hierarchical in nature?
   c. If stepwise refinement is used to create the software structure, is each level of the refinement complete, internally consistent, and consistent with the immediately higher level (if any)?
   d. Is the design such that safety-critical functions are separated from normal operating functions, with well-defined interfaces between them?
3. Design Element Questions.
   a. If any of the following concepts are used in the design, is adequate justification given for their use?
      — Floating point arithmetic.
      — Recursion.
      — Interrupts, except for periodic timer interrupts.
      — Multi-processing on a single processor.
      — Dynamic memory management.
      — Event-driven communications between processes.
   b. If more than one formal design method is used, are they mutually consistent?
   c. Is the input to each modules checked for validity?

4.3.3. Design Safety Analysis

The purpose of the safety analysis is to identify any errors or deficiencies in the design that could contribute to a hazard.

The risk of not performing a safety analysis is that some hazards that were identified in the requirements specification may be overlooked in the design, and that additional hazards may be added.

4.3.3.1. Recommendation

A Design Safety Analysis should be performed and documented. The analysis should determine which software design elements are critical to system safety, that all safety requirements imposed by the protection system design have been correctly implemented in the design, and that no additional hazards have been created by the design specification.

4.3.3.2. Guideline

The five analyses recommended in Section 3.3.3 may be performed.

4.3.3.3. Assessment Questions

Some of the assessment questions listed here should also be asked of the implemented system (code and data).

1. Logic Questions.
   a. Do the equations and algorithms in the software design correctly implement the safety-critical requirements?
   b. Does the control logic on the software design completely and correctly implement the safety-critical requirements?
   c. Does the control logic correctly implement error handling, off-normal processing, and emergency processing requirements?

   d. Is the design logic such that design elements that are not considered safety critical cannot adversely affect the operation of the safety-critical design elements?
2. Data Questions.
   a. Are safety-critical data items identified, with their types, units, ranges, and error bounds?
   b. Is it known what design elements can change each safety-critical data item?
   c. Is there convincing evidence that no safety-critical data item can have its value changed in an unanticipated manner, or by an unanticipated design element?
   d. Is there convincing evidence that no interrupt will change the value of a safety-critical data item in an unanticipated manner?
3. Interface Questions.
   a. Are the control linkages between design elements correctly and consistently designed?
   b. Has it been demonstrated that all parameters passed between design elements are consistent in type, structure, physical units, and direction (input/output/input-output)?
   c. Is there convincing evidence that no safety-critical data item is used before being initialized?
4. Constraint Questions.
   a. Have the design constraints listed in the requirements specification been followed in the design?
   b. Have known external limitations on the design been recognized and included in the design? This includes hardware limitations, instrumentation limitations, operation of the protection system equipment, physical laws, and similar matters.
   c. Will the design meet the timing and sizing requirements?
   d. Will equations and algorithms work across the complete range of input data item values?
   e. Will equations and algorithms provide sufficient accuracy and response times as specified in the requirements specification?

4.4. Implementation Activities

Implementation consists of the translation of the completed software design into code and data stores. The risks involved in writing the code are that the design may not be correctly implemented in the code, or coding errors may add additional hazards. Most of the assessment effort on implementation products involves code walk-throughs, inspections, and testing. Those topics are covered in Barter 1993 and Thomas 1993.

4.4.1. Code Listings

Coding may require the use of programming languages, database design languages, screen design languages, and other languages. The language level may vary from assembler to high level block languages.

4.4.1.1. Guideline

Assembly language should not be used in a safety-critical application without convincing justification. The programming language should be block-structured and strongly typed.

4.4.1.2. Guideline

Certain coding practices should not be used in a safety-critical application without convincing justification. This justification should substantiate that the use of the coding practice in a particular instance is safer than not using it. For example, not using the practice may, in some cases, require much additional coding, or obscure module structure, or increased probability of coding errors for some other reason. The following list of practices is indicative, not exhaustive. The ordering of the list contains no implication of severity.

- Floating point arithmetic.
- Recursion.
- Interrupts, except for periodic timer interrupts.
- Use of pointers and indirect addressing.
- Event driven communications.
- Time-slicing by the operating system.
- Dynamic memory management.
- Swapping of code into and out of memory.
- Loops that cannot guarantee termination.
- Storing into an array in such a way that the index cannot be guaranteed to fall within the index bounds of the array.
- Unconditional branches.
- Branches into loops or modules.

- Branching out of loops other than to the statement following the end of the loop, or to error processing.
- Nesting 'if' statements more than 3-4 levels deep.
- Use of default conditions in a 'case' statement.
- Use of multiple entry points in a subroutine.
- Overloading of variables (using the same variable for more than one purpose).
- Dynamic instruction modification.
- Implicit typing of variables.
- Use of 'equivalence' statements in FORTRAN.
- Modules with side effects other than output to an actuator, terminal or file.
- Passing procedures as parameters to subroutines.
- Testing floating point numbers for exact equality.

4.4.1.3. Assessment Questions

Some assessment questions given in Section 4.3.2.3 also apply to the code. Additional questions will arise through the V&V activity. No additional questions are suggested here.

4.4.2. Code Safety Analysis

The purpose of the safety analysis is to identify any errors or deficiencies in the code that could contribute to a hazard.

The risk of not performing a safety analysis is that some hazards that were identified in the requirements specification and covered by the design may be overlooked in the coding activity, and that additional hazards may be added.

4.4.2.1. Recommendation

A Code Safety Analysis should be performed and documented. The analysis should determine that the code correctly implements the software design, that the code does not violate any safety requirements and that no additional hazards have been created by the coding activity.

4.4.2.2. Guideline

The five analyses recommended in Section 3.3.3 may be performed.

4.4.2.3. Assessment Questions

In addition to the questions listed here, the questions given above in Section 4.3.4.3 should be considered as well, as far as they apply to the code.

1. Logic Questions.
   a. Does the code logic correctly implement the safety-critical design criteria?
   b. Are design equations and algorithms corrected implemented in the code?
   c. Does the code correctly implement the error handling design?
   d. Does the code correctly implement the off-normal and emergency operations design?
   e. Is there convincing evidence that no code considered non-critical can adversely impact the function, timing, and reliability of the safety-critical code?
   f. Is there convincing evidence that any interrupts that may be included in the code will not take precedence over or prevent the execution of safety-critical code modules?
2. Data Questions.
   a. Are the definition and use of data items in the code consistent with the software design?
   b. Is each data item in the code explicitly typed?
   c. Is there a convincing argument that no safety-critical data item can have its value changed in an unanticipated manner, or by an unanticipated module?
   d. Is there a convincing argument that no interrupt can destroy safety-critical data items?
3. Interface Questions.
   a. Have parameters that were passed between code modules been analyzed for consistency, including typing, structure, physical units, and number and order of parameters?
   b. Is the direction of parameters consistent, both internally in the code and externally with the software design?
   c. Have external interfaces been evaluated for correct format of messages, content, timing, and consistency with the Software Interface Design Description?
4. Constraint Questions.
   a. Is there adequate memory space in the target computer for the safety-critical code and data

structures? This should consider normal, off-normal, and emergency operating modes.

b. Is the actual timing of events in the code consistent with the timing analysis performed as part of the software design?

c. Can timing requirements actually be met?

d. Is there a convincing argument that the target computer will not be halted if an error occurs, unless such halting cannot impose a hazard?

e. Is it known what will happen if actual input values exceed the design specification in terms of values and frequency of occurrence?

4.5. Integration Activities

Integration consists of the activities that combine the various software and hardware components into a single system. The risk to an incorrect integration activity is that the system will not operate as intended, and that this will not be discovered until actual operation, possibly during an emergency.

Verifying that the integration activity has been successfully completed is part of the V&V inspection, analysis, and test activities. This is discussed in Barter 1993.

4.5.1. System Build Documents

The System Build Documents describe precisely how the system hardware and software components are combined into an operational system.

4.5.1.1. Recommendation

A System Build Specification should be written. It should describe precisely how the system is assembled, including hardware and software component names and versions, the location of particular software components in particular hardware components, the method by which the hardware components are connected together and to the sensors, actuators, and terminals, and the assignment of logical paths connecting software modules to hardware communication paths.

4.5.1.2. Assessment Questions

1. General Questions.

   a. Has it been verified that the System Build Specification actually works to build a correct system?

   b. Is there evidence that the system that is built and made ready for verification was actually built in accordance with the build specification?

4.5.2. Integration Safety Analysis

The Integration Safety Analysis ensures that no hazards have been introduced during the integration activity.

4.5.2.1. Recommendation

An Integration Safety Analysis should be performed and documented. The analysis should determine that the complete system does not violate any safety requirements and that no additional hazards have been created by the integration activity.

4.5.2.2. Assessment Questions

1. General Questions.

   a. Is there convincing evidence that the system meets protection system safety requirements?

   b. Is there convincing evidence that the system does not introduce any new hazards?

4.6. Validation Activities

Validation consists of the activities that ensure that the protection computer system, as actually implemented and integrated, satisfies the original externally-imposed requirements. This is discussed further in Barter 1993.

4.6.1. Validation Safety Analysis

The Validation Safety Analysis examines the entire system and the process of developing that system to ensure that system safety requirements have been met and that no hazards have been introduced at any point in the development process.

4.6.1.1. Recommendation

A Validation Safety Analysis should be performed and documented. The analysis should determine that the complete system does not violate any safety requirements and that no additional hazards have been created during the development process.

4.6.1.2. Assessment Questions

1. General Questions.

a. Is there convincing evidence that the completed protection computer system meets protection system safety requirements?
b. Is there convincing evidence that no new hazards were introduced during the development process?

4.7. Installation Activities

Installation is the process of moving the complete system from the developer's site to the operational site, possibly after considerable time delays. Only the safety analysis is discussed here.

4.7.1. Installation Safety Analysis

This final safety analysis verifies that the installed system operates correctly.

4.7.1.1. Recommendation

An Installation Safety Analysis should be performed and documented. The analysis should verify that the system was installed correctly, that it operates correctly, that the installed system does not violate any protection system safety requirements, and that the installed system does not introduce any new hazards.

4.7.1.2. Assessment Questions

1. General Questions.
   a. Has the hardware been installed correctly?
   b. Are hardware communication paths installed correctly?
   c. Is the software installed correctly?
   d. Have configuration tables been correctly initialized, if such are used?
   e. Are operating documents present, correct, complete, and consistent?

APPENDIX: TECHNICAL BACKGROUND

This appendix contains information on certain technical issues that are pertinent to this report. This appendix begins with a discussion of two prominent techniques that are occasionally recommended for software fault tolerance. This is followed by a section describing some of the modeling techniques that may be used to model reliability in general, and software reliability in particular. The final section briefly discusses software reliability growth models.

A.1. Software Fault Tolerance Techniques

Anderson (1985) and Laprie (1985) point out that the reliability of a computer system is determined by three different phases. First, one tries to keep faults out of the system. Second, since some faults will escape this effort, one tries to identify and eliminate them. Finally, since neither design nor elimination is perfect, and since some failures will occur during operation due to operational faults, one attempts to cope with them once they appear. There is a useful analogy here to the security of a bank. Bankers try to keep most robbers out; stop those that get in the door; and recover the loot from those that get away. The underlying philosophy is that of defense in depth.

Fault avoidance is concerned with keeping faults out of the system. It will involve selecting techniques and technologies that will help eliminate faults during the analysis, design, and implementation of a system. This includes such activities as selecting high-quality, reliable people to design and implement the system, selecting high-quality, reliable hardware, and using formal conservative system design principles in the software design.

Fault removal techniques are necessary to detect and eliminate any faults that have survived the fault avoidance phase. Analysis, review, and testing are the usual techniques for removing faults from software. Review and testing principles and procedures are well known, and are discussed in detail in Barter 1993 and Thomas 1993.

Fault tolerance is the last line of defense. The intent is to incorporate techniques that permit the system to detect faults and avoid failures, or to detect faults and recover from the resulting errors, or at least to warn the user that errors exist in the system. Since many systems designers are unfamiliar with fault tolerance ideas and techniques, this is discussed in Sections A.1.1-A.1.3.

In spite of the developer's best efforts, computer system failure may occur. If this does happen, the computer should fail in such a way as to not cause harm or lose information. This can be called *graceful degradation*, or *graceful failure*, and usually requires both hardware and software facilities. For example, sufficient battery power may be provided to permit a controlled shut-down by the software system.

Of course nothing can be done by the computer system to overcome some types of failures (such as those caused by fire, flood, or lightening strikes); other techniques such as seismic hardening or Uninterruptible power supplies may be required.

A fault-tolerant computer system requires fault-tolerant hardware and fault-tolerant software. The former is considered to be beyond the scope of this report. The following sources discuss hardware fault tolerance. The August 1984 and July 1990 issues of *IEEE Computer* are both devoted to fault tolerant hardware systems. Additional material can be found in the following papers and books listed in the References: Anderson 1985, Maxion 1987, Nelson 1987, Pradhan 1986 and Siewiorek 1982.

Nelson (1990) points out that the reliability of a system can be written as

R = Prob[no fault] + Prob[correct action | fault] * Prob[fault].

The first term represents the probability that the system is free from fault, and is achieved by high-quality design and manufacturing. If it is good enough, then fault tolerance may be unnecessary.

The second term represents the fault tolerance built into the system. It is the probability that faults may occur and the system continue to function.

Fault tolerance always has a cost attached, in terms of development time and money and operating time and money. This cost must be weighed against the costs of system failure. (See Berg 1987, Krishna 1987.)

Operator interaction with a computer system may not always be correct, so fault tolerance is applicable to creating a robust man-machine interface. This is discussed further in Maxion 1986.

This section begins with a brief general discussion of the use of redundancy in achieving fault tolerance. This is followed by an outline of the general process of recovering from software failures. The section ends with a description of two methods of achieving software fault tolerance that are frequently mentioned in the literature. Other commonly-available methods, such as the use of formal methods in software specification and design, or exception handling facilities within a programming language, are not discussed.

A.1.1. Fault Tolerance and Redundancy

Fault tolerance is always implemented by some form of redundancy. Nelson and Carroll (1987) identify four forms of redundancy; several others have been added here.

- *Hardware redundancy*. Extra hardware is used to achieve some aspects of fault tolerance. In triple modular redundancy (TMR), for example, three processors perform the same computation, and the results are compared. If all agree, it is assumed that all three processors are operating correctly. If two agree, it is assumed that the other processor has failed. This idea can be extended to communication paths, disk drives, memories, sensors, and other parts of the hardware system. The technique cannot compensate for design errors or for common-mode failures.
- *Information redundancy*. Redundant bits can be used to permit both detection and (in some cases) correction of errors in data or instructions. Information can also be duplicated by saving the result of a calculation in two variables, by maintaining duplicate files, or by redoing calculations. This can be effective in overcoming certain types of operational faults, but cannot overcome errors in the calculation itself. (See also Ammann 1988.)
- *Software redundancy*. Extra software can be provided to assist in detecting and recovering from failures.
- *Computational redundancy*. A calculation can be carried out using several different algorithms, with the results compared and one chosen that is deemed most likely to be correct.
- *Temporal redundancy*. Operations can be repeated to permit recovery from transient faults. A sensor can be re-read, a message can be retransmitted, or a record can be re-read from a disk. This technique is not useful, of course, if the fault is permanent or if there are hard real-time constraints.
- *Human redundancy*. It is possible to require that two or more individuals independently authorize action. This is sometimes called a "two-man rule."
- *Mixed redundancy*. Combinations of the techniques just listed can be quite effective in achieving redundancy. For example, one could require that hardware, software, and an operator all agree that a system is not in a hazardous state before permitting a potentially dangerous operation to take place.

None of these techniques covers all possible situations, all have potential flaws, and all have associated costs. In practice, a variety of techniques must be used, either at different levels within a system or in different subsystems (Bihari 1988).

A.1.2. General Aspects of Recovery

A fault-tolerant computer system must be capable of recovering from failures (by definition). Since the failure is just a symptom of one or more underlying faults that caused one or more errors, this recovery process must deal with these factors. Note that nothing is said here about how the techniques are to be implemented. The operating system, the application program, or a separate recovery system are among the possibilities.

The recovery process consists of the following general steps. Each is discussed further below.

1. Detect the failure.
2. Treat the error.
   a. Locate the error.
   b. Assess the damage.
   c. Recover from the error.
3. Treat the underlying fault.
   a. Locate and confine the fault.
   b. Diagnosis the fault.
   c. Repair and reconfigure the system.
   d. Continue normal operation.

A.1.2.1. Failure Detection

There are two major classes of failure detection techniques—off-line detection and on-line detection (Maxion 1987). The former is generally easier, but may not be possible if continuous operation of a system is required.

- Diagnostic programs can be run in an attempt to detect hardware faults, either periodically or continuously as a background process.
- Duplication of a calculation can be effective in detecting a failure. A calculation is performed two or more times, and the results are compared. Two CPUs could be used, two communication paths or two different algorithms. The technique can be quite effective in many circumstances, although there are a few potential problems that must be avoided. Identical failures (caused, for example, by design faults) often cannot be detected, and failures in the comparison unit may mask failures elsewhere or give a spurious indication of failure.
- Error detecting codes can be used to detect failures and to detect errors caused by failures. Parity checks, checksums, and cyclic codes are all applicable. This technique is particularly applicable to communication lines, memory, and external file data.
- Watchdog timers can be used to help ensure that a calculation has completed. The watchdog can be implemented in hardware or software, but must be distinct from the operation being controlled. The latter has a three-step algorithm:
    1. Set timer.
    2. Carry out operation.
    3. Upon successful completion, cancel timer.

If the time period elapses without the cancel, it is assumed that a failure has occurred, and the timing process will interrupt the controlled operation. Examples where this technique proves useful include detecting a process in an infinite loop, detecting a process that has aborted prematurely, detecting that no acknowledgment has been received from the recipient of a message, and detecting the failure of an I/O event to complete. In many cases the timer is initialized at the beginning of a cycle and then reinitialized each time the cycle is re-started. Thus the time stays on until some time when, for whatever reason, it isn't reinitialized before its time period is exhausted. It then does off and signals that a failure has occurred.

A.1.2.2. Error Treatment

There are three aspects to coping with errors once a failure has occurred: error detection, damage assessment, and recovery.

1. **Error Detection**. The success of all fault tolerance techniques is critically dependent upon the effectiveness of detecting errors. This activity is highly system dependent, so it is difficult to give general rules. The following suggestions are offered by Anderson (1985).
    - **Replication Checks**. It may be possible to duplicate a system action. The use of triply-redundant hardware is an example. A calculation is carried out by at least three separate computers and the results are compared. If one result differs from the other two, the idiosyncratic computer is assumed to have failed.
    - **Timing Checks**. If timing constraints are involved in system actions, it is useful to check that they are satisfied. For example a time-out mechanism is frequently used in process-to-process communications.
    - **Reversal Checks**. A module is a function from inputs to outputs. If this function is 1-1 (has an inverse) it may be possible to reproduce the inputs from the outputs and compare the result to the original inputs.
    - **Coding Checks**. Parity checks, checksums, cyclic redundancy codes, and other schemes can be used to detect data corruption. This is particularly important for communication lines and permanent file storage.
    - **Reasonableness Checks**. In many cases the results of a calculation must fall within a certain range in order to be valid. While such checks cannot guarantee correct results, they can certainly reduce the probability of undetected failures.
    - **Structural Checks**. In a network of cooperating processes it is frequently useful to perform periodic checks to see which processes are still running.
2. **Damage Assessment**. Once an error has been detected it is necessary to discover the full extent

of the damage. Calculations carried out using a latent error can spread the damage quite widely. Design techniques can be used to confine the effect of an error, and are therefore quite useful. One method is to carefully control the flow of information among the various components of the system. Note that these methods are almost impossible to retrofit to a system; this issue must be considered at the design stage.

3. **Error Recovery**. The final step is to eliminate all errors from the system state. This is done by some form of error recovery action. Two general approaches are available.
   - — **Forward Error Recovery**. In a few cases a sufficient amount of correct state is available to permit the errors to be eliminated. This is quite system dependent. If a calculation was performed on incorrect input (spreading the damage), and the correct input can be recovered, the calculation can simply be redone. Forward error recovery can be quite cost effective when it is possible.
   - — **Backward Error Recovery**. If forward error recovery is not possible, one must restore the system to a prior state that is known to be correct. There are three general categories of such techniques, and they are usually used in combination.
     - * **Checkpointing**. All (or a part) of the correct system state is saved, usually in a disk file.
     - * **Audit Trails**. All changes that are made to the system state are kept in a transaction log. If the system fails it can be reset to the latest checkpoint (or to the initial correct state), and the audit trail can be used to bring the system state forward to a correct current state. This technique is frequently used in database management systems (Date 1983) and can also be quite effective in application systems. Careful planning is necessary.
     - * **Recovery Cache**. Instead of logging every change to a system, it is possible to incrementally copy only those portions of the system state that are changed. The system can be restored only to the latest incremental copy unless an audit trail is also kept. Without the audit trail all transactions since the latest incremental copy are lost. In some cases this is sufficient.
     - * **Restarting**. When all else fails the system can be re-started from an initial known state.
   - — **Error Compensation**. This technique is possible if the erroneous state contains enough redundancy to enable the delivery of an error-free service from the erroneous (internal) state.

A.1.2.3. Fault Treatment

It may be possible to continue operation after error recovery. Although transient faults can possibly be ignored, nontransient faults must be treated sooner or later. Nelson and Carroll (1987) suggest four stages to fault treatment.

1. **Fault Location and Confinement**. One begins by isolating the fault by means of diagnostic checking (or some other technique). Whether or not the fault is immediately repaired after it has been located, it is generally wise to limit its effects as much as possible so that the remainder of the system can be protected. Where high availability is required, the component containing the fault may be replaced so that operation may continue; in such cases the next two steps take place off-line.
2. **Fault Diagnosis**. Once the fault has been isolated it will be necessary to uncover the exact cause. This could be a hardware operational fault, a software bug, a human mistake, or something else. The technique to use depends on the source of the fault.
3. **System Repair and Reconfiguration**. If there is sufficient redundancy this can take place during (possibly degraded) operation. Otherwise the system must be stopped and repaired or reconfigured.
4. **Continued Service**. The repaired system is restored to a working condition and restarted.

A.1.3. Software Fault Tolerance Techniques

Software can be used to compensate for failures in other portions of a computer system (such as hardware or operators) and to tolerate faults in the software portion of the system. This section is devoted to the latter problem. The assumption is that faults remain in

the software despite all the care taken in design, coding, and testing.

Two techniques are discussed here that have been used in attempts to achieve software fault tolerance: *n*-version programming and recovery blocks. In each case, the technique is described, advantages and disadvantages are discussed, and explanations are given as to how the technique implements the general recovery process described in the previous section.

It must be recognized that either of these techniques may actually decrease reliability, due both to the fact that more software has been written and to the added complexity of the software system. For example, suppose a fixed budget is available for testing. If three algorithms are written instead of one, then it is possible that only one-third as much testing will be done on each of the three. It is generally better to create a software system that is sufficiently simple so the design and code can be understood by competent people and declared safe; adding all the layers that are claimed to add fault tolerance and self-diagnosis may be self defeating.

Both techniques need watchdog timers to cope with errors (such as infinite loops) that cause an algorithm to fail to terminate. Both require a protection mechanism to avoid those errors caused when an error in one module corrupts the global state in unanticipated ways. These may be required in any case, so may not actually increase costs much.

The two techniques have been compared and analyzed in a number of articles: Abbott 1990, Arlat 1990, Knight 1991, Laprie 1990, Purtilo 1991, and Shimeall 1991.

A.1.3.1. N-Version Programming Technique

In the *n*-Version Programming Technique (Pradhan 1986) a program specification is given to *n* different programmers (or groups of programmers), who write the programs independently. Generally *n* must be at least 3 (and should be odd). The basic premise is that the errors that programmers make are independent; consequently the multiple versions are unlikely to go wrong in the same way. In execution, all versions are executed concurrently and voting is used to select the preferred answer if there is disagreement. Whenever possible different algorithms and programming languages are used, in the belief that this will further reduce the likelihood of common errors. The technique has been described in a series of papers from UCLA (Avizenis 1985; Bishop 1985; Kelly 1986; Saglietti 1986; Strigini 1985; Tso 1986).

N-version programming achieves redundancy through the use of multiple versions. Failures are detected by comparing the results of the different versions. Error location is done by assuming that versions whose results do not "win" the vote contain an error. No damage can occur so long as each version uses only local storage, and recovery is automatic. Faults are assumed to be located in the versions that are in error; diagnosis and repair must be done off-line.

N-version programming can be used to make decisions as to a course of action, but cannot be used to implement an action. That is, one would not want a motor to be activated by multiple pieces of software before voting occurs.

The voting mechanism is critical to *n*-version programming, since failure here can be much more serious than failures of the individual versions. Three types of faults must be considered: faulty implementation of the voting module, acceptance of incorrect results, and rejection of correct results. Brilliant (1989) points out that comparing floating point calculations in a voting algorithm must be done very carefully. Knight and Ammann (1991) give a hypothetical (and plausible) example of a 3-version system giving three different but acceptable answers, which would complicate voting.

There are a number of problems with this approach. To begin with the cost of programming and program maintenance is multiplied, since several versions are being written. Errors in the specification will not be detected (but ambiguities may be easier to find). Additional design is required to prevent serious errors in any of the versions from crashing the operating system, and to synchronize the different tasks in order to compare results. Code must also be written to do the comparison. Finally, there is some reason to believe that programmers do indeed make the same sorts of mistakes so that the assumption of independence is incorrect (Knight 1985; Knight 1986; Knight 1990; Brilliant 1990; Leveson 1990). This point is somewhat controversial; if it proves to be true, it is a fatal flaw.

Note that specification errors, voting errors, contamination of shared state data, and non-independent design and coding errors are all common-mode failures.

N-version programming has the potential of increasing the reliability of some aspects of a program, provided that development and testing time and funding are increased to cover the extra costs. However, the probability of common-mode failures must be factored into any calculation of increased reliability, and one should show that the extra time and money couldn't be better spent improving a single version.

A.1.3.2. Recovery Block Technique

The Recovery Block Technique uses redundancy in a different way (Cha 1986; Kim 1989; Pucci 1990). Here there is just one program, but it incorporates algorithms to redo code that proves to be in error. The program consists of three parts. There is a primary procedure that executes the algorithm. When it has finished, an acceptance test is executed to judge the validity of the result. If the result is judged to be valid that answer is passed back as the correct answer. However if the acceptance test judges the answer to be incorrect, an alternative routine is invoked. This alternative will generally be simpler to code, but is likely to run slower or use a different algorithm. The idea can be nested. The effect is to have a program module structured as follows:

```
answer = primary algorithm
if answer is valid then return (answer)
        else    {answer = second algorithm
                if answer is valid then return
        (answer)
                else    {answer = third algorithm
                .....
                        }
                }
```

Usually only the first algorithm will be required. One result of failure is to slow down the program's execution, so this technique may not always be usable.

In this technique, failures are detected by the acceptance test. The location of the error is determined by assuming that an algorithm is defective if the acceptance test fails for that algorithm. No damage can occur provided that the defective algorithm uses only local storage, and recovery is done by trying a different algorithm. Faults are assumed to be located in the versions in error; diagnosis and repair must be done off-line.

Many of the criticisms offered for the *n*-Version Programming Technique apply here as well, in one form or another. The independence assumption among programmers is not required, but independence of the various fall-back algorithms and the checking routine is required. There is an assumption that succeeding algorithms are more likely to be simpler, and therefore more likely to be correct; since they are presumably executed less frequently, this assumption must be verified.

The critical aspect of this technique is the acceptance test. It must be possible to distinguish correct from incorrect results or the technique cannot be used. It may happen that the acceptance test is as complex as the algorithm, in which case little may be gained. Indeed, an alternative algorithm may be the only available acceptance test, which again leads to *n*-version programming.

As discussed for *n*-version programming, recovery blocks can be used to decide on a course of action, but not to implement the action. The problem here, though, is not one of having multiple pieces of software attempting to control the same instrument. With recovery block, the failure of an acceptance test requires that the system state be returned to its condition prior to the execution of the block. If external devices are being controlled, this implies that they must be returned to their positions prior to the block, which may not be possible.

Common-mode failures exist here as well: specification errors, acceptance test errors, contamination of common state data, and common design and coding errors. Failures of this type will invalidate the assumption that the various versions are independent.

Another problem with recovery block is the time required to execute the back-up versions. In a hard real-time system this time may not be available, or execution timing may not be predictable The developer must verify that all timing restrictions can be met under all circumstances.

The conclusion is much the same as for *n*-version programming, since the costs and benefits are approximately the same. There is little reason to choose one over the other, except that the recovery block method probably requires less execution time when errors are not found. This may be balanced by the increase in execution time when an acceptance test is failed.

A.2. Reliability and Safety Analysis and Modeling Techniques

Section A.1 presented some ideas for developing software that must have unusually high reliability. This section examines a variety of techniques for understanding the reliability and safety of such software. These techniques are variously referred to as "analyses" or "models," but this distinction is somewhat pedantic. An analysis (such as fault tree analysis) is carried out by creating a model (the fault tree) of a system, and then using that model to calculate properties of interest, such as reliability. Likewise, a model (such as a Petri net model) may be created for a system in order to analyze some property (such as reachability).

Many techniques have been created for analyzing the reliability and safety of physical systems. Their extension to software is somewhat problematical for two reasons. First, software faults are design faults, while most well-known analysis techniques are directed primarily at operational faults (equipment breakage or human error). Second, software is generally much more complex than physical systems, so the methods may be very difficult to use when applied to software. However, the techniques are well understood, and there appear to be few alternatives.

The discussion begins by examining an elementary technique known as reliability block diagrams. It then moves to three classical techniques for analyzing system safety. Many models can be converted to Markov models, so this is discussed next. The section ends with an introduction to Petri nets, a modeling technique specifically developed for software. The descriptions given here are necessarily simplified, and the reader will need to consult the references given in each section for more detail. Many more techniques are available; see Bishop 1990 for brief summaries of many of them. One interesting approach not discussed here is that of formal proofs of correctness; see Atkinson 1991, Bishop 1990, Bloomfield 1991, and Linger 1979 for more information.

Safety analysis must be done for a system as a whole, not just for software. The word system includes computer hardware, software and operators; equipment being controlled by the computer system; system operators, users, and customers; other equipment; and (in some cases) the environment within which the system operates. This implies that software safety analysis will be only a portion of a larger safety analysis, and must be performed as part of that larger analysis.

The three techniques considered in Sections A.2.2 through A.2.4 all involve hazards, triggering events, and system components. They differ in emphasis, and all are generally required, in order to reduce the chances of missing important aspects of the system.

Fault tree analysis (FTA) and event tree analysis (ETA) were developed for use in the aerospace industry during the 1950s and 1960s. The techniques are complementary. FTA starts by assuming that some undesirable event (such as a particular type of accident) has occurred, and works backward attempting to uncover potential causes of the event. ETA assumes that an initiating event (such as a fault) has occurred and works forward, tracing the effects of that event. In both cases, the intent is to calculate the probability that consequences such as loss of life or property damage may occur. Failure modes and effects analysis (FMEA) starts with the system components and investigates the effect of a failed component on the rest of the system. FMEA and ETA are related.

Modeling generally requires computer assistance. A survey of software tools that can be used to evaluate reliability, availability and serviceability can be found in Johnson 1988. See also Bavuso 1987, Berg 1986, Feo 1986, Goyal 1986, Mainini 1990, Mulazzani 1985, Sahner 1987, Sahner 1987a, and Stiffler 1986.

A.2.1. Reliability Block Diagrams

Reliability block diagrams are easy to construct and analyze, and are completely adequate for many cases involving the operational reliability of simple systems. For a more complete discussion, see Bishop 1990, Chae 1986, Dhillon 1983, Frankel 1984, Kim 1989, Lloyd 1977, Pages 1986, and Phillis 1986.

Consider a system S composed of n components $C_1, C_2, \ldots, C_n$. Each component will continue to operate until it fails; repair is not carried out. The question is: what is the reliability of the entire system?

Suppose that component C_j has constant failure rate λ_j (and, therefore, a mttf of $1/\lambda_j$). The reliability of the component at time t is given by $R_j(t) = e^{-\lambda_j t}$. For example, a computer system might consist of a CPU with a failure rate of .23 (per thousand hours), a memory with a failure rate of .17, a communication line with a failure rate of .68 and an application

program with a failure rate of .11. For these components, the mean time to failure is 6 months, 8 months, 2 months, and 12 months, respectively.

In the reliability block diagram, blocks represent components. These are connected together to represent failure dependencies. If the failure of any of a set of components will cause the system to fail, a series connection is appropriate. If the system will fail only if all components fail, a parallel connection is appropriate. More complex topologies are also possible.

In the example, this system will fail if any of the components fail. Hence, a series solution is appropriate, as shown in Figure A-1.

The failure rate for a series system is equal to the sum of the failure rates of the components:

$$\lambda_S = \sum_{i=1}^{n} \lambda_i$$

For the example, the failure rate is 1.19 per thousand hours, giving a mttf of 840 hours (1.15 months). Notice that this is significantly smaller than the mttf for the least reliable component, the communication line.

The failure rate for a parallel system is more complex:

$$\frac{1}{\lambda_S} = \sum_{i=1}^{n} \frac{1}{\lambda_i}$$

Suppose the communication line is made dually or triply redundant. Communication line failure rates are now .68, .34, and .23, respectively, yielding a mttf of 2 months, 4 months, and 6 months. The reliability block diagram just the communication line portion is shown in Figure A-2.

If the dually redundant communication line is now inserted into the original system reliability block diagram, Figure A-3 results.

This improves the failure rate to .85 per thousand hours, for a mttf of 1.6 months. Making the communication line triply redundant gives a failure rate of .74 per thousand hours, improving the mttf to 1.85 months.

The connecting lines in a reliability block diagram reflect failure dependencies, not any form of information transfer (although these are sometimes the same). Figure A-3 is interpreted to mean that failure of the system will occur if the CPU fails, if the memory fails, if both communication lines fail, or if the application program fails. Failures in real systems are sometimes more complex than can be represented by simply building up diagrams from series and parallel parts. Figure A-4 gives an example.

Analysis of a reliability block diagram is, in the most general cases, rather complex. However, if the graph can be constructed from series and parallel components, the solution is quite easy.

The advantages of the reliability block diagram are its simplicity and the fact that failure rates can frequently be calculated rather simply. The reliability block diagram is frequently similar to the system block diagram. Severe limitations are the assumptions that failures of components are statistically independent, and that failure rates are constant over time. Complex nonrepairable systems are better analyzed using a fault tree, while analysis of repairable systems requires the use of a Markov model.

A.2.2. Fault Tree Analysis

The use of fault tree models has developed out of the missile, space, and nuclear power industries. Additional discussion on general fault tree analysis (FTA) and its application to computer software can be found in Altschul 1987, Belli 1990, Bishop 1990, Bowman 1991, Connolly 1989, Dhillon 1983, Frankel 1984, Guarro 1991, Hansen 1989, Henley 1985, Kandel 1988, Leveson 1983, McCormick 1981, and Pages 1986.

One begins by selecting an undesirable event, such as the failure of the computer system. In more complex systems that use computers as subsystems, the failure could involve portions of the larger reactor system as well.

The fault tree is developed by successively breaking down events into lower-level events that generate the upper-level event; the tree is an extended form of AND-OR tree, shown in Figure A-5.

This diagram means that event E1 occurs only if both of events E2 and E3 occur. E2 can occur if either of E4 or E5 occur. Event E3 will occur if any two of E6, E7, and E8 occur. Fault trees are generally constructed using AND, OR, and R-of-N combinations.

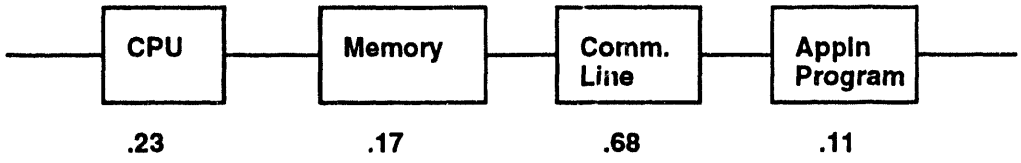


Figure A-1. Reliability Block Diagram of a Simple System

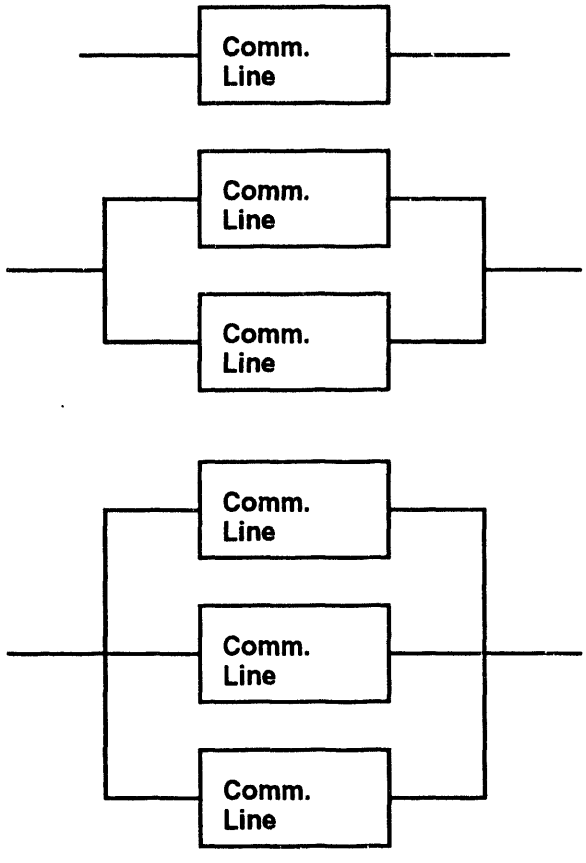
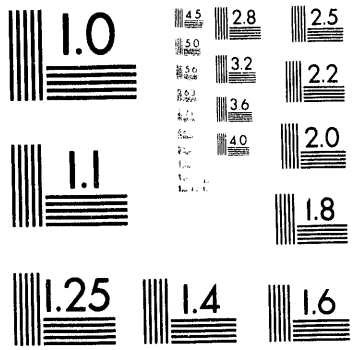


Figure A-2. Reliability Block Diagram of Single, Duplex, and Triplex Communication Line


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

2 of 2

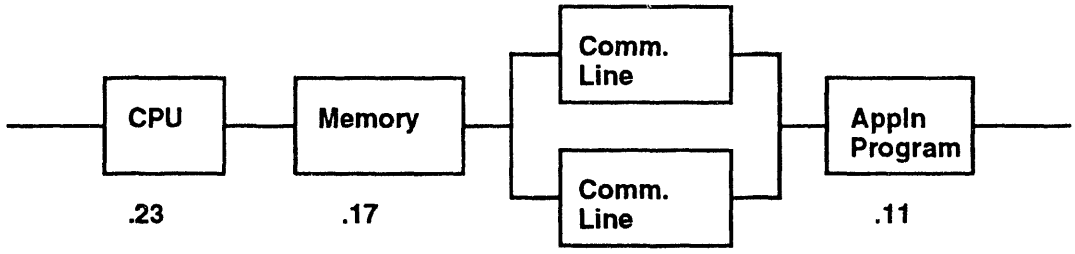


Figure A-3. Reliability Block Diagram of Simple System with Duplexed Communication Line

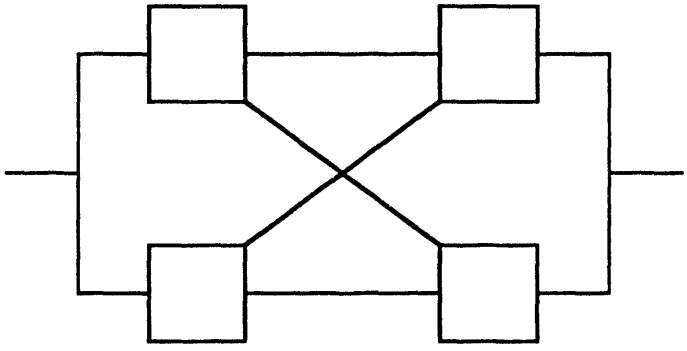


Figure A-4. Reliability Block Diagram that Cannot Be Constructed from Serial and Parallel Parts

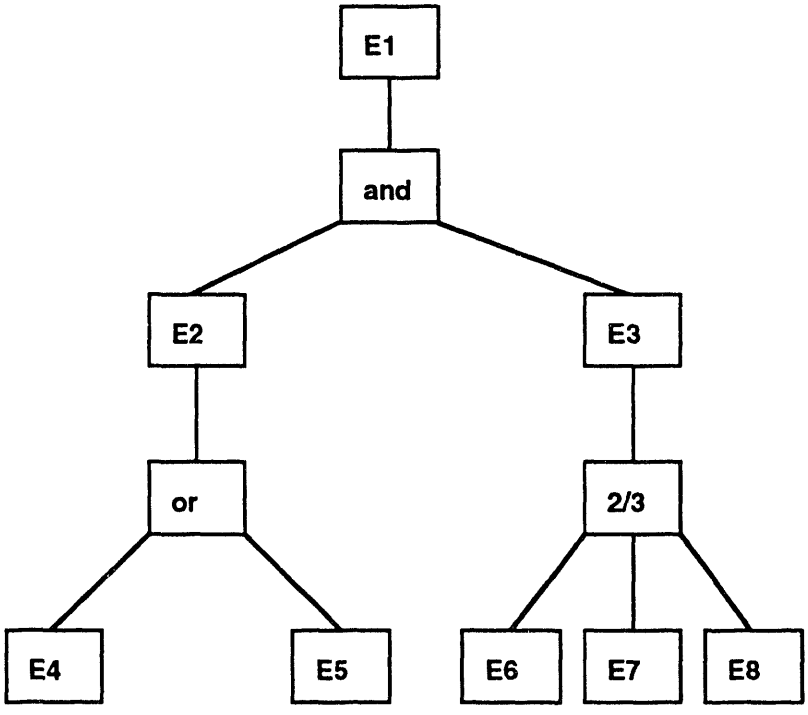


Figure A-5. Simple Fault Tree

The fault tree is expanded "downwards" until events are reached whose probability can be given directly. Note the assumption that the occurrence of the events at the bottom of the tree are mutually independent. In many cases, the actual probabilities of these events are estimated (or simply guessed); this is particularly true if they represent human failures, uncontrollable external events, software failures or the like. In general, the same event may occur several times at the lowest level if it can contribute to the main failure in several ways.

The fault tree can be evaluated from bottom to top. Consider the tree shown in Figure A-6, in which the lowest level events are not replicated. Suppose $p_j(t)$ denotes the probability that event E_j will occur by time t, and an AND node is to be evaluated.

Here, $p_1(t) = p_2(t) \cdot p_3(t)$; probabilities at AND nodes multiply. If there is an OR node, as shown in Figure A-7, have $p_1(t) = 1 - (1 - p_2(t)) \cdot (1 - p_3(t))$. Generalization to AND and OR nodes with more than two events should be clear. An R-of-N node represents a Boolean combination of AND and OR nodes, so its evaluation is straightforward, though tedious.

In practice, fault trees tend to have thousands of basic events, and replication of basic events is common. Analysis of such a tree requires computer assistance. Some minor generalizations of fault trees are possible; in particular, common mode failures can be handled with some difficulty.

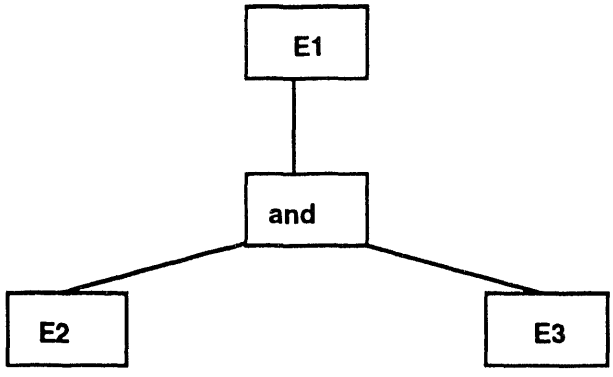


Figure A-6. AND Node Evaluation in a Fault Tree

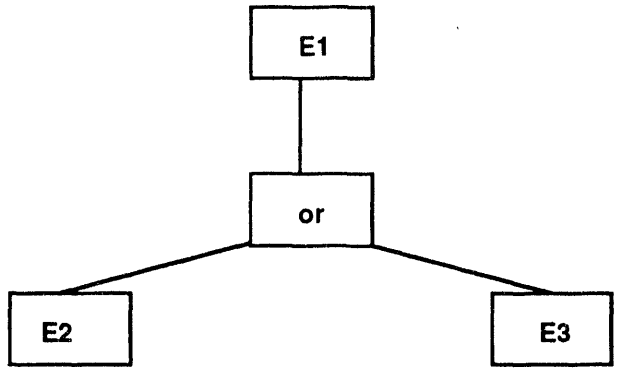


Figure A-7. OR Node Evaluation in a Fault Tree

Connolly gives an example of a software fault tree for a patient monitoring system. A portion of the tree is shown in Figure A-8, modified from Connolly 1989. The various events in the tree are described as follows:

E1. The patient monitor incorrectly reports the patient status.

E2. Faulty data algorithms

E3. Faulty patient data detection and reporting algorithms

E5. Failure to accurately report patient data over specified input range.

E51. ECG accuracy.

E52. Pressure accuracy.

E53. Temperature accuracy.

E54. SaO2 accuracy.

E55. NIBP accuracy.

E56. CO accuracy.

E57. ECG resp accuracy.

E6. Failure to update patient data values within specified response time

E61. ECG response time.

E62. Pressure response time.

E63. Temperature response time.

E64. SaO2 response time.

E65. NIBP response time.

E66. CO response time.

E67. ECG resp response time.

E4. Faulty alarm detection and reporting algorithms.

E7. Failure of each supported parameter to alarm within specified limit.

E71. ECG alarm limit.

E72. Pressure alarm limit.

E73. Temperature alarm limit.

E74. SaO2 alarm limit.

E75. NIBP alarm limit.

E76. CO alarm limit.

E77. ECG resp alarm limit.

E8. Failure of each supported parameter to alarm within specified time.

E81. ECG alarm time.

E82. Pressure alarm time.

E83. Temperature alarm time.

E84. SaO2 alarm time.

E85. NIBP alarm time.

E86. CO alarm time.

E87. ECG resp alarm time.

Fault trees provide a systematic and widely-used method for analyzing failures. Provided that one has accurate knowledge of the probabilities of the basic events, calculation of the probability of the top event is straightforward. Computer programs exist that can do this computation.

On the other hand, fault trees tend to become very large, particularly for software. Fault trees do not handle dynamic time-dependent events very well, and may not reveal the consequences of events occurring in the middle of the tree.

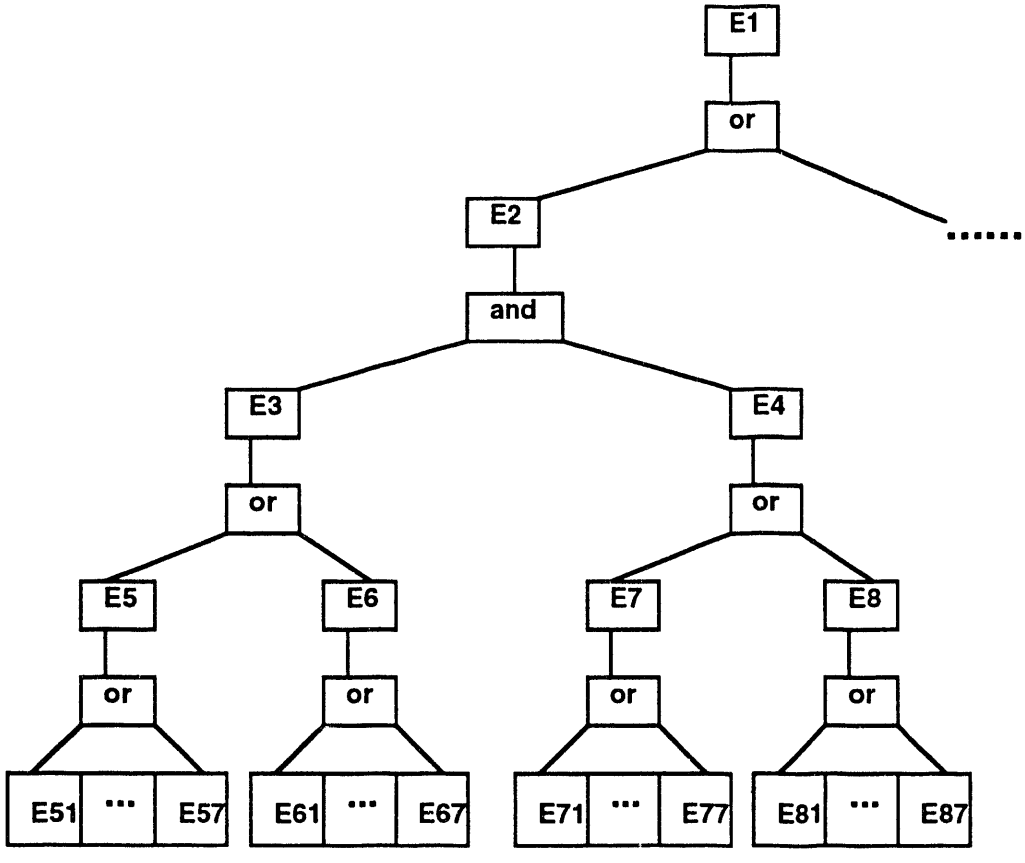


Figure A-8. Example of a Software Fault Tree

A.2.3. Event Tree Analysis

Event trees are similar to fault trees, except that an event tree is used to analyze the consequences of an event instead of its cause. The author has not seen any literature on the use of event trees for software. Brief general descriptions are given in Bishop 1990 and McCormick 1981. The discussion here is correspondingly brief.

To construct an event tree, an initiating event is selected in some system component, such as a wire breaking, an operator giving an incorrect command, or a software bug being executed. The component may react in a way that will be considered successful or unsuccessful. The system reacts by entering a *success state* or a *failure state*. This new state may itself cause a system reaction that can again be labeled as success or failure. Some states may have no successors if no further response to the initiating event can occur. For example, complete recovery from the fault or the occurrence of an accident may have no successor states.

This results in a tree called the *event tree*. An example is shown in Figure A-9, where the E_j are events, S_j is the probability that the system will react successfully to E_j and F_j is the probability that the system will react unsuccessfully. Note that $S_j + F_j = 1$.

The probability of the final outcomes can be calculated by multiplying the probabilities of the path from E_0 to the terminal event. For example,

$$prob[E_7] = F_0 \cdot S_2 \cdot S_5$$

Some of the terminal events will constitute system failures, such as accidents. The probability of ending up in a system failure state can be calculated by adding up the probabilities of the events in the set. Suppose E_4, E_6 and E_8 represent accidents. Then the probability of having an accident is just

$$prob[accident] = prob[E_4] + prob[E_6] + prob[E_8]$$
$$= S_0 \cdot F_1 + F_0 \cdot F_2 + F_0 \cdot S_2 \cdot F_5$$

Event trees are easy to draw and easy to understand. However, it is difficult to take different failure modes into account, particularly those involving dependent failures, common mode failures and subsystem interactions. This should not be the only analysis method used.

A.2.4. Failure Modes and Effects Analysis

Failure modes and effects analysis (FMEA), and its extension to failure modes, effects and criticality analysis (FMECA), are used to analyze the consequences of component failures. They are frequently performed during the design of a system in order to identify components that require extra care in design and construction. The technique can also be used to identify portions of a system in which redesign can yield significantly improved reliability and safety. The author has not seen either of these techniques suggested specifically for software. For more information, see Bishop 1990, Frankel 1984, Kara-Zaitri 1991, McCormick 1981, McKinney 1991, and Wei 1991.

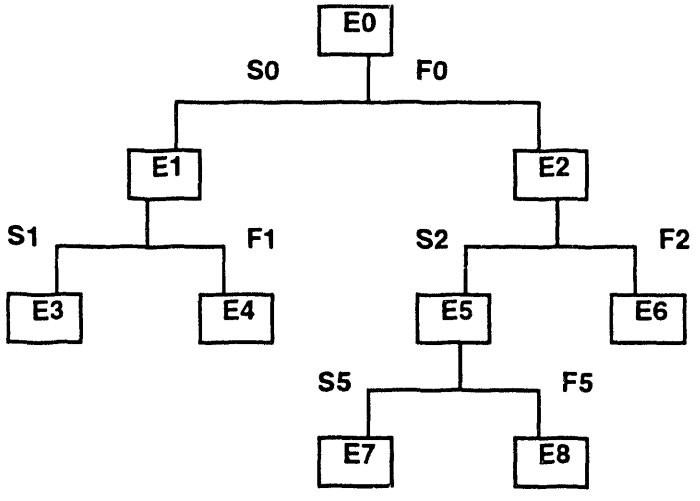


Figure A-9. Simple Event Tree

FMECA is largely a qualitative analysis procedure. The results will be a worksheet documenting failure modes, their effects and proposed corrective actions. Wei suggests the following procedure.

1. Define the ground rules. The following items should be considered:
    a. Determine the level of detail to which components will be divided. During the early design phase, not much detail will be known, so the analysis will be carried out at a relatively high level. Later on, when more detail is known about the structure of the system, the FMECA can be redone at the new level. At some point, software may become a system component.
    b. Define the conditions under which the system will operate. For electrical and mechanical equipment, this would include operating temperature, humidity, and cleanliness. For software it could include the computer system on which it will run, I/O equipment, communication equipment, and operating systems and compilers.
    c. Define extraordinary environmental conditions that must be considered. For hardware, this includes water, wind, fire, and earthquake. For software, this might be a hardware failure, such as a stuck bit in a register.
    d. Define successful operation. That is, how will one know that the system is operating correctly?
    e. Define failure. That is, how will one know that the system is performing incorrectly?
2. Prepare the FMECA plan. Wei states that the following should be included in the plan: worksheet formats, ground rules (discussed above), analysis assumptions, identification of the lowest level components, methods of coding, a description of the system, and definitions of failures.
3. Execute the plan. The following steps will be carried out.
    a. Determine the breakdown of the system structure, and document it. This can be done using a Functional Level Breakdown Structure (FLBS), Work Breakdown Structure (WBS), or any equivalent method.
    b. Use a consistent coding system so that each component can be uniquely identified. This is typically done by numbers separated by decimal points: for example, a motor assembly could be number 3.4; the motor in the assembly, 3.4.5; and a fuse in the motor, 3.4.5.2.
    c. Construct a functional block diagram (FBD) of the system and a reliability block diagram. These illustrate the flow of materials through the system, the flow of information through the system, system interfaces, interrelationships and interdependencies of the various components.
    d. Identify potential failure modes for all components and interfaces. Define the effects of such failures on the operation of the system.
    e. Classify each failure in terms of the severity of the worst potential consequences of the failure and the probability of the failure occurring.
    f. Calculate the criticality factor for each failure, as the product of the probability of failure and the severity of the consequences. Further analysis using fault trees, event trees, or other methods may be necessary for failures with large criticality factors.
    g. Identify changes to the design that can eliminate the failure or control the risk.
    h. Identify the effects of such design changes, and make recommendations as to carrying out the changes.
    i. Document the entire analysis, and discuss all problems that could not be resolved.

Severity may be determined by a four-level classification scheme. Each failure mode is assigned a number from 4 to 1, according to the following list.

4 Catastrophic failures can lead to loss of life, severe reduction of the system's potential output, or failures of high level components.

3 Critical failures can lead to personal injury, severe reduction in the output of a portion of the system, or serious degradation of system performance.

2 Minor (or marginal) failures cause some degradation of system performance or output, but they cannot lead to death or injury.

1 Insignificant (safe) failures have negligible effect on the system's performance or output.

The probability of occurrence factor may also be defined using a scale. Again, assign a number chosen from the following list, or assign the estimated probability of failure.

5 Frequent failures, defined as a single failure mode probability during an operating time interval that is greater than 0.20.

4 Reasonable failures, defined as a single failure mode probability between 0.10 and 0.20.

3 Occasional failures, defined as a single failure mode probability between 0.01 and 0.10.

2 Remote failures, defined as a single failure mode probability between 0.001 and 0.01.

1 Extremely unlikely failures, defined as a single failure mode probability less than 0.001.

FMEA and FMECA are systematic methods that are widely used to analyze failures and to discover the top events for fault trees. They can be used early in the system design stage, and can thus affect the design in such a way as to reduce hazards. This analysis does tend to be quite expensive and time consuming, although that must be weighed against the consequences of accidents and other failures that might be missed.

A.2.5. Markov Models

Markov models are used to capture the idea of system state, and a probabilistic transition between states. Here, the *state* represents knowledge of which components are operational and which are being repaired (if any). See Aldemir 1987, Amer 1987, Bishop 1990, Bobbio 1986, Cheung 1980, Geist 1986, Pages 1986, Siegrist 1988, Siegrist 1988a, and Smith 1988 for a more complete introduction.

Straightforward block diagram models and fault tree models can be evaluated using Boolean algebra, as described earlier. More complex models are generally translated into Markov models first. Systems whose components are repairable and systems where component failures have interactions are usually modeled directly by Markov models, with cycles. A number of examples are given here.

Throughout this section, a hardware system S with components $C_1, C_2, \ldots, C_n$ is considered; there may be more than one instance of each component. Component C_j has constant failure rate λ_j and constant repair rate μ_j.

Begin with a system that contains three CPUs. Only one is required for operation; the other two provide redundancy. Only one repair station is available, so even if more than one CPU is down, repairs happen one at a time. If state k is used to mean "k CPUs are operating," the Markov model is shown in Figure A-10.

This has a striking resemblance to a performance model of a single queue with three servers and limited queue size and can be solved in the same way to yield the probabilities of being in each of the four states (Sauer 1981). The interpretation is that the system is operational except in state 0.

Now, suppose two memories are added to the system. Failure rates are λ_c and μ_m for the CPUs and memories, respectively, and similar notation is used for repair rates. The label "k,l" for states, means "k CPUs and l memories are operational." The system is operational except in states "0,0," "0,i," and "j,0." The diagram is given in Figure A-11.

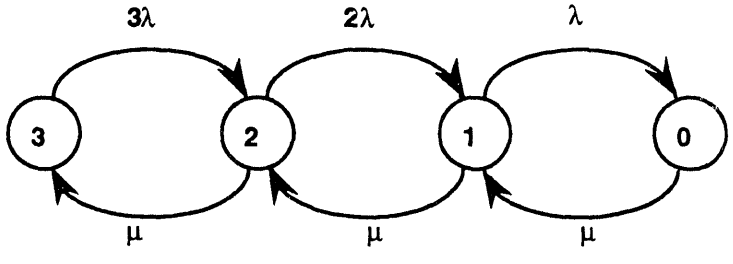


Figure A-10. A Simple Markov Model of a System with Three CPUs

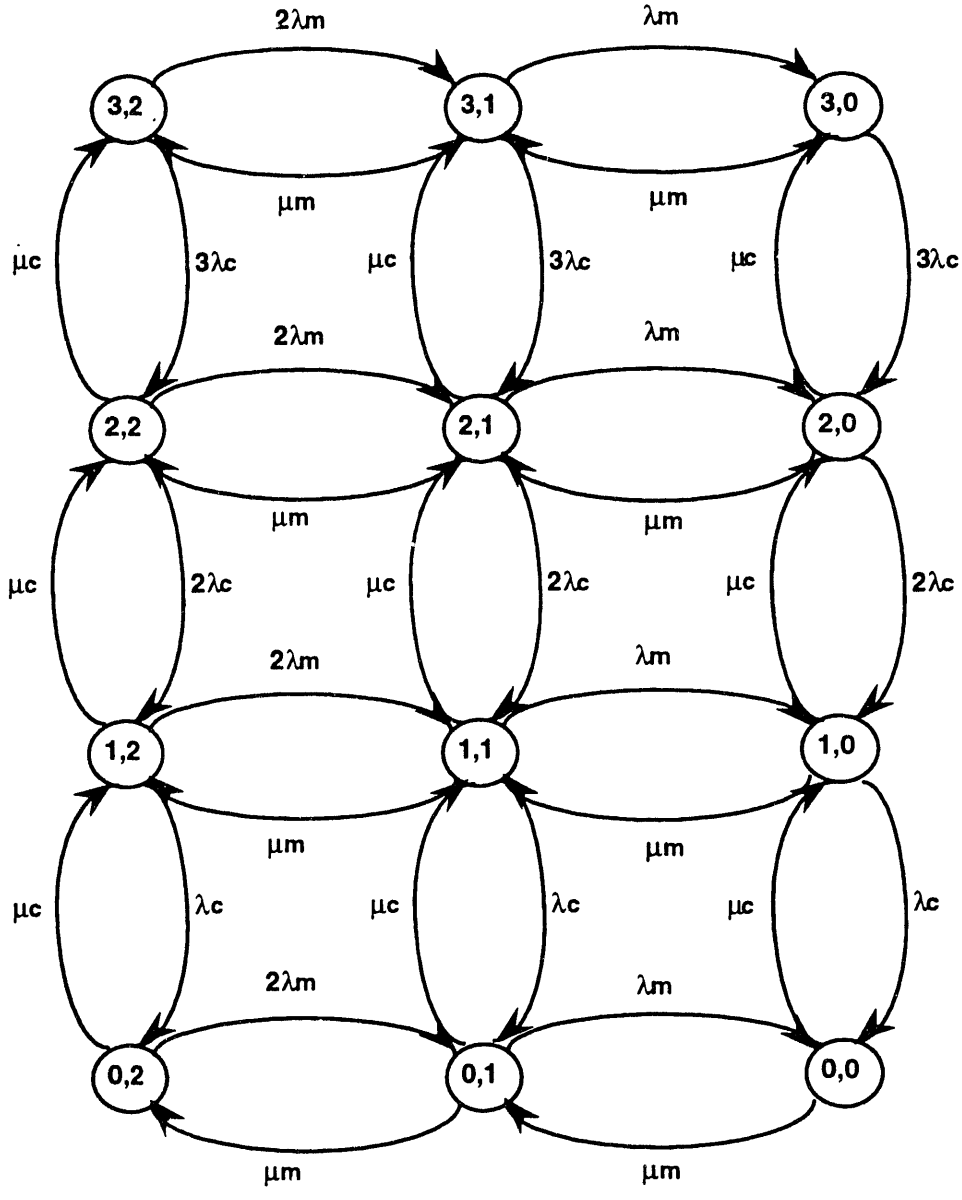


Figure A-11. Markov Model of a System with CPUs and Memories

It is clearly possible to model the case where failure rates vary with circumstances. In the first example (with three CPUs), it might happen that each CPU has a failure rate of λ when all three are available, λ' if two are available and λ'' if only one is available. This might reflect a situation where the load is shared among all operational CPUs; increased load causes an increased failure rate for some reason: $\lambda < \lambda' < \lambda''$. The diagram is modified as shown in Figure A-12.

Failure rates in computer systems vary over time. In particular, failures tend to be more frequent immediately after preventive maintenance; after this transition period, the failure rate will return to the normal value. Transient failures frequently show a similar pattern: a memory unit will show no failures for months; then a day with hundreds of transient faults will occur; the next day, the situation is back to normal, even though no repair was done. Such situations cannot be modeled with reliability block diagrams or fault trees, but can easily be modeled using Markov chains. The next example shows four states, modeling the memory problem. State 1 is the normal operational state and state 0 represents a "hard" memory failure. The failure rate from state 1 is λ and the memory repair rate from state 0 is μ.

There is, however, a very small chance of changing to state 3, where frequent transient memory errors are possible. Once there, memory faults occur with rate λ' >> λ. Since these are transient, "repair" happens very rapidly (within milliseconds). Eventually the system returns to state 1 and normality resumes. Note that hard failures can also occur in state 3; it is assumed that the process of repairing these will abort the period of transient faults, so a transition is made to state 0. Other models are possible, of course. See Figure A-13.

This technique can be used to fit a variety of problems, ranging from simple to complex. Parametric studies can be carried out, which can be used to uncover the impacts of different system configurations, different activity rates, and different repair strategies. However, large Markov models are difficult to solve analytically, so simulation becomes necessary. Most Markov models of real systems have very many states, so are large. Solving Markov models usually requires much computer assistance, and computer programs do exist to help.

A.2.6. Petri Net Models

A *Petri net* is an abstract formal model of information flow. Petri nets are used to analyze the flow of information and control in systems, particularly systems that may exhibit asynchronous and concurrent activities. The major use of Petri nets has been the modeling of systems of events in which it is possible for some events to occur concurrently, but there are constraints on the concurrence, precedence, or frequency of these occurrences. One application, for example, is to analyze resource usage in multiprogramming systems. Extensions have been created to permit performance analysis, and to model time in computer systems. See Peterson 1977 for a general introduction to Petri nets, and the following for applications to reliability and safety: Bishop 1990, Geist 1986, Hansen 1989, Hura 1988, Jorgenson 1989, Kohda 1991, Leveson 1987, Ostroff 1989, Shieh 1989.

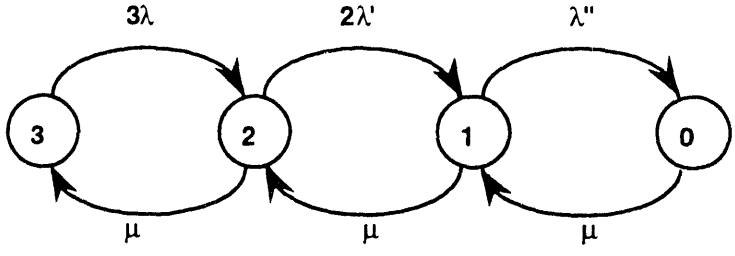


Figure A-12. Simple Markov Model with Varying Failure Rates

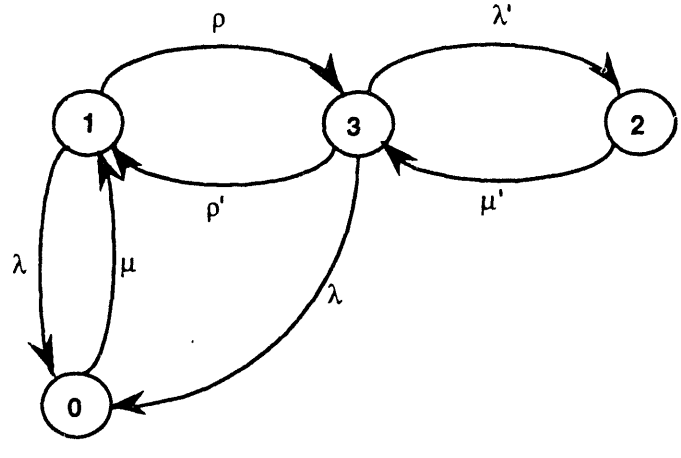


Figure A-13. Markov Model of a Simple System with Transient Faults

A Petri net is a marked bipartite directed graph. The graph contains two types of nodes: circles (called *places*) and bars (called *transitions*). These nodes, places, and transitions are connected by directed arcs from places to transitions and from transitions to places. If an arc is directed from node *i* to node *j* (one of which is a place and the other of which is a transition), then *i* is an *input* to *j* and *j* is an *output* of *i*. In Figure A-14, place p_1 is an input to transition t_2, while places p_2 and p_3 are outputs of transition t_2..

The execution of a Petri net is controlled by the position and movement of markers (called *tokens*) in the net. Tokens, indicated by black dots, reside in the circles representing the places of the net. A Petri net with tokens is a *marked Petri net*. Figure A-15 shows an example of a marked Petri net; it is just the former example, with a marking added.

The use of tokens is subject to certain rules. Tokens are moved by the *firing* of the transitions of the net. A transition is enabled for firing only if there is at least one token in each of its input places. The transition fires by removing one token from each input place, and placing a new token in each output place. In Figure A-15, transition t_2 is the only one enabled. If it fires, Figure A-16 results.

The result of the firing is to remove the token from p_1 and add tokens to p_2 and p_3. As a consequence, t_1, t_3 and t_5 are enabled. If t_1 and then t_2 now fire, notice that place p_3 will have two tokens. As a result, t_5 may now fire, followed by t_3.

On the other hand, if t_3 fires, only t_1 remains enabled. t_5 cannot fire, since the token in its input place p_3 has been used up.

This brief discussion encompasses the components of basic Petri nets.

Consider a computer system that permits concurrent or parallel events to occur. Petri nets can be used to model (among other things) two important aspects of such systems: *events* and *conditions*. In this view, certain conditions will hold in the system at any moment in time. The fact that these conditions hold may cause the occurrence of certain events. The occurrence of these events may change the state of the system, causing some of the previous conditions to cease holding, and causing other conditions to begin to hold.

For example, suppose the following two conditions hold: a disk drive is needed and a disk drive is available. This might cause the event allocate the disk drive to occur. The occurrence of this event results in the termination of the condition a disk drive is available and the beginning of the event no disk drive is available.

The Petri net models conditions by places and events by transitions. The occurrence of an event is modeled by the firing of the corresponding transition.

Another example involves the mutual exclusion problem. This is a problem of enforcing coordination of processes in such a way that particular sections of code called *critical regions*, one in each process, are mutually excluded in time. That is, if Process 1 is executing its critical region, then Process 2 may not begin the critical region until Process 1 has left its critical region.

This problem can easily be solved by using P and V synchronization operations. They operate on *semaphores*, and P and V are the only instructions allowed to execute on a semaphore. A semaphore can be thought of as a variable that takes integer values. P and V are defined as follows; each of these definitions describes a primitive instruction, which is carried out by the computer without interruption.

P(S)	wait until S > 0; then set S = S—1
V(S)	S = S + 1

Notice that a process executing a P instruction must wait until the semaphore is positive before it can decrement it and continue. Code for both processes looks like this:

P(mutex)

execute critical region

V(mutex)

where *mutex* is a global mutual exclusion semaphore used by both processes.

To model this as a Petri net, consider the semaphore to be a place. A V operation places a token in the semaphore; a P operation removes a token. The solution to the mutual exclusion problem is shown in Figure A-17.

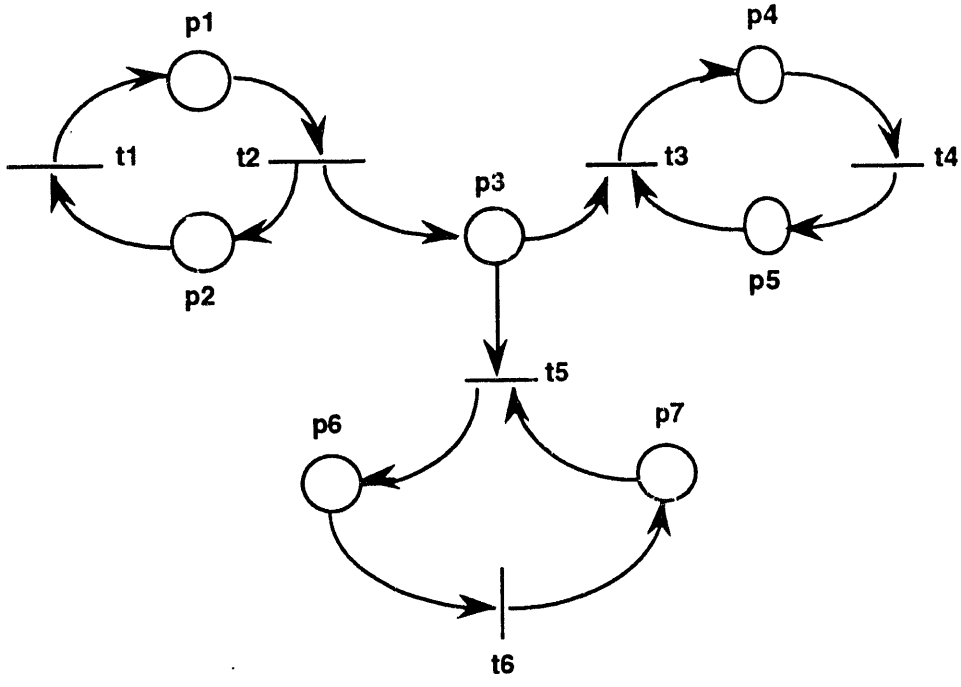


Figure A-14. An Unmarked Petri Net

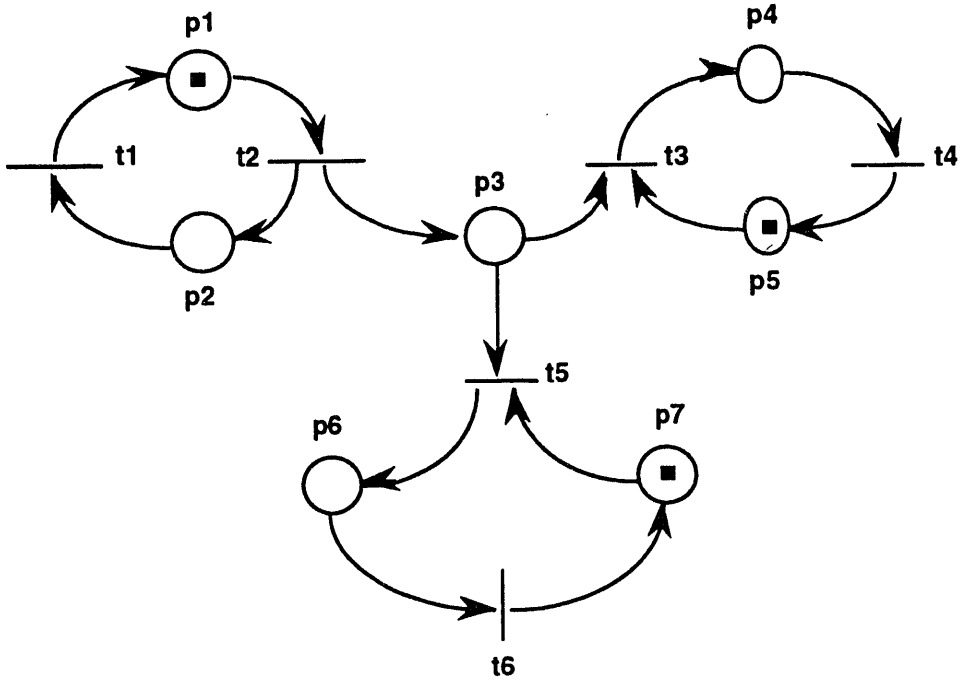


Figure A-15. Example of a Marked Petri Net

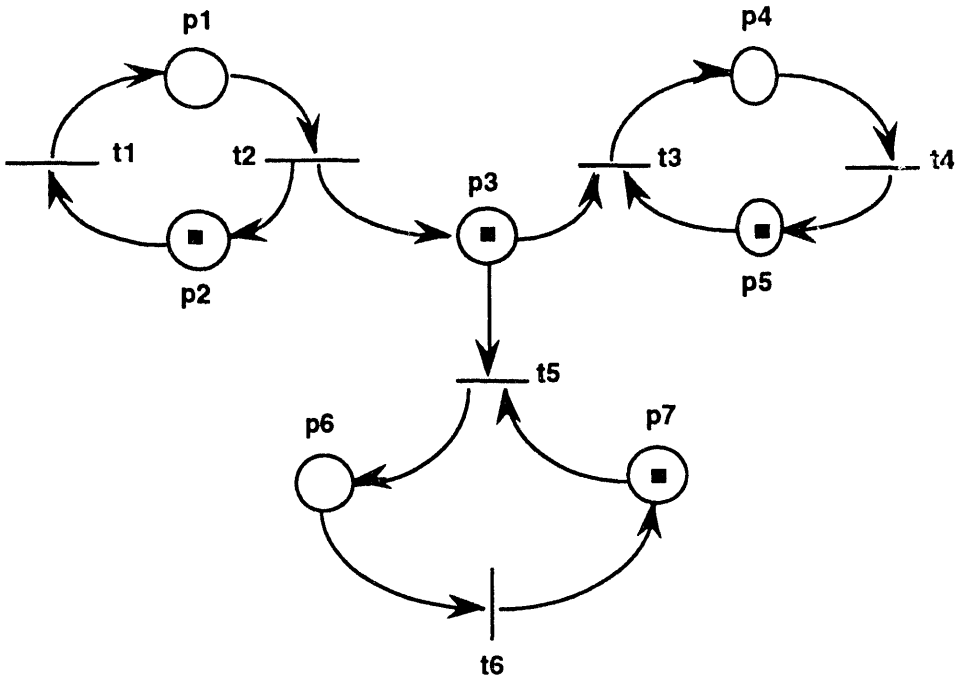


Figure A-16. The Result of Firing Figure A-15

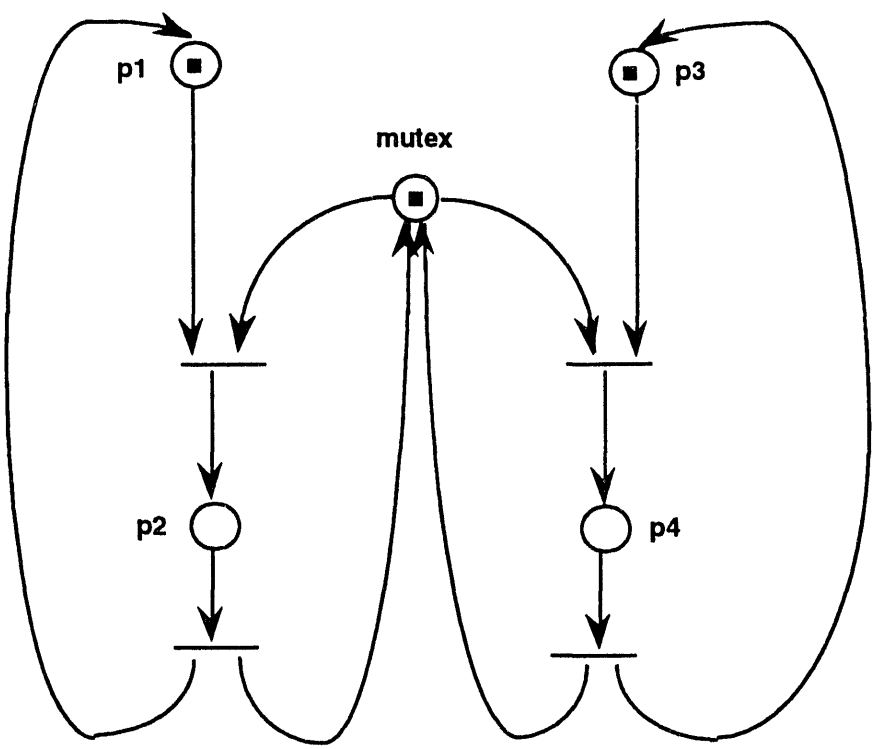


Figure A-17. A Petri Net for the Mutual Exclusion Problem

Finally, Figure A-18 shows an example of a Petri net that models a computer-controlled railroad crossing gate, taken from Leveson 1987. Notice that p_1 fires as the train approaches the crossing. At this point, p_2 and p_5 are marked, so transitions t_2 and t_4 may fire. If t_2 fires first, the train is located within the crossing and the gate is still up, which is a hazard.

Petri nets can be very useful in modeling system state changes that are caused by triggering events. They can be analyzed to show the presence or absence of safety properties, such as hazardous conditions, system deadlock, or unreachable states. Concurrency control and resource allocation can be modeled, and mixed process, computer hardware, computer software, and operator components can be included in the model. Extensions exist that incorporate timing, both deterministic and stochastic.

A.3. Reliability Growth Models

The models considered in the last section all implicitly assume that the system being modeled doesn't change. If, for example, the failure rate of a component is changed by replacing it with a new model, the reliability model must be re-evaluated.

This standard reliability model is satisfactory for systems whose components remain unchanged for long periods of time, so sufficient faults occur to permit a failure rate to be determined. It does not apply to systems that are undergoing design changes. In many cases, faults in software systems are fixed as they are discovered, so there is never enough experience to directly calculate a failure rate.

From an application viewpoint, software and hardware faults are different in kind. As a general rule, all hardware faults that users see are operational or transient in nature. All application software faults, on the other hand, are design faults. When a system failure is traced to a software fault (bug), the software is repaired (the bug is fixed). In a sense, this results in a new program—certainly the failure rate has changed. As a consequence, too few faults are executed to permit a failure rate to be calculated before the program is changed.

Consider a software system S. Failures occur at execution times $t_1, t_2, \ldots, t_n$. After each failure the fault that caused the failure may be repaired; thus, there is a sequence of programs $S = S_0, S_1, \ldots, S_n$, where S_j represents a modification of S_{j-1}, $1 \le j \le n$ If the bug couldn't be found before another failure occurs, it could happen that $S_j = S_{j-1}$.

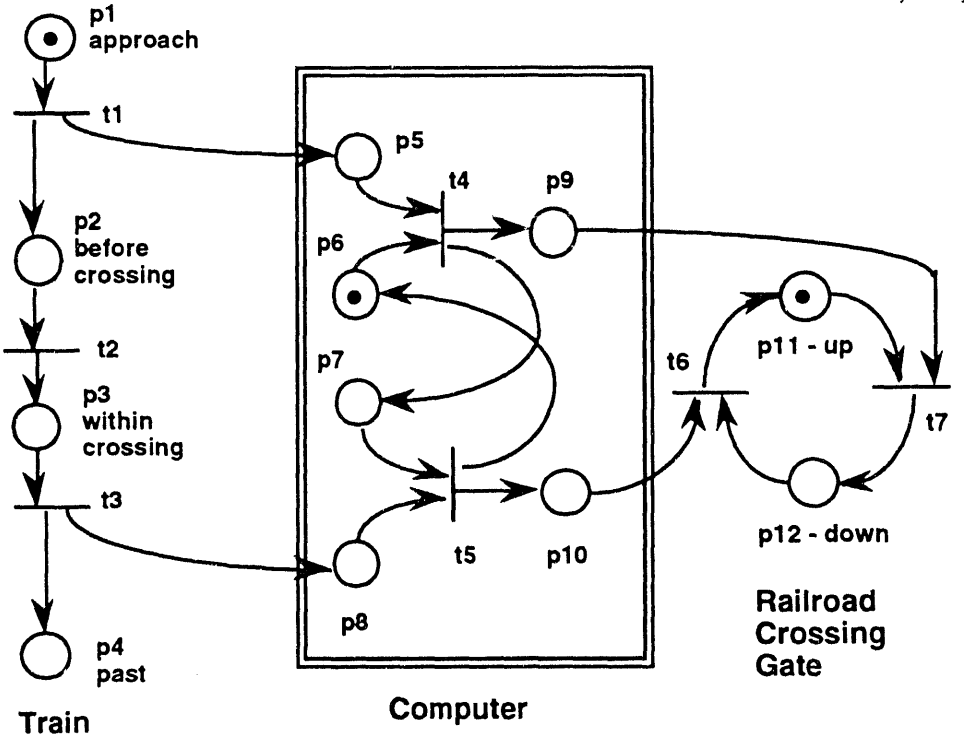


Figure A-18. Petri Net for a Railroad Crossing

A technique was developed several decades ago in the aerospace industry for modeling hardware reliability during design and test. Such a model is called a *Reliability Growth Model*. A component is tested for a period of time, during which failures occur. These failures lead to modifications to the design or manufacture of the component; the new version then goes back into test. This cycle is continued until design objectives are met. A modification of this technique seems to work quite well in modeling software (which, in a sense, never leaves the test phase).

Software reliability growth is a very active research area, and no attempt is made here to list all models that have been described. The book by Musa (Musa 1987) is an excellent starting point for additional information.

Figure A-19 shows some typical failure data (taken from Musa 1987, p. 305) of a program running in a user environment. In spite of the random fluctuations shown by the data (pluses in the figure), it seems clear that the program is getting better—the time between failures appears to be increasing. This is confirmed by the solid curve, showing a five point moving average.

A reliability growth model can be used on data such as shown in the figure to predict future failure rates from past behavior of the program, even when the program is continually changing as bugs are fixed. There are at least three important applications of an estimate of future failure rates.

- As a general rule, the testing phase of a software project continues until personnel or money are exhausted. This is not exactly a scientific way to determine when to stop testing. As an alternative, testing can continue until the predicted future failure rate has decreased to a level specified before testing begins. Indeed, this was an original motivation for the development of reliability growth models.

When used for this purpose, it is important to note that the testing environment is generally quite different from the production environment. Since testing is intended to force failures, the failure rate predicted during testing should be much higher than the actual failure rate that will be seen in production.

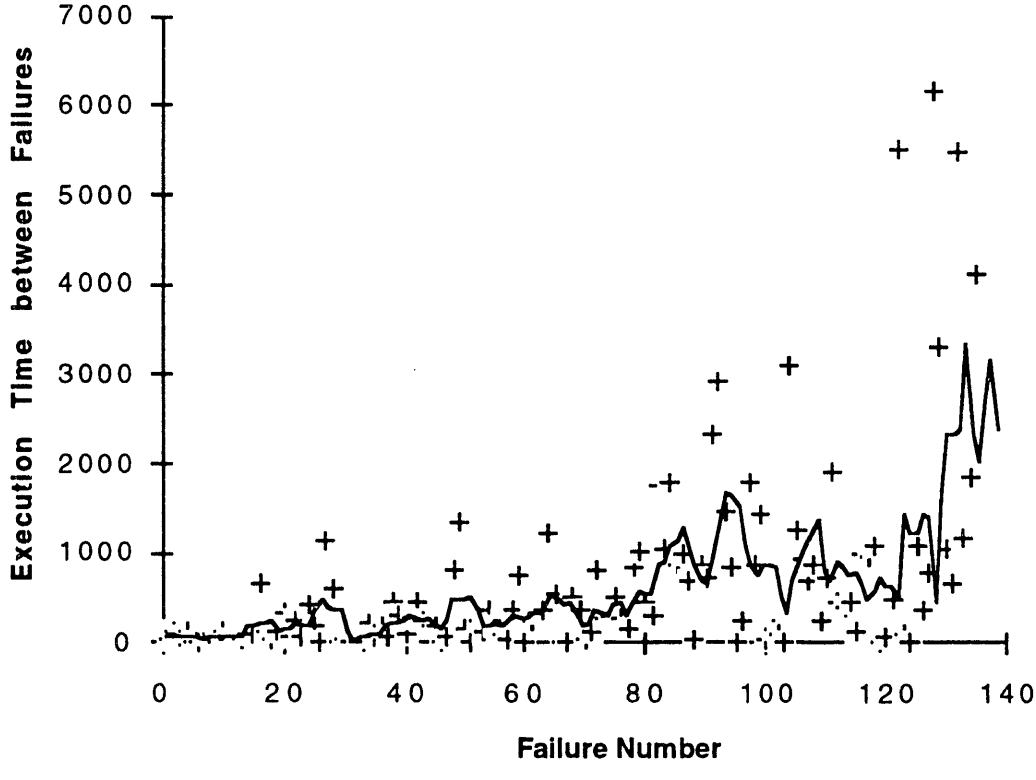


Figure A-19. Execution Time Between Successive Failures of an Actual System

- Once into production, the failure rate can be monitored. Most software is maintained and enhanced during its lifetime; monitoring failure rates can be used to judge the quality of such efforts. The process of modifying software inevitably perturbs the program's structure. Eventually, this decreases quality to the point that failures occur faster than they can be fixed. Monitoring the failure rate over time can help predict this point, in time for management to make plans to replace the program.
- Some types of real world systems have (or should have) strict legal requirements on failure rates. Nuclear reactor control systems are an example. If a control system is part of the larger system, the failure rate for the entire system will require knowledge of the failure rate of the computer portion. However, note that reliability growth modeling is not a substitute for actual analysis, review and testing. The technique can be used to predict when the software will be ready for acceptance testing, and can be used to monitor the progress up to that state. Acceptance testing will still be required if an accurate reliability figure is wanted.

There is an important limitation to this technique when very high reliability is required. As Butler points out, reliability growth techniques cannot be used if failure rates of less than about 10^{-4} failures per hour are required (Butler 1991). For instance, six programs were examined using data derived from reliability growth testing. These programs would require from 20–65 years of testing to demonstrate failure rates of 10^{-11}.

A great variety of reliability growth models have been developed. These vary according to the basic assumptions of the specific model; for example, the functional form of the failure intensity. Choice of a specific model will depend, of course, on the particular objectives of the modeling effort. Once this is done, and failure data is collected over time, the model can be used to calculate a point or interval estimate of the failure rate. This is done periodically—say, after every tenth failure.

In the remainder of this section, a few of the early software reliability growth models will be described. See Musa 1987 for more information.

A.3.1. Duane Model

The original reliability growth model proposed by Duane in 1964 suggests that the failure rate at time t can be given by $\lambda(t) = \alpha \cdot \beta \cdot t^{\beta-1}$ (Healey 1987). Knowing the times t_i that the first m failures occur permits maximum likelihood estimates of α and β to be calculated:

$$\beta = m \div \sum_{i=1}^{m-1} \ln(t_m / t_i)$$

$$\alpha = \frac{m}{t_m^{\beta}}$$

Healey pointed out that the Duane model is sensitive to early failures, and suggested a recursive procedure:

$$\gamma_4 = 0.25 \cdot (t_1 + t_2 + t_3 + t_4)$$

$$\gamma_m = (1 - w) \cdot \gamma_{m-1} + w \cdot (t_m - t_{m-1}),$$

for $m > 4$,

yielding an estimate of the failure rate as

$$\gamma(t_m) = \frac{1}{\gamma_m}$$

Healey recommends a weight of $w = 0.25$.

Using these two methods of calculating a failure rate (there are others) on the data of Figure A-19 gives the estimates of failure rate shown in Table A-1.

A.3.2. Musa Model

This model, developed by Musa in 1975, begins by assuming that all software faults are equally likely to occur and are statistically independent. After each failure the cause is determined and the software is repaired. Execution does not begin again until after this has happened. Since execution time is the time parameter being used, this means that bug fixing in the model happens instantaneously. It is assumed that no new bugs are added during the repair process and that all bugs are equally likely to occur.

Consequently, the failure rate takes on an exponential form:

$$\lambda(t) = \alpha \cdot n \cdot e^{-\alpha t}$$

where the software originally had n bugs and α is a parameter relating to the failure rate of a single fault. Integrating gives the number of bugs found by time t:

$$m(t) = n \cdot (1 - e^{-\alpha t})$$

Table A-1. Failure Rate Calculation

Failure Number	Time (hh:mm:ss)	Duane	Healey
10	9:31	0.014421	0.015750
20	33:06	0.006730	0.006881
30	1:24:09	0.003369	0.002903
40	1:46:20	0.004079	0.004414
50	2:48:09	0.003004	0.001733
60	3:29:19	0.003028	0.002692
70	4:29:45	0.002726	0.002727
80	5:42:47	0.002409	0.001711
90	8:09:21	0.001734	0.001325
100	11:40:15	0.001236	0.000885
110	13:43:36	0.001168	0.001286
120	15:41:25	0.001133	0.001615
130	20:39:24	0.000876	0.000407

The parameters can be estimated in standard ways after a number of bugs have been found and corrected.

This model has been reported to give good results. The primary difficulty is the assumption that all bugs are equally likely to occur. This seems unreasonable. In practice some bugs seem to occur much more often than others; indeed the most common bugs will normally be seen and fixed first. This line of reasoning gave rise to the next model.

A.3.3. Littlewood Model

Suppose that the program has n faults when testing begins. Each of these faults will cause a failure after some period of time that is distributed exponentially and independently from any other fault. Instantaneous debugging is assumed. Assume that failures occur at times $t_1, t_2, \ldots, t_i$. After the i^{th} bug has been found, Littlewood gives a failure rate of

$$\lambda(t) = \frac{(n-i)\cdot\alpha}{\beta + t_i + t} \qquad \text{for } t_i < t < t_{i+1}$$

where α and β are parameters to be determined by maximum likelihood estimates. The expected number of failures by time t is given by

$$m(t) = (n-i)\cdot\alpha\cdot\ln\left(\frac{\beta + t_i + t}{\beta + t_i}\right)$$

Notice that this function has a step after each failure; the failure rate decreases abruptly after each bug is found. This is supposed to reflect the fact that more common bugs are found first. It has been reported that this model gives good results.

A.3.4. Musa-Okumoto Model

This model (also known as the logarithmic Poisson execution time model) is like the Littlewood model in that the failure rate function decreases exponentially with the number of failures experienced. The basic assumptions are the same as those for the Littlewood model. The result is:

$$\lambda(t) = \frac{\alpha}{\alpha\cdot\beta\cdot t + 1}$$

where α is the initial failure rate and β is a parameter. The expected number of failures by time t is given by

$$m(t) = \frac{1}{\beta} \cdot \ln(\alpha \cdot \beta \cdot t + 1)$$

As is indicated by the large variety of models and the number of technical papers that are being published, software reliability models are not as mature as the models considered in Appendix A.2. They do appear very promising, and the need is certainly great. It appears that the models that have been developed so far are well worth using as part of the software development process, provided that they are not used to claim that the software system is sufficiently reliable for a safety-related application.

One problem with all the models discussed is that they assume the number of faults in the software is fixed, although the number is not necessarily known. In a protection system, the environment may affect this assumption, since things that were not faults at one time may become faults due to changes in the environment.

There is also some kind of assumption made about demands. For a protection system, the steady state demand requires no action on the part of the system. It is only when things go wrong that important parts of the code are actuated—parts that were never used in normal operation. So the models need to be augmented to take this difference in execution frequency into account.

Different kinds of models can be used to solve parts of a large problem. For example, a complex problem involving repairable and nonrepairable components might be modeled using a Markov technique. The nonrepairable portion might consist of a submodel analyzed by a reliability block diagram. The application program portion of that could, in turn, by modeled by a reliability growth model. This use of combinations of modeling techniques is a powerful method of analysis.

REFERENCES

Standards

ANS 7-4.3.2. "Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations," ANS 7-4.3.2 draft 7 (1992).

ANS 8.3. "Criticality Accident Alarm System," ANSI/ANS 8.3 (1986).

ANS 10.4. "Guidelines for the Verification and Validation of Scientific and Engineering Computer Programs for the Nuclear Industry," ANSI/ANS 10.4 (1987).

GFAS 250. "Software Development Standard for the German Federal Armed Forces: V-Model, Software Lifecycle Process Model," General Directive 250 (February 1991).

IEC 880. "Software for Computers in the Safety Systems of Nuclear Power Stations," IEC Publication 880 (1986).

IEEE 603. "IEEE Standard Criteria for Safety Systems for Nuclear Power Generating Stations," IEEE 603 (1991).

IEEE 610.12. "IEEE Standard Glossary of Software Engineering Terminology," IEEE 610.12 (1990).

IEEE 730.1. "IEEE Standard for Quality Assurance Plans," ANSI/IEEE 730.1 (1989)

IEEE 730.2. :IEEE Guide to Software Quality Assurance Planning,: IEEE Draft 730.2 (1993).

IEEE 828. "IEEE Standard for Software Configuration Management Plans," ANSI/IEEE 828 (1983).

IEEE 829. "IEEE Standard for Software Test Documentation, ANSI/IEEE 829 (1983).

IEEE 830. "IEEE Guide to Software Requirements Specification," ANSI/IEEE 830 (1984).

IEEE 982.1. "IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE 982.1 (1988).

IEEE 982.2. "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," IEEE 982.2 (1988).

IEEE 1008. "IEEE Standard for Software Unit Testing," ANSI/IEEE 1008 (1987).

IEEE 1012. "IEEE Standard for Software Verification and Validation Plans," ANSI/IEEE 1012 (1986).

IEEE 1016. "IEEE Recommended Practice for Software Design Descriptions," ANSI/IEEE 1016 (1987).

IEEE 1028. "IEEE Standard for Software Reviews and Audits," IEEE 1028 (1988).

IEEE 1042. "IEEE Guide to Software Configuration Management," ANSI/IEEE 1042 (1987).

IEEE 1044. "IEEE Standard Classification for Software Errors, Faults and Failures," IEEE 1044 draft (198.

IEEE 1058.1. "IEEE Standard for Software Project Management Plans," ANSI/IEEE Std 1058.1 (1987).

IEEE 1063. "IEEE Standard for Software User Documentation," ANSI/IEEE 1063 (1987).

IEEE 1074. "IEEE Standard for Developing Life Cycle Processes," IEEE 1074 (1991).

IEEE 1228. "IEEE Standard for Software Safety Plans," IEEE Draft 1228 (1993).

IEEE 1298. "IEEE Standard Software Quality Management System, Part 1: Requirements," IEEE 1298 (1992) and AS 3563.1 (1991).

IEEE C37.1. "IEEE Standard Definition, Specification and Analysis of Systems Used for Supervisory Control, Data Acquisition and Automatic Control," ANSI/IEEE C37.1 (1987).

ISA S5.1. "Instrumentation Symbols and Identification," ANSI/ISA Std. S5.1 (1984).

MOD 1989. "Requirements for the Procurement of Safety Critical Software in Defense Equipment," Draft Defense Standard 00-55, Ministry of Defence, Glasgow (May 1989) (superseded by a later draft).

Books, Articles, and Reports

Abbott 1990. Russell J. Abbott, "Resourceful Systems for Fault Tolerance, Reliability and Safety," *Computing Surveys* **22**, 1 (March 1990), 35–68.

Aldemir 1987. Tunc Aldemir, "Computer-Assisted Markov Failure Modeling of Process Control Systems," *IEEE Trans. Rel.* **36**, 1 (April 1987), 133–144.

Altschul 1987. Roberto E. Altschul and Phyllis M. Nagel, "The Efficient Simulation of Phased Fault Trees," *Proc Annual Rel. and Maint. Symp.* (January 1987), 292–295.

Amer 1987. Hassanein Amer and Edward J. McCluskey, "Weighted Coverage in Fault-Tolerant Systems," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1987), 187–191.

Anderson 1983. Thomas Anderson and John C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Trans. Soft. Eng.* **9**, 3 (May 1983), 355–364.

Anderson 1985. Thomas Anderson, "Fault Tolerant Computing," in *Resilient Computing Systems*, T. Anderson, ed, Wiley (1985), 1–10.

Atkinson 1991. Will Atkinson and Jim Cunningham, "Proving Properties of a Safety-Critical System," *Soft. Eng. J.* **6**, 2 (March 1991), 41–50.

Avizenis 1985. A. Avirienis, P. Gunningberg, J. P. J. Kelly, R. T. Lyu, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, "Software Fault-Tolerance by Design Diversity DEDIX: A Tool for Experiments," *Safety of Computer Control Systems (SafeComp)* (October 1985), 173–178.

Babich 1986. Wayne A. Babich, *Software Configuration Management, Coordination for Team Productivity*, Addison-Wesley (1986).

Barter 1993. Robert Barter and Lin Zucconi, "Verification and Validation Techniques and Auditing Criteria for Critical System-Control Software," Lawrence Livermore National Laboratory, Livermore, CA (1993).

Bavuso 1987. S. J. Bavuso, J. B. Dugan, K. S. Trivedi, E. M. Rothmann and W. E. Smith, "Analysis of Typical Fault-Tolerant Architectures Using HARP," *IEEE Trans. Rel.* **36**, 2 (June 1987), 176–185.

Belli 1990. Fevzi Belli and Plotr Jedrzejowicz, "Fault-Tolerant Programs and their Reliability," *IEEE Trans. Rel.* **39**, 2 (June 1990), 184–102.

Berg 1986. Ulf Berg, "RELTREE—A Fault Tree Code for Personal Computers," in *Reliability Technology—Theory and Applications*, J. Moltoft and F. Jensen, ed, Elsevier (1986), 433–440.

Berg 1987. Menachem Berg and Israel Koren, "On Switching Policies for Modular Redundancy Fault-Tolerant Computing Systems," *IEEE Trans. Comp.* **36**, 9 (September 1987), 1052–1062.

Bihari 1988. Thomas E. Bihari and Karsten Schwan, "A Comparison of Four Adaptation Algorithms for Increasing the Reliability of Real-Time Software," *Proc. Real-Time Systems Symp.* (December 1988), 232–241.

Bishop 1985. P. G. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahil, J. Lahti, and S. Yoshimura, "Project On Diverse Software—An Experiment in Software Reliability," *Safety of Computer Control Systems (SafeComp)* (October 1985), 153–158.

Bishop 1990. P. G. Bishop, ed, *Dependability of Critical Computer Systems 3: Techniques Directory*, Elsevier (1990). Guidelines produced by the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7).

Bloomfield 1991. R. E. Bloomfield, P. K. D. Froome and B. Q. Monahan, "Formal Methods in the Production and Assessment of Safety Critical Software," *Rel. Eng. and System Safety* **32**, 1 (1991), 51–66.

Bobbio 1986. Andrea Bobbio and Kishor S. Trivedi, "An Aggregation Technique for the Transient Analysis of Stiff Markov Chains," *IEEE Trans. Comp.* **35**, 9 (September 1986), 803–814.

Boehm 1988. Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer* (May 1988), 61–72.

Bowman 1991. William C. Bowman, Glenn H. Archinoff, Vijay M. Raina, David R. Tremaine and Nancy G. Leveson, "An Application of Fault Tree Analysis to Safety Critical Software at Ontario Hydro," *Probabilistic Safety Assessment and Management*, G. Apostolakis, ed (1991), 363–368.

Brilliant 1989. Susan S. Brilliant, John C. Knight and Nancy G. Leveson, "The Consistent Comparison Problem in N-Version Software," *IEEE Trans. Soft. Eng.* **15**, 11 (November 1989), 1481–1485.

Brilliant 1990. Susan S. Brilliant, John C. Knight and Nancy G. Leveson, "Analysis of Faults in an N-Version Software Experiment," *IEEE Trans. Soft. Eng.* **16**, 2 (February 1990), 238–247.

Butler 1991. Ricky W. Butler and George Finelli, "The Infeasibility of Experimental Quantification of Life-Critical Software Reliability," *ACM SIGSOFT Conf. on Soft. for Critical Systems* (December 1991), 66–76.

Cha 1986. Sung D. Cha, "A Recovery Block Model and its Analysis," *Safety of Computer Control Systems (SafeComp)* (October 1986), 21–26.

Chae 1986. Kyung C. Chae and Gordon M. Clark, "System Reliability in the Presence of Common-Cause Failures," *IEEE Trans. Rel.* **35**, 1 (April 1986), 32–35.

Cheung 1980. Roger C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Trans. Soft. Eng.* **6**, 2 (March 1980), 118–125.

Connolly 1989. Brian Connolly, "Software Safety Goal Verification Using Fault Tree Techniques: A Critically Ill Patient Monitor Example," *Computer Assurance (Compass)* (1989), 18–21.

Date 1983. Christopher J. Date, *An Introduction to Database Systems*, vol. 2, Addison-Wesley (1983).

Dhillon 1983. Balbir S. Dhillon, *Reliability Engineering in Systems Design and Operation*, Van Nostrand (1983).

Dhillon 1987. Balbir S. Dhillon, *Reliability in Computer System Design*, Ablex Pub. Co. (1987).

EPRI 1990. *Advanced Light Water Reactor Utility Requirements Document. Volume III, ALWR Passive Plant. Chapter 10, Man-Machine Interface Systems*, Electric Power Research Institute, Palo Alto, CA (1990).

Evans 1983. Michael W. Evans, Pamela Piazza and James B. Dolkas, *Principles of Productive Software Management*, John Wiley (1983).

Evans 1987. Michael W. Evans and John J. Marciniak, *Software Quality Assurance and Management*, Wiley (1987).

Feo 1986. T. Feo, "PAFT F77, Program for the Analysis of Fault Trees," *IEEE Trans. Rel.* **35**, 1 (April 1986), 48–50.

Frankel 1984. Ernst G. Frankel, *Systems Reliability and Risk Analysis*, Martinus Nijhoff (1984).

Geist 1986. Robert M. Geist, Mark Smotherman, Kishor Trivedi, and Joanne B. Dugan, "The Reliability of Life-Critical Computer Systems," *Acta Informatica* **23**, 6 (1986), 621–642.

Goyal 1986. A. Goyal, W. C. Carter, E. de Souza, E. Silva and S. S. Lavenberg, "The System Availability Estimator," *IEEE Annual Int'l Symp. on Fault-Tolerant Computer Systems* (July 1986), 84–89.

Guarro 1991. S. B. Guarro, J. S. Wu, G. E. Apostolakis and M. Yau, "Embedded System Reliability and Safety Analysis in the UCLA ESSAE Project," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, ed (1991), 383–388.

Hansen 1989. Mark D. Hansen, "Survey of Available Software-Safety Analysis Techniques," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 46–49.

Hatley 1987. Derek J. Hatley and Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House (1987).

Healy 1987. John Healy, "A Simple Procedure for Reliability Growth Modeling," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1987), 171–175.

Henley 1985. E. J. Henley and H. Kumamoto, *Designing for Reliability and Safety Control*, Prentice Hall (1985).

Hura 1988. G. S. Hura and J. W. Atwood, "The Use of Petri Nets to Analyze Coherent Fault Trees," *IEEE Trans. Rel.* **37**, 5 (December 1988), 469–473.

Johnson 1988. A. M. Johnson and M. Malek, "Survey of Software Tools for Evaluating Reliability, Availability and Serviceability," *ACM Comp. Surv.* **20**, 4 (December 1988), 227–269.

Jorgensen 1989. Paul C. Jorgensen, "Using Petri Net Theory to Analyze Software Safety Case Studies," *Computer Assurance (Compass)* (1989), 22–25.

Kandel 1988. Abraham Kandel and Eitan Avni, *Engineering Risk and Hazard Assessment*, CRC Press (1988), 2 volumes.

Kara-Zaitri 1991. Chakib Kara-Zaitri, Alfred Z. Keller, Imre Barody and Paulo V. Fleming, "An Improved FMEA Methodology," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 248–252.

Kelly 1986. John P. J. Kelly, A. Avizienis, B. T. Ulery, B. J. Swain, R. T. Lyu, A. Tai and K. S. Tso, "Multi-Version Software Development," *Safety of Computer Control Systems (SafeComp)* (October 1986), 43–49.

Kim 1989. K. H. Kim and Howard O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications," *IEEE Trans. Comp.* **38**, 5 (May 1989), 626–636.

Knight 1985. John C. Knight and Nancy G. Leveson, "Correlated Failures in Multi-Version Software," *Safety of Computer Control Systems (SafeComp)* (October 1985), 159–165.

Knight 1990. John C. Knight and Nancy G. Leveson, "A Reply to the Criticisms of the Knight & Leveson Experiment," *Soft. Eng. Notes* **15**, 1 (January 1990), 24–35.

Knight 1991. John C. Knight and P. E. Ammann, "Design Fault Tolerance," *Rel. Eng. and System Safety* **32**, 1 (1991), 25–49.

Kohda 1991. Takehisa Kohda and Koichi Inoue, "A Petri Net Approach to Probabilistic Safety Assessment for Obtaining Event Sequences from Component Models," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, ed (1991), 729–734.

Kopetz 1985. H. Kopetz, "Resilient Real-Time Systems," in *Resilient Computer Systems*, T. Anderson, ed, Wiley (1985), 91–101.

Krishna 1987. C. M. Krishna, Kang G. Shin and Inderpal S. Bhandari, "Processor Tradeoffs in Distributed Real-Time Systems," *IEEE Trans. Comp.* **36**, 9 (September 1987), 1030–1040.

Laprie 1985. Jean-Claude Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Annual Int'l Symp. on Fault-Tolerant Computing* (1985), 2–11.

Laprie 1990. Jean-Claude Laprie, Jean Arlat, Christian Béounes and Karama Kanoun, "Definition and Analysis of Hardware- and Software-Fault Tolerant Architectures," *IEEE Computer* **23**, 7 (July 1990), 39–51.

Lee 1992. H. Grady Lee and Paul J. Senger, *Software Safety Analysis*, Vitro Corp. (July 15, 1992).

Leveson 1983. Nancy G. Leveson and Peter R. Harvey, "Analyzing Software Safety," *IEEE Trans. Soft. Eng.* **9**, 5 (September 1983), 569–579.

Leveson 1987. Nancy G. Leveson and Janice L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Trans. Soft. Eng.* **13**, 3 (March 1987), 386–397.

Leveson 1990. Nancy G. Leveson, S. S. Cha, John C. Knight and T. J. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study," *IEEE Trans. Soft. Eng.* **16**, 4 (April 1990), 432–443.

Leveson 1991. Nancy G. Leveson, "Safety," in *Aerospace Software Engineering*, Christine Anderson and Merlin Dorfman, ed, AIAA (1991), 319–336.

Linger 1979. Richard Linger, Harlan D. Mills and Bernard I. Witt, *Structured Programming, Theory and Practice*, Addison-Wesley (1979).

Lloyd 1977. D. K. Lloyd and M. Lipow, *Reliability: Management, Methods and Mathematics*, Prentice-Hall (1977).

Macro 1990. Allen Macro, *Software Engineering Concepts and Management*, Prentice-Hall (1990).

Mainini 1990. Maria T. Mainini and Luc Billot, "PERFIDE: An Environment for Evaluation and Monitoring of Software Reliability Metrics during the Test Phase," *Soft. Eng. J.* **5**, 1 (January 1990), 27–32.

Maxion 1986. R. A. Maxion, "Toward Fault-Tolerant User Interfaces," *Safety of Computer Control Systems (SafeComp)* (October 1986), 117–122.

Maxion 1987. R. A. Maxion, D. P. Siewiorek and S. A. Elkind, "Techniques and Architectures for Fault-Tolerant Computing," *Annual Review Computer Science* (1987), 469–520.

McCall 1977. James A. McCall, Paul K. Richards and Gene F. Walters, *Factors in Software Quality*, RADC-TR-77-369, Rome Air Development Center (November 1977).

McCormick 1981. N. J. McCormick, *Reliability and Risk Analysis*, Academic Press (1981).

Mulazzani 1985. M. Muiazzani, "Reliability Versus Safety," *Safety of Computer Control Systems (SafeComp)* (October 1985), 141–145.

Musa 1987. John D. Musa, Anthony Iannino and Kazuhira Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw Hill (1987).

Nelson 1987. Victor A. Nelson and Bill D. Carroll, "Introduction to Fault-Tolerant Computing," in *Tutorial: Fault-Tolerant Computing*, Nelson and Carroll, ed (1987), 1–4.

Nelson 1990. Victor P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts," *IEEE Comp.* **23**, 7 (July 1990), 19–25.

Ostroff 1989. J. S. Ostroff, "Verifying Finite State Real-Time Discrete Event Processes," *Proc. Int'l Conf. Dist. Sys.* (1989), 207–216.

Pages 1986. A. Pages and M. Gondran, *System Reliability Evaluation and Prediction in Engineering*, Springer-Verlag (1986).

Peterson 1977. J. L. Peterson, "Petri Nets," *ACM Comp. Surv.* **9**, 3 (September 1977), 223–252.

Phillis 1986. Yannis A. Phillis, Henry D'Angelo and Gordon C. Saussy, "Analysis of Series–Parallel Production Networks Without Buffers," *IEEE Trans. Rel.* **35**, 2 (June 1986), 179–184.

Pradhan 1986. D. J. Pradhan, ed, *Fault Tolerant Computing: Theory and Practice*, Prentice-Hall (1986).

Preckshot 1992. George G. Preckshot, "Real-Time Systems Complexity and Scalability," Lawrence Livermore National Laboratory, Livermore, CA (1992).

Preckshot 1992a. George G. Preckshot, "Communication Systems in Nuclear Power Plants," Lawrence Livermore National Laboratory, Livermore, CA (1992).

Pressman 1987. Roger S. Pressman, *Software Engineering: A Practitioner's Approach*, 2nd ed, McGraw-Hill (1987).

Pucci 1990. Geppino Pucci, "On the Modeling and Testing of Recovery Block Structures," *Int'l Symp. Fault-Tolerant Comp.* (June 1990), 356–363.

Purtilo 1991. James M. Purtilo and Jankaj Jalote, "An Environment for Developing Fault-Tolerant Software," *IEEE Trans. Soft. Eng.* **17**, 2 (February 1991), 153–159.

Randell 1978. B. P. A. L. Randell and P. C. Treleaven, "Reliability Issues in Computing System Design," *ACM Comp. Surv.* **10**, 2 (June 1978), 123–165.

Redmill 1988. F. J. Redmill, ed, *Dependability of Critical Computer Systems 1*, Elsevier (1988). Guidelines produced by the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7).

Redmill 1989. F. J. Redmill, ed, *Dependability of Critical Computer Systems 2* Elsevier (1989). Guidelines produced by the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7).

Rumbaugh 1991. James Rumbaugh, et al, *Object Oriented Modeling and Design*, Prentice-Hall (1991).

Saglietti 1986. F. Sagietti and W. Ehrenberger, "Software Diversity—Some Considerations About its Benefits and Its Limitations," *Safety of Computer Control Systems (SafeComp)* (October 1986), 27–34.

Sahner 1987. R. A. Sahner and Kishor S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Trans. Rel.* **36**, 2 (June 1987), 186–193.

Sahner 1987a. R. A. Sahner and Kishor S. Trivedi, "Performance and Reliability Analysis Using Directed Acyclic Graphs," *IEEE Trans. Soft. Eng.* **13**, 10 (October 1987), 1105–1114.

Sauer 1981. Charles H. Sauer and K. Mani Chandy, *Computer Systems Performance Modeling*, Prentice-Hall (1981).

Shieh 1989. Yuan-Bao Shieh, Dipak Ghosal, Prasad R. Chintamaneni and Satish K. Tripathi, "Application of Petri Net Models for the Evaluation of Fault-Tolerant Techniques in Distributed Systems," *Proc Int'l Conf. Dist. Syst.* (1989), 151–159.

Shimeall 1991.Timothy J. Shimeall and Nancy G. Leveson, "An Empirical Comparison of Software Fault Tolerance and Fault Elimination," *IEEE Trans. Soft. Eng.* **17**, 2 (February 1991), 173–182.

Siegrist 1988. Kyle Siegrist, "Reliability of Systems with Markov Transfer of Control," *IEEE Trans. Soft. Eng.* **14**, 7 (July 1988), 1049–1053.

Siegrist 1988a. Kyle Siegrist, "Reliability of Systems with Markov Transfer of Control II," *IEEE Trans. Soft. Eng.* **14**, 10 (October 1988), 1478–1481.

Siewiorek 1982. D. P. Sieriorek and R. S. Swarz, *the Theory and Practice of Reliable System Design*, Digital Press (1982).

Smith 1972. David J. Smith, *Reliability Engineering*, Pitnam (1972).

Smith 1988. R. M. Smith, Kishor S. Trivedi and A. V. Ramesh, "Performability Analysis: Measures, an Algorithm, and a Case Study," *IEEE Trans. Comp.* **37**, 4 (April 1988), 406–417.

Stepney 1987. S. Stepney and S. Lord, "Formal Specification of an Access Control System," *Software—Practice and Experience* **17**, 9 (1987), 575–593.

Stiffler 1986. J. J. Stiffler, "Computer-Aided Reliability Estimation," in *Fault-Tolerant Computing, Theory and Techniques*, D. K. Pradhan, ed, Prentice Hall (1986), 633–657.

Strigini 1985. L. Strigini and A. Avizienis, "Software Fault-Tolerance and Design Diversity: Past Experience and Future Evolution," *Safety of Computer Control Systems (SafeComp)* (October 1985), 167–172.

Thomas 1993. Booker Thomas, "Testing Existing Software for Safety-Related Applications," Lawrence Livermore National Laboratory, Livermore, CA (1993).

Tso 1986. K. S. Tso, A. Avizienis and J. P. J. Kelly, "Error Recovery in Multi-Version Software," *Safety of Computer Control Systems (SafeComp)* (October 1986), 35–40.

Vincent 1988. James Vincent, Albert Waters and John Sinclair, *Software Quality Assurance. Volume I, Practice and Implementation*, Prentice-Hall (1988).

Wei 1991. Benjamin C. Wei, "A Unified Approach to Failure Mode, Effects and Criticality Analysis (FMECA)," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 260–271.

Williams 1992. Lloyd G. Williams, "Formal Methods in the Development of Safety Critical Software Systems," SERM-014-91, Software Engineering Research, Boulder, CO (April 1992).

Yourdon 1979. Edward Yourdon and Larry L. Constantine, *Structural Design, Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall (1979).

BIBLIOGRAPHY

The bibliography below contains a list of articles, books and reports that relate to software reliability and safety.

Russell J. Abbott, "Resourceful Systems for Fault Tolerance, Reliability and Safety," *Computing Surveys* **22**, 1 (March 1990), 35–68.

Abdalla A. Abdel-Ghaly, P. Y. Chan, and Bev Littlewood, "Evaluation of Competing Software Reliability Predictions," *IEEE Trans. Soft. Eng.* **12**, 9 (September 1986), 950–967.

K. K. Aggarwal, "Integration of Reliability and Capacity in Performance Measure of a Telecommunication Network," *IEEE Trans. Rel.* **34**, 2 (June 1985), 184–186.

Prathima Agrawal, "Fault Tolerance in Multiprocessor Systems Without Dedicated Redundancy," *IEEE Trans. Comp.* **37**, 3 (March 1988), 358–362.

Air Force Inspection and Safety Center, *Software System Safety*, AFISC SSH 1-1 (September 5, 1985).

Air Force Systems Command, *Software Independent Verification and Validation*, Wright-Patterson Air Force Base (May 20, 1988).

Tunc Aldemir, "Computer-Assisted Markov Failure Modeling of Process Control Systems," *IEEE Trans. Rel.* **36**, 1 (April 1987), 133–144.

Roberto E. Altschul and Phyllis M. Nagel, "The Efficient Simulation of Phased Fault Trees," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1987), 292–295.

Hassanein Amer and Edward J. McCluskey, "Weighted Coverage in Fault-Tolerant Systems," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1987), 187–191.

Nina Amla and Paul Ammann, "Using Z Specifications in Category Partition Testing," *Computer Assurance (Compass)* (June 1992), 3–10.

Paul E. Ammann and John C. Knight, "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Trans. Comp.* **37**, 4 (April 1988), 418–425.

Thomas Anderson and John C. Knight, "A Framework for Software Fault Tolerance in Real-Time Systems," *IEEE Trans. Soft. Eng.* **9**, 3 (May 1983), 355–364.

Thomas Anderson, "Fault Tolerant Computing," in *Resilient Computing Systems*, T. Anderson, Ed, Wiley (1985), 1–10.

Thomas Anderson, Peter A. Barrett, Dave N. Halliwell, and Michael R. Moulding, "Software Fault Tolerance: An Evaluation," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1502–1509.

Peter Andow, "Real Time Fault Diagnosis for Process Plants," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 461–466.

George Apostolakis, "The Concept of Probability in Safety Assessments of Technological Systems," *Science* **250** (December 7, 1990), 1359–1364.

Jean Arlat and Karama Kanoun, "Modeling and Dependability Evaluation of Safety Systems in Control and Monitoring Applications," *Safety of Computer Control Systems (Safecomp)* (October 1986), 157–164.

Jean Arlat, Karama Kanoun and Jean-Claude Laprie, "Dependability Modeling and Evaluation of Software Fault-Tolerant Systems," *IEEE Trans. Comp.* **39**, 4 (April 1990), 504–512.

Pasquale Armenise, "A Structured Approach to Program Optimization," *IEEE Trans. Soft. Eng.* **15**, 2 (February 1989), 101–108.

H.R. Aschmann, "Broadcast Remote Procedure Calls for Resilient Computation," *Safety of Computer Control Systems (Safecomp)* (October 1985), 109–115.

Noushin Ashrafi and Oded Berman, "Optimization Models for Selection of Programs, Considering Cost and Reliability," *IEEE Trans. Rel.* **41**, 2 (June 1992), 281–287.

Will Atkinson and Jim Cunningham, "Proving Properties of a Safety-Critical System," *Soft. Eng. J.* **6**, 2 (March 1991), 41–50.

Joanne Atlee and John Gannon, "State-Based Model Checking of Event-Driven System Requirements," *ACM Sigsoft Conf. On Soft. for Critical Systems* (December 1991), 16–28.

Terje Aven, "Some Considerations On Reliability Theory and Its Applications," *Reliability Engineering and System Safety* **21**, 3 (1988), 215–223.

A. Avirienis, P. Gunningberg, J. P. J. Kelly, R. T. Lyu, L. Strigini, P. J. Traverse, K. S. Tso, and U. Voges, "Software Fault-Tolerance By Design Diversity Dedix: A Tool for Experiments," *Safety of Computer Control Systems (Safecomp)* (October 1985), 173–178

S. Bagnasco, F. Manzo and F. Piazza, "Requirements and Design for a Distributed Computerized System for Safety and Control Applications," *Safety of Computer Control Systems (Safecomp)* (October 1985), 85–93.

D. S. Bai and W. Y. Yun, "Optimum Number of Errors Corrected Before Releasing a Software System," *IEEE Trans. Rel.* **37**, 1 (April 1988), 41–44.

F. Baiardi and M. Vanneschi, "Structuring Processes as a Sequence of Nested Atomic Actions," *Safety of Computer Control Systems (Safecomp)* (October 1985), 1–6.

John H. Bailey and Richard A. Kowalski, "Reliability-Growth Analysis for An Ada-Coding Process," *Proc. Annual Rel. and Maint. Symp.* (January 1992), 280–284.

C. T. Baker, "Effects of Field Service on Software Reliability," *IEEE Trans. Soft. Eng.* **14**, 2 (February 1988), 254–258.

Michael O. Ball, "Computational Complexity of Network Reliability Analysis: An Overview," *IEEE Trans. Rel.* **35**, 3 (August 1986), 230–239.

Prithviraj Banerjee and Jacob A. Abraham, "A Probabilistic Model of Algorithm-Based Fault Tolerance in Array Processors for Real-Time Systems," *Proc. Real-Time Systems Symp.*, (December 1986), 72–78.

Richard E. Barlow, Jerry B. Fussell and Nozer D. Singpurwalla, Ed, *Reliability and Fault Tree Analysis*, Siam (1974).

Lewis Bass and Daniel L. Martin, "Cost-Effective Software Safety Analysis," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 35–40.

F. B. Bastani and C. V. Ramamoorthy, "Input-Domain-Based Models for Estimating the Correctness of Process Control Programs," in Serra, A., and R. E. Barlow, Ed., *Theory of Reliability* (1984), 321–378.

Farokh B. Bastani, "On the Uncertainty in the Correctness of Computer Programs," *IEEE Trans. Soft. Eng.* **11**, 9 (September 1985), 857–864.

P. Baur and R. Lauber, "Design Verification for (Safety-Related) Software Systems," *Safety of Computer Control Systems (Safecomp)* (October 1985), 31–37.

S. J. Bavuso, J. B. Dugan, K. S. Trivedi, E. M. Rothmann and W. E. Smith, "Analysis of Typical Fault-Tolerant Architectures Using HARP," *IEEE Trans. Rel.* **36**, 2 (June 1987), 176–185.

Salvatore J. Bavuso, Joanne B. Dugan, Kishor Trivedi, Beth Rothmann and Mark Boyd, "Applications of the Hybrid Automated Reliability Predictor," NASA Technical Paper (1987).

Salvatore J. Bavuso and Joanne Bechta Dugan, "HiRel: Reliability/Availability Integrated Workstation Tool," *Proc. Annual Rel. Maint. Symp.* (January 1992), 491–500.

G. Becker and L. Camarinopoulos, "A Bayesian Estimation Method for the Failure Rate of a Possibly Correct Program," *IEEE Trans. Soft. Eng.* **16**, 11 (November 1990), 1307–1310.

Thomas Becker, "Keeping Processes Under Surveillance," *Symp. on the Rel. of Dist. Systems* (1991), 198–205.

Mohamed Belhadj, Mahbubul Hassan, and Tunc Aldemir, "The Sensitivity of Process Control System Interval Reliability to Process Dynamics—A Case Study," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 533–538.

R. Bell and S. Smith, "An Overview of IEC Draft Standard: Functional Safety of Programmable Electronic Systems," *Computer Assurance (Compass)* (June 1990), 151–163.

Fevzi Belli and Plotr Jedrzejowicz, "Fault-Tolerant Programs and Their Reliability," *IEEE Trans. Rel.* **39**, 2 (June 1990), 184–102.

Daniel Bell, Lisa Cox, Steve Jackson and Phil Schaefer, "Using Causal Reasoning for Automated Failure Modes and Effects Analysis (FMEA)," *Proc. Annual Rel. Maint. Symp.* (January 1992), 343–352.

Fevzi Belli and Piotr Jedrejowicz, "Comparative Analysis of Concurrent Fault Tolerance Techniques for Real-Time Applications," *Int'l Symp. on Soft. Rel. Eng.* (1991), 202–209.

Steve G. Belovich and Vijaya K. Konangi, "An Algorithm for the Computation of Computer Network Reliability in Linear-Time," *Proc. Phoenix Conf. Computers and Comm.* (March 1990), 655–660.

Tony Bendell, "An Overview of Collection, Analysis and Application of Reliability Data in the Process Industries," *IEEE Trans. Rel.* **37**, 2 (June 1988), 132–137.

P. A. Bennett, "Forwards to Safety Standards," *Soft. Eng. J.* **6**, 2 (March 1991), 37–40.

H. Claudio Benski and Emmanuel Cabau, "Experimental-Design Techniques in Reliability-Growth Assessment," *Proc. Annual Rel. Maint. Symp.* (January 1992), 322–326.

Ulf Berg, "RELTREE—A Fault Tree Code for Personal Computers," in *Reliability Technology—Theory and Applications*, J. Moltoft and F. Jensen, Ed, Elsevier (1986), 433–440.

Menachem Berg and Israel Koren, "On Switching Policies for Modular Redundancy Fault-Tolerant Computing Systems," *IEEE Trans. Comp.* **36**, 9 (September 1987), 1052–1062.

B. Bergman, "On Some New Reliability Importance Measures," *Safety of Computer Control Systems (Safecomp)* (October 1985), 61–64.

Bernard Berthomieu and Michel Diaz, "Modeling and Verification of Time Dependent Systems Using Petri Nets," *IEEE Trans. Soft. Eng.* **17**, 3 (March 1991), 259–273.

Thomas E. Bihari and Karsten Schwan, "A Comparison of Four Adaptation Algorithms for Increasing the Reliability of Real-Time Software," *Proc. Real-Time Systems Symp.* (December 1988), 232–241.

Brian Billard, "Improving Verification Tools," Trusted Computer Systems, Electronics Research Laboratory, Australian Defence Science and Technology Organization, Salisbury, South Australia (1990).

Peter G. Bishop, D. Esp, M. Barnes, P. Humphreys, G. Dahil, J. Lahti, and S. Yoshimura, "Project on Diverse Software—An Experiment in Software Reliability," *Safety of Computer Control Systems (Safecomp)* (October 1985), 153–158.

Peter G. Bishop, David G. Esp, Mel Barnes, Peter Humphreys, Gustav Dahill and Jaakko Lahti, "Pods—A Project on Diverse Software," *IEEE Trans. Soft. Eng.* **12**, 9 (September 1986), 929–940.

Peter G. Bishop, Ed, *Dependability of Critical Computer Systems 3: Techniques Directory*, Elsevier (1990). Guidelines Produced By the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7).

R. E. Bloomfield, "Application of Finite State Models for System Design and Reliability Assessment," *Safety of Computer Control Systems (Safecomp)* (September 1983), 23–28.

R. E. Bloomfield, P. K. D. Froome and B. Q. Monahan, "Formal Methods in the Production and Assessment of Safety Critical Software," *Rel. Eng. and System Safety* **32**, 1 (1991), 51–66.

Andrea Bobbio and Kishor S. Trivedi, "An Aggregation Technique for the Transient Analysis of Stiff Markov Chains," *IEEE Trans. Computers* **35**, 9 (September 1986), 803–814.

John B. Bowles and Venkatesh Swaminathan, "A Combined Hardware, Software and Usage Model of Network Reliability and Availability," *Proc. Phoenix Conf. Computers and Comm.* (March 1990), 649–654.

William C. Bowman, Glenn H. Archinoff, Vijay M. Raina, David R. Tremaine and Nancy G. Leveson, "An Application of Fault Tree Analysis to Safety Critical Software at Ontario Hydro," *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 363–368.

Klaus-Peter Brand and Jurgen Kopainsky, "Principles and Engineering of Process Control With Petri Nets," *IEEE Trans. Automatic Control* **33**, 2 (February 1988), 138–149.

Lionel C. Briand, Victor R. Basili and Christopher J. Hetmanski, "Providing an Empirical Basis for Optimizing the Verification and Testing Phases of Software Development," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 329–338.

Susan S. Brilliant, John C. Knight and Nancy G. Leveson, "The Consistent Comparison Problem in N-Version Software," *IEEE Trans. Soft. Eng.* **15**, 11 (November 1989), 1481–1485.

Susan S. Brilliant, John C. Knight and Nancy G. Leveson, "Analysis of Faults in an N-Version Software Experiment," *IEEE Trans. Soft. Eng.* **16**, 2 (February 1990), 238–247.

Susan S. Brilliant, John C. Knight and P. E. Ammann, "On the Performance of Software Testing Using Multiple Versions," *Int'l Symp. on Fault-Tolerant Computing* (June 1990), 408–415.

Sarah Brocklehurst, P. Y. Chan, Bev Littlewood and John Snell, "Recalibrating Software Reliability Models," *IEEE Trans. Soft. Eng.* **16**, 4 (April 1990), 458–470.

Sarah Brocklehurst and Bev Littlewood, "New Ways to Get Accurate Reliability Measures," *IEEE Software* **9**, 4 (July 1992), 34–42.

Michael L. Brown, "Software Safety for Complex Systems," *IEEE Annual Conf. of Eng. in Medicine and Biology Society* (1985), 210–216.

Michael L. Brown, "Software Systems Safety and Human Errors," *Computer Assurance (Compass) (1988)*, 19–28.

Michael J. D. Brown, "Rationale for the Development of the UK Defence Standards for Safety-Critical Computer Software," *Computer Assurance (Compass)* (June 1990), 144–150.

David B. Brown, Robert F. Roggio, James H. Cross and Carolyn L. McCreary, "An Automated Oracle for Software Testing," *IEEE Trans. Rel* **41**, 2 (June 1992), 272–279.

Manfred Broy, "Experiences With Software Specification and Verification Using LP, the Larch Proof Assistant," DEC Systems Research Center Rpt. 93.

Julia V. Bukowski and William M. Goble, "Practical Lessons for Improving Software Quality," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1990), 436–440.

Julia V. Bukowski, David A. Johnson and William M. Goble, "Software-Reliability Feedback: A Physics-of-Failure Approach," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 285–289.

A. Burns, "The HCI Component of Dependable Real-Time Systems," *Soft. Eng. J.* (July 1991), 168–174.

A. Burns, J. McDermid and J. Dobson, "On the Meaning of Safety and Security," *the Comp. J.* **35**, 1 (January 1992), 3–15.

Ricky W. Butler, "An Abstract Language for Specifying Markov Reliability Models," *IEEE Trans. Rel.* **35**, 5 (December 1986), 595–601.

Ricky W. Butler and Allan L. White, *Sure Reliability Analysis*, NASA Tech. Paper 2764 (1988).

Ricky W. Butler and Sally C. Johnson, *the Art of Fault-Tolerant System Reliability Modeling*, NASA Tech. Memo. 102623 (March 1990).

Ricky W. Butler, James L. Caldwell, and Ben L. Di Vito, "Design Strategy for a Formally Verified Reliable Computing Platform," *Computer Assurance (Compass)* (June 1991), 125–133.

Ricky W. Butler and Jon A. Sjogren, "Formal Design Verification of Digital Circuitry," *Rel. Eng. and System Safety* **32**, 1 (1991), 67–93.

Ricky W. Butler and George B. Finelli, "The Infeasibility of Experimental Quantification of Life-Critical Software Reliability," *ACM Sigsoft Conf. on Soft. for Critical Systems* (December 1991), 66–76.

Ricky W. Butler, "The Sure Approach to Reliability Analysis," *IEEE Trans. Rel.* **41**, 2 (June 1992), 210–218.

Michel Carnot, Clara DaSilva, Babak Dehbonei and Fernando Mejia, "Error-Free Software Development for Critical Systems Using the B-Methodology," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 274–281.

W. C. Carter, "Hardware Fault Tolerance," *Resilient Computing Systems*, T. Anderson (Ed), Wiley (1985), 11-63.

Joseph Caruso and David W. Desormeau, "Integrating Prior Knowledge With a Software Reliability Growth Model," *Proc IEEE Int'l Conf. Soft. Eng.* (May 1991), 238–245.

Joseph P. Cavano, "Toward High Confidence Software," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1449–1455.

Sung D. Cha, "A Recovery Block Model and Its Analysis," *Safety of Computer Control Systems (Safecomp)* (October 1986), 21–26.

Kyung C. Chae and Gordon M. Clark, "System Reliability in the Presence of Common-Cause Failures," *IEEE Trans. Rel.* **35**, 1 (April 1986), 32–35.

P. Y. Chan, "Adaptive Models," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 3–18.

P. Y. Chan, "Stochastic Treatment of the Failure Rate in Software Reliability Growth Models," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 19–29.

K. Mani Chandy, "Representing Faulty Distributed Systems as Nondeterministic Sequential Systems," *Proc. Symp. on Rel. in Dist. Soft. & Database Systems*, IEEE/ACM (March 1987), 171–173.

Chi-Ming Chen and Jose D. Ortiz, "Reliability Issues With Multiprocessor Distributed Database Systems: A Case Study," *IEEE Trans. Rel.* **38**, 1 (April 1989), 153–158.

Albert Mo Kim Cheng, James C. Browne, Aloysius K. Mok, and Rwo-Hsi Wang, "Estella: A Facility for Specifying Behavioral Constraint Assertions in Real-Time Rule-Based Systems," *Computer Assurance (Compass)* (June 1991), 107–123.

Albert Mo Kim Cheng and Chia-Hung Chen, "Efficient Response Time Bound Analysis of Real-Time Rule-Based Systems," *Computer Assurance (Compass)* (June 1992), 63–76.

H. B. Chenoweth, "Soft Failures and Reliability," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1990), 419–424.

H. B. Chenoweth, "Reliability Prediction, in the Conceptual Phase, of a Processor System With Its Embedded Software," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 416–422.

Vladimir Cherkassky, and Miroslaw Malek, "A Measure of Graceful Degradation in Parallel Computer Systems," *IEEE Trans. Rel.* **38**, 1 (April 1989), 76–81.

John C. Cherniavsky, "Validation Through Exclusion: Techniques for Ensuring Software Safety," *Computer Assurance (Compass)* (1989), 56–59.

Roger C. Cheung, "A User-Oriented Software Reliability Model," *IEEE Trans. Soft. Eng.* **6**, 2 (March 1980), 118–125.

Morey J. Chick, "Interrelationships of Problematic Components of Safety Related Automated Information Systems," *Computer Assurance (Compass)* (June 1991), 53–62.

Dong-Hae Chi, Hsin-Hui Lin and Way Kuo, "Software Reliability and Redundancy Optimization," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 41–45.

Ram Chillarege, Wei-Lun Kao, and Richard G. Condit, "Defect Type and Its Impact on the Growth Curve," *Proc IEEE Int'l Conf. Soft. Eng.* (May 1991), 246–255.

Mario D. Cin, "Availability Analysis of a Fault-Tolerant Computer System," *IEEE Trans. Rel.* **29**, 3 (August 1980), 265–268.

Kent C. Clapp, Ravishankar K. Iyer and Ytzhak Levendel, "Analysis of Large System Black-Box Test Data," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 94–103.

Douglas B. Clarkson, "Estimates for the Generalized F Distribution in Reliability/Survival Models," *Directions* **6**, 2 (1989), 2–4.

Brian Connolly, "Software Safety Goal Verification Using Fault Tree Techniques: A Critically Ill Patient Monitor Example," *Computer Assurance (Compass)* (1989), 18–21.

Donna K. Cover, "Issues Affecting the Reliability of Software-Cost Estimates," *Proc. IEEE Annual Rel. and Maint. Symposium* (1988), 195–201.

Randy Cramp, Mladen A. Vouk and Wendell Jones, "On Operational Availability of a Large Software-Based Telecommunications System," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 358–366.

Larry H. Crow, "New International Standards on Reliability Growth," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 478–480.

Attila Csenki, "Bayes Predictive Analysis of a Fundamental Software Reliability Model," *IEEE Trans. Rel.* **39**, 2 (June 1990), 177–183.

W. J. Cullyer, S. J. Goodenough and B. A. Wichmann, "The Choice of Computer Languages for Use in Safety-Critical Systems," *Soft. Eng. J.* **6**, 2 (March 1991), 51–58.

P. Allen Currit, Michael Dyer, and Harlan D. Mills, "Certifying the Reliability of Software," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 3–11.

Edward W. Czeck and Daniel P. Siewiorek, "Observations on the Effects of Fault Manifestation as a Function of Workload," *IEEE Trans. Comp.* **41**, 5 (May 1992), 559–565.

Siddhartha R. Dalal and Colin L. Mallows, "Some Graphical Aids for Deciding When to Stop Testing Software," *IEEE J. Sel. Areas in Comm.* **8**, 2 (February 1990), 169–175.

C. Dale, "Software Reliability Models," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 31–44.

Chris Dale, "The Assessment of Software Reliability," *Rel. Eng. and Syst. Safety* **34** (1991), 91–103.

B. K. Daniels, R. Bell and R. I. Wright, "Safety Integrity Assessment of Programmable Electronic Systems," *Safety of Computer Control Systems (Safecomp)* (September 1983), 1–12.

Miachael K. Daskalantonakis, "A Practical View of Software Measurement and Implementation Experiences Within Motorola," *IEEE Trans. Soft. Eng.* **18**, 11 (November 1992), 998–1010.

H. T. Daughtrey, S. H. Levinson and D. M. Kimmel, "Developing a Standard for the Qualification of Software for Safety System Applications," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 687–692.

J. W. Day, "The Reliability of the Sizewell 'B' Primary Protection System," Reactor Protection Equipment Group (January 19, 1990).

R. De Lemos, A. Saeed and T. Anderson, "A Train Set as a Case Study for the Requirements Analysis of Safety-Critical Systems," *the Comp. J.* **35**, 1 (January 1992), 30–40.

Bob De Santo, "A Methodology for Analyzing Avionics Software Safety," *Computer Assurance (Compass)* (1988), 113–118.

Edmundo De Souza, E Silva and H. Richard Gail, "Calculating Cumulative Operational Time Distributions of Repairable Computer Systems," *IEEE Trans. Computer* **35**, 4 (April 1986), 322–332.

Edmundo De Souza E Silva, and H. Richard Gail, "Calculating Availability and Performability Measures of Repairable Computer Systems Using Randomization," *J. ACM* **36**, 1 (January 1989), 171–193.

Norman Delisle and David Garlan, A *Formal Specification of an Oscilloscope*, Tech. Report Cr-88–13, Tektronix, Inc. (October 27, 1988).

Balbir S. Dhillon, and C. Singh, *Engineering Reliability: New Techniques and Applications*, Wiley (1981).

Balbir S. Dhillon, "Life Distributions," *IEEE Trans. Rel.* **30**, 5 (December 1981), 457–460.

Balbir S. Dhillon, *Systems Reliability, Maintainability and Management*, Petrocelli (1983).

Balbir S. Dhillon, *Reliability Engineering in Systems Design and Operation*, Van Nostrand (1983).

Balbir S. Dhillon, *Reliability in Computer System Design*, Ablex Pub. Co. (1987).

James H. Dobbins, "Software Safety Management," *Computer Assurance (Compass)* (1988), 108–112.

Lorenzo Donatiello and Balakrishna R. Iyer, "Analysis of a Composite Performance-Reliability Measure for Fault-Tolerant Systems," *J. ACM* **34**, 1 (January 1987), 179–199.

E. P. Doolan, "Experience With Fagan's Inspection Method," *Soft. Prac. and Exper.* **22**, 2 (February 1992), 173–182.

Thomas Downs and P. Garrone, "Some New Models of Software Testing With Performance Comparisons," *IEEE Trans. Rel.* **40**, 3 (August 1991), 322–328.

Thomas Downs and Anthony Scott, "Evaluating the Performance of Software-Reliability Models," *IEEE Trans. Rel.* **41**, 4 (December 1992), 533–538.

M. R. Drury, E. V. Walker, D. W. Wightman and A. Bendell, "Proportional Hazards Modeling in the Analysis of Computer Systems Reliability," *Rel. Eng. and System Safety* **21**, 3 (1988), 197–214.

J. T. Duane, "Learning Curve Approach to Reliability Monitoring," *IEEE Trans. Aerospace* **2**, 2 (April 1964), 563–566.

Joanne B. Dugan, Kishor S. Trivedi, Mark K. Smotherman and Robert M. Geist, "The Hybrid Automated Reliability Predictor," *J. Guidance* **9**, 3 (May-June 1986), 319–331.

Joanne B. Dugan, "On Measurement and Modeling of Computer Systems Dependability: A Dialog Among Experts," *IEEE Trans. Rel.* **39**, 4 (October 1990), 506–510.

Janet R. Dunham, "Experiments in Software Reliability: Life-Critical Applications," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 110–123.

Janet R. Dunham and George B. Finelli, "Real-Time Software Failure Characterization," *Computer Assurance (Compass)* (June 1990), 39–45.

William R. Dunn and Lloyd D. Corliss, "Software Safety: A User's Practical Perspective," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1990), 430–435.

Ivor Durham and Mary Shaw, *Specifying Reliability As a Software Attribute*, Department of Computer Science, Carnegie-Mellon University (December 6, 1982).

Michael Dyer, "Inspection Data," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 45–54.

Michael Dyer, "A Formal Approach to Software Error Removal," *J. Systems and Software* **7** (1987), 109–114.

Michael Dyer, "An Approach to Software Reliability Measurement," *Information and Software Technology* **29**, 8 (October 1987), 415–420.

Kenneth H. Eagle and Ajay S. Agarwala, "Redundancy Design Philosophy for Catastrophic Loss Protection," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 1–4.

Dave E. Eckhart and Larry D. Lee, "A Theoretical Basis for the Analysis of Multiversion Software Subject to Coincident Errors," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1511–1517.

V. N. Efanov, V. G. Krymsky and Yu. G. Krymsky, "Safety and Integrity of Large-Scale Dynamic Systems and Control Algorithm Structure: The Design Technique," *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 741–746.

Willa K. Ehrlich and Thomas J. Emerson, "Modeling Software Failures and Reliability Growth During System Testing," *Int'l. Conf. on Soft. Eng.*, (March-April 1987), 72–82.

Willa K. Ehrlich, John P. Stampfel and Jar R. Wu, "Application of Software Reliability Modeling to Product Quality and Test Process," *Proc. IEEE Int'l Conf. Soft. Safety* (March 1990), 108–116.

Willa K. Ehrlich, S. Keith Lee and Rex H. Molisani, "Applying Reliability Measurement: A Case Study," *IEEE Software* **7**, 2 (March 1990), 57–64.

Alex Elentukh and Ray Wang, "Optimizing Software Reliability Management Procee," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 34–40.

Karen E. Ellis, "Robustness of Reliability-Growth Analysis Techniques," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 303–315.

D. E. Embrey, "Human Reliability," in Serra, A., and R. E. Barlow, Ed, *Theory of Reliability* (1984), 465–490.

Max Engelhardt and Lee J. Bain, "On the Mean Time Between Failures for Repairable Systems," *IEEE Trans. Rel.* **35**, 4 (October 1986), 419–422.

William W. Everett, "Software Reliability Measurement," *IEEE J. Sel. Areas in Comm.* **8**, 2 (February 1990), 247–252.

William W. Everett, "An 'Extended Execution Time' Software Reliability Model," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 4–13.

T. Feo, "Paft F77, Program for the Analysis of Fault Trees," *IEEE Trans. Rel.* **35**, 1 (April 1986), 48–50.

K. C. Ferrara, S. J. Keene and C. Lane, "Software Reliability From a System Perspective," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 332–336.

George B. Finelli, "NASA Software Failure Characterization Experiments," *Rel. Eng. and System Safety* **32**, 1 (1991), 155–169.

J. M. Finkelstein, "A Logarithmic Reliability-Growth Model for Single-Mission Systems," *IEEE Trans. Rel.* **32**, 5 (December 1983), 508–511.

Gregory G. Finn, "Reducing the Vulnerability of Dynamic Computer Networks," ISI Research Report 88–201, Information Sciences Institute, University of Southern California (June 1988).

Radu A. Florescu, "A Lower Bound for System Availability Computation," *IEEE Trans. Rel.* **35**, 5 (December 1986), 619–620.

Maurice Forrest, "Software Safety," *Hazard Prevention* (July/September 1988), 20–21.

Ernst G. Frankel, *Systems Reliability and Risk Analysis*, Martinus Nijhoff (1984).

Matthew K. Franklin and Armen Gabrielian, "A Transformational Method for Verifying Safety Properties in Real-Time Systems," *Proc. IEEE Real Time Systems Symp.* (December 1989), 112–123.

Phyllis G. Frankl and Elaine J. Weyuker, "Assessing the Fault-Detecting Ability of Testing Methods," *ACM Sigsoft Conf. on Soft. for Critical Systems* (December 1991), 77–91.

James W. Freeman and Richard B. Neely, "A Structured Approach to Code Correspondence Analysis," *Computer Assurance (Compass)* (June 1990), 109–116.

K. Frehauf and H. Sandmayr, "Quality of the Software Development Process," *Safety of Computer Control Systems (Safecomp)* (September 1983), 145–151.

G. Frewin, "Program and Process Property Models," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 55–64

Michael Friedman, "Modeling the Penalty Costs of Software Failure," *Proc Annual Rel. and Maint. Symp.* (1987), 359–363.

Michael A. Friedman and Phuong Tran, "Reliability Techniques for Combined Hardware/Software Systems," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 290–293.

Michael O. Fryer, "Risk Assessment of Computer Controlled Systems," *IEEE Trans. Soft. Eng.* **11**, 1 (January 1985), 125–129.

Eiji Fujiwara and Dhiraj K. Pradhan, "Error Control Coding in Computers," *IEEE Computer* **23**, 7 (July 1990), 63–72.

R. Fullwood, R. Lofaro and P. Samanta, "Reliability Assurance for Regulation of Advanced Reactors," *IEEE Trans. Nuclear Sci.* **39**, 5 (October 1992), 1357–1362.

A. Gana, "Software Reliability Experience in the 5ESS U.S. Project," *Second Int'l Conf. on Soft. Quality* (October 1992), 210–214.

Olivier Gaudoin and Jean-Louis Soler, "Statistical Analysis of the Geometric De-Eutrophication Software-Reliability Model," *IEEE Trans. Rel.* **41**, 4 (December 1992), 518–524.

Olivier Gaudoin, "Optimal Properties of the Laplace Trent Test for Software-Reliability Models," *IEEE Trans. Rel.* **41**, 4 (December 1992), 525–532.

Robert M. Geist and Kishor S. Trivedi, "Ultrahigh Reliability Prediction for Fault Tolerant Computer Systems," *IEEE Trans. Comp.* **32**, 12 (December 1983), 1118–1127.

Robert M. Geist, Mark Smotherman, Kishor Trivedi, and Joanne B. Dugan, "The Reliability of Life-Critical Computer Systems," *Acta Informatica* **23**, 6 (1986), 621–642.

Robert M. Geist and Kishor Trivedi, "Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques," *IEEE Computer* **23**, 7 (July 1990), 52–61.

Robert Geist, A. Jefferson Offutt and Frederick C. Harris, "Estimation and Enhancement of Real-Time Software Reliability Through Mutation Analysis," *IEEE Trans. Comp.* **41**, 5 (May 1992), 550–557.

Erol Gelenbe, David Finkel, and Satish K. Tripathi, "Availability of a Distributed Computer System With Failures," *Acta Informatica* **23**, 6 (1986), 643–655.

Erol Gelenbe and Marisela Hernández, "Enhanced Availability of Transaction Oriented Systems Using Failure Tests," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 342–350.

General Accounting Office, "Embedded Computer Systems: Significant Software Problems on C-17 Must Be Addressed," Report to the Chairman, Subcommittee on Legislation and National Security, Committee on Government Operations, House of Representatives (May 1992).

George L. Gentzler and Nelson M. Andrews, "Data Stability in An Application of a Software Reliability Model," *IEEE J. Sel. Areas of Comm.* **8**, 2 (February 1990), 273–275.

Luther P. Gerlach and Steve Rayner, "Culture and the Common Management of Global Risks," *Practicing Anthropology* (1987), 15–18.

Carlo Ghezzi, Dino Mandrioli, Sandro Morasca and Mauro Pezzè, "A Unified High-Level Petri Net Formalism for Time-Critical Systems," *IEEE Trans. Soft. Eng.* **17**, 2 (February 1991), 161–172.

T. Giammo, "Relaxation of the Common Failure Rate Assumption in Modeling Software Reliability," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 65–79.

Amrit L. Goel, "A Summary of the Discussion on 'An Analysis of Competing Software Reliability Models," *IEEE Trans. Soft. Eng.* **6**, 5 (September 1980), 501–502.

Amrit L. Goel, "Software Reliability Models: Assumptions, Limitations and Applicability," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1411–1423.

S. J. Goldsack and A. C. W. Finkelstein, "Requirements Engineering for Real-Time Systems," *Soft. Eng. J.* (May 1991), 101–115.

Donald I. Good, "Predicting Computer Behavior," *Computer Assurance (Compass)* (1988), 75–83.

J. Gorski, "Design for Safety Using Temporal Logic," *Safety of Computer Control Systems (Safecomp)* (October 1986), 149–155.

Paul Gottfried, "Some Aspects of Reliability Growth," *IEEE Trans. Rel.* **36**, 1 (April 1987), 11–16.

Lon D. Gowen, James S. Collofello and Frank W. Calliss, "Preliminary Hazard Analysis for Safety-Critical Software Systems," *Phoenix Conf. Comp. and Comm.* (April 1992), 501–508.

A. Goyal, W. C. Carter, E. De Souza E Silva and S. S. Lavenberg, "The System Availability Estimator," *IEEE Annual Int'l Symp. on Fault-Tolerant Computer Systems* (July 1986), 84–89.

Ambuj Goyal, "A Measure of Guaranteed Availability and Its Numerical Evaluation," *IEEE Trans. Comp.* **37**, 1 (January 1988), 25–32.

Robert B. Grady, "Dissecting Software Failures," *Hewlett-Packard J.* **40**, 2 (April 1989), 57–63.

C. T. Gray, "A Framework for Modeling Software Reliability," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 81–94.

A. E. Green, *Safety Systems Reliability*, Wiley (1983).

S. B. Guarro, J. S. Wu, G. E. Apostolakis and M. Yau, "Embedded System Reliability and Safety Analysis in the UCLA ESSAE Project," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 383–388.

Vassos Hadzilacos, "A Theory of Reliability in Database Systems," *J. ACM* **35**, 1 (January 1988), 121–145.

Don N. Hagist, "Reliability Testing," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 347–349.

Fred Hall, "Computer Program Stress Testing," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1985), 256–261.

P. Hamer, "Types of Metric," in *Software Reliability State of the Art Report* **14**, 2 Pergamon Infotech (1986), 95–103.

Richard Hamlet, "Unit Testing for Software Assurance," *Computer Assurance (Compass)* (1989), 42–48.

Richard Hamlet, "Are We Testing for True Reliability?" *IEEE Software* **9**, 4 (July 1992), 21–27.

Allen L. Hankinson, "Computer Assurance: Security, Safety and Economics," *Computer Assurance (Compass)* (1989), 1–7.

Mark D. Hansen and Ronald L. Watts, "Software System Safety and Reliability," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1988), 214–217.

Mark D. Hansen, "Survey of Available Software-Safety Analysis Techniques," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 46–49.

Kirsten M. Hansen, Anders P. Ravn and Hans Rischel, "Specifying and Verifying Requirements of Real-Time Systems," *ACM Sigsoft Conf. on Soft. for Critical Systems* (December 1991), 44–54.

Hans Hansson and Bengt Jonsson, "A Framework for Reasoning About Time and Reliability," *IEEE Real Time Systems Symp.* (December 1989), 102–111.

L. N. Harris, "Reliability Prediction: A Matter of Logic," *Safety of Computer Control Systems (Safecomp)* (September 1983), 29–35.

L. N. Harris, "Software Reliability Modeling—Prospects and Perspective," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 105–118.

Xudong He, "Specifying and Verifying Real-Time Systems Using Time Petri Nets and Real-Time Temporal Logic," *Computer Assurance Conf. (Compass)* (June 1991), 135–140.

John Healy, "A Simple Procedure for Reliability Growth Modeling," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1987), 171–175.

Herbert Hecht and Myron Hecht, "Software Reliability in the System Context," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 51–58.

Herbert Hecht and Myron Hecht, "Fault-Tolerant Software," in *Fault-Tolerant Computing*, D. K. Pradhan, Ed, Prentice Hall (1986), 658–696.

David I. Heimann and W. D. Clark, "Process-Related Reliability-Growth Modeling—How and Why," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 316–321.

Martin Helander, Ed, *Handbook of Human-Computer Interaction*, North Holland (1990).

E. J. Henley and H. Kumamoto, *Designing for Reliability and Safety Control*, Prentice Hall (1985).

M. A. Hennell, "Testing for the Achievement of Software Reliability," in *Software Reliability State of the Report* **14**, 2, Pergamon Infotech (1986), 119–129.

M. A. Hennell, "Testing for the Achievement of Software Reliability," *Rel. Eng. and System Safety* **32**, 1 (1991), 119–134.

Jun Hishitani, Shigeru Yamada and Shunji Osaki, "Comparison of Two Estimation Methods of the Mean Time Interval Between Software Failures," *Proc. Phoenix Conf. on Computers and Comm.* (March 1990), 418–424.

Per Hokstad and Lars Bodsberg, "Reliability Model for Computerized Safety Systems," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 435–440.

Erik Hollnagel, "What Is a Man That He Can Be Expressed By a Number?" in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 501–506.

M. C. Hsueh, R. K. Iyer and Kishor S. Trivedi, "Performability Modeling Based on Real Data: A Case Study," *IEEE Trans. Comp.* **37**, 4 (April 1988), 478–484.

Jiandong Huang, John A. Stanovic, Don Towsley and Krithi Ramamrithan, "Experimental Evaluation of Real-Time Transaction Processing," *IEEE Real Time Systems Symp.* (December 1989), 144–153.

R. A. Humphreys, "Software—Correct Beyond Reasonable Doubt?" *Proc. Advances in Rel. Tech. Symp.* (April 1988), 236–246.

G. S. Hura and J. W. Atwood, "The Use of Petri Nets to Analyze Coherent Fault Trees," *IEEE Trans. Rel.* **37**, 5 (December 1988), 469–473.

Anthony Iannino, John D. Musa, Kazuhira Okumoto and Bev Littlewood, "Criteria for Software Reliability Model Comparisons," *IEEE Trans. Soft. Eng.* **10**, 6 (November 1984), 687–691.

Anthony Iannino and John D. Musa, "Software Reliability Engineering At AT&T," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 485–491.

Oliver C. Ibe, Richard C. Howe and Kishor S. Trivedi, "Approximate Availability Analysis of Vaxcluster Systems," *IEEE Trans. Rel.* **38**, 1 (April 1989), 146–152.

IEEE, "Software in Safety-Related Systems," Institution of Electrical Engineers and British Computer Society Joint Report (October 1989).

IEEE, IEEE Software Safety Plans Working Group, *Standard for Software Safety Plans*, Working Paper, Draft H (April 1, 1992).

Toshiyuki Inagaki, "A Mathematical Analysis of Human-Machine Interface Configurations for a Safety Monitoring System," *IEEE Trans. Rel.* **37**, 1 (April 1988), 35–40.

Infotech, *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986).

Ravishankar K. Iyer and David J. Rossetti, "Permanent CPU Errors and System Activity: Measurement and Modeling," *Proc. Real-Time Systems Symp.*, (1983), 61–72.

Ravishanker K. Iyer and Paola Velardi, "Hardware-Related Software Errors: Measurement and Analysis," *IEEE Trans. Soft. Eng.* **11**, 2 (February 1985), 223–231.

Balakrishna R. Iyer, Lorenzo Donatiello and Philip Heidelberger, "Analysis of Performability for Stochastic Models of Fault-Tolerant Systems," *IEEE Trans. Comp.* **35**, 10 (October 1986), 902–907.

Ravishanker K. Iyer and Dong Tang, "Software Reliability Measurements in the Operational Environment," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 479–484.

P. R. Jackson and B. A. White, "The Application of Fault Tolerant Techniques to a Real Time System," *Safety of Computer Control Systems (Safecomp)* (September 1983), 75–81.

Raymond Jacoby and Yoshihiro Tohma, "Parameter Value Computation By Least Square Method and Evaluation of Software Availability and Reliability At Service-Operation By the Hyper-Geometric Distribution Software Reliability Growth Model (HGDM)," *Proc IEEE Int'l Conf. Soft. Eng.* (May 1991), 226–237.

Raymond Jacoby and Kaori Masuzawa, "Test Coverage Dependent Software Reliability Estimation By the HGD Model," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 193–204.

Matthew S. Jaffe and Nancy G. Leveson, "Completeness, Robustness and Safety in Real-Time Software Requirements Specification," *Proc IEEE Int'l Conf. Soft. Eng.* (1989), 302–311.

Matthew S. Jaffe, Nancy G. Leveson, Mats P. E. Heimdahl and Bonnie E. Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems," *IEEE Trans. Soft. Eng.* **17**, 3 (March 1991), 241–258.

Farnam Jahanian and Aloysius K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems," *IEEE Trans. Soft. Eng.* **12**, 9 (September 1986), 890–904.

Farnam Jahanian and Ambuj Goyal, "A Formalism for Monitoring Real-Time Constraints at Run-Time," *Proc. Int'l Symp. for Fault-Tolerant Computing* (June 1990), 148–155.

Pankaj Jalote and Roy H. Campbell, "Atomic Actions for Fault-Tolerance Using CSP,." *IEEE Trans. Soft. Eng* **12**, 1 (January 1986), 59–68.

Menkae Jeng and Howard J. Siegel, "Implementation Approach and Reliability Estimation of Dynamic Redundancy Networks," *Proc. Real-Time Systems Symp.* (December 1986), 79–88.

Jet Propulsion Lab, "Feasibility Study: Formal Methods Demonstration Project for Space Applications," JPL, Johnson Space Center, Langley Research Center (October 23, 1992).

William S. Jewell, "Bayesian Extensions to a Basic Model of Software Reliability," in Serra, A., and R. E. Barlow, Ed, *Theory of Reliability* (1984), 394–404.

William S. Jewell, "Bayesian Estimation of Undetected Errors," in Serra, A., and R. E. Barlow, Ed, *Theory of Reliability* (1984), 405–425.

William S. Jewell, "A General Framework for Learning-Curve Reliability Growth Models," in Serra, A., and R. E. Barlow, Ed, *Theory of Reliability* (1984), 426–449.

William S. Jewell, "Bayesian Extensions to a Basic Model of Software Reliability," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1465–1471.

Harry Joe and N. Reid, "On the Software Reliability Models of Jelinski-Moranda and Littlewood," *IEEE Trans. Rel.* **34**, 3 (August 1985), 216–218.

Harry Joe, "Statistical Inference for General-Order Statistics and Nonhomogeneous Poisson Process Software Reliability Models," *IEEE Trans. Soft. Eng.* **15**, 11 (November 1989), 1485–1490.

A. M. Johnson and M. Malek, "Survey of Software Tools for Evaluating Reliability, Availability and Serviceability," *ACM Comp. Surv.* **20**, 4 (December 1988), 227–269.

Sally C. Johnson and Ricky W. Butler, "Design for Validation," *IEEE Aerospace and Elect. Syst. J.* **7**, 1 (January 1992), 38–43.

L. K. Jokubaitis and M. F. Wuinn, "New Army Method Is Stating and Assessing RAM Requirements," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 19–27.

Paul C. Jorgensen, "Early Detection of Requirements Specification Errors," *Computer Assurance (Compass)* (1988), 44–48.

Paul C. Jorgensen, "Using Petri Net Theory to Analyze Software Safety Case Studies," *Computer Assurance (Compass)* (1989), 22–25.

Philip R. Joslin, "Software for Reliability Testing of Computer Peripherals: A Case History," *IEEE Trans. Rel.* **35**, 3 (August 1986), 279–284.

B. D. Juhlin, "Implementing Operational Profiles to Measure System Reliability," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 286–295.

M. Kaâniche and K. Kanoun, "The Discrete Time Hyperexponential Model for Software Reliability Growth Evaluation," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 64–75.

S. H. Kan, "Modeling and Software Development Quality," *IBM Systems J.* **30**, 3 (1991), 351–362.

Abraham Kandel and Eitan Avni, *Engineering Risk and Hazard Assessment*, CRC Press (1988), 2 Volumes.

Karama Kanoun, Marta Rettelbusch De Bastos Martini and Jorge Moreira De Souza, "A Method for Software Reliability Analysis and Prediction Application to the Tropico-R Switching System," *IEEE Trans. Soft. Eng.* **17**, 4 (April 1991), 334–344.

Krishna Kant, "Performance Analysis of Real-Time Software Supporting Fault-Tolerant Operation," *IEEE Trans. Comp.* **39**, 7 (July 1990), 906–918.

P. K. Kapur and R. B. Garg, "A Software Reliability Growth Model for An Error-Removal Phenomenon," *Soft. Eng. J.* **7**, 4 (July 1992), 291–294.

Chakib Kara-Zaitri, Alfred Z. Keller, Imre Barody and Paulo V. Fleming, "An Improved FMEA Methodology," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 248–252.

A. P. Karadimce, R. R. Seban and A. L. Grnarov, "A Fast, General-Purpose Algorithm for Reliability Evaluation of Distributed Systems," *Proc. Phoenix Conf. Computers and Comm.* (March 1990), 137–146.

Nachimuthu Karunanithi, Darrell Whitley and Yashwant K. Malaiya, "Using Neural Networks in Reliability Prediction," *IEEE Software* **9**, 4 (July 1992), 53–59.

Nachimuthu Karunanithi and Y. K. Malaiya, "The Scaling Problem in Neural Networks for Software Reliability Prediction," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 76–82.

Krishna M. Kavi and U. Narayan Bhat, "Reliability Analysis of Computer Systems Using Dataflow Graph Models," *IEEE Trans. Rel.* **35**, 5 (December 1986), 529–531.

Samuel J. Keene, "Cost Effective Software Quality," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 433–437.

Samuel J. Keene, "Assuring Software Safety," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 274–279.

Karen C. Keene and Samuel J. Keene, "Concurrent Engineering Aspects of Software Development," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 51–62.

Peter A. Keiller and Douglas R. Miller, "On the Use and the Performance of Software Reliability Growth Models," *Rel. Eng. and System Safety* **32**, 1 (1991), 95–117.

John P. J. Kelly, A. Avizienis, B. T. Ulery, B. J. Swain, R.-T. Lyu, A. Tai and K.-S. Tso, "Multi-Version Software Development," *Safety of Computer Control Systems (Safecomp)* (October 1986), 43–49.

John P. J. Kelly and Susan C. Murphy, "Achieving Dependability Throughout the Development Process: A Distributed Software Experiment," *IEEE Trans. Soft. Eng.* **16**, 2 (February 1990), 153–165.

John P. J. Kelly, Thomas I. McVittie and Susan C. Murphy, "Techniques for Building Dependable Distributed Systems: Multi-Version Software Testing," *Int'l Symp. for Fault-Tolerant Computing* (June 1990), 400–407.

Ron Kenett and Moshe Pollak, "A Semi-Parametric Approach to Testing for Reliability Growth, With Application to Software Systems," *IEEE Trans. Rel.* **35**, 3 (August 1986), 304–311.

Garrison Q. Kenney and Mladen A. Vouk, "Measuring the Field Quality of Wide-Distribution Commercial Software," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 351–357.

Taghi M. Khoshgoftaar and John C. Munson, "Predicting Software Development Errors Using Software Complexity Metrics," *IEEE J. Sel. Areas in Comm.* **8**, 2 (February 1990), 253–261.

Taghi M. Khoshgoftaar, Bibhuti B. Bhattacharyya and Gary D. Richardson, "Predicting Software Errors, During Development, Using Nonlinear Regression Models: A Comparative Study," *IEEE Trans. Rel* **41**, 3 (September 1992), 390–395.

Taghi M. Khoshgoftaar, Abhijit S. Pandya and Hement B. More, "A Neural Network Approach for Predicting Software Development Faults," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 83–89.

K. H. Kim, "Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation," *IEEE Trans. Soft. Eng.* **14**, 6 (June 1988), 810–821.

K. H. Kim and Howard O. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications," *IEEE Trans. Comp.* **38**, 5 (May 1989), 626–636.

B. A. Kitchenham, "Metrics in Practice," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 131–144.

George J. Knafl, "Solving Maximum Likelihood Equations for Two-Parameter Software Reliability Models Using Grouped Data," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 205–213.

John C. Knight and Nancy G. Leveson, "Correlated Failures in Multi-Version Software," *Safety of Computer Control Systems (Safecomp)* (October 1985), 159–165.

John C. Knight and Nancy G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 96–109.

John C. Knight and Nancy G. Leveson, "A Reply to the Criticisms of the Knight & Leveson Experiment," *Soft. Eng. Notes* **15**, 1 (January 1990), 24–35.

John C. Knight and P. E. Ammann, "Design Fault Tolerance," *Rel. Eng. and System Safety* **32**, 1 (1991), 25–49.

Takehisa Kohda and Koichi Enoue, "A Petri Net Approach to Probabilistic Safety Assessment for Obtaining Event Sequences From Component Models," *Probabilistic Safety Assessment and Management* G. Apostolakis, Ed (1991), 729–734.

Walter H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys* **13**, 2 (June 1981), 149–183.

B. G. Kolkhorst and A. J. Macina, "Developing Error-Free Software," *Computer Assurance (Compass)* (1988), 99–107.

H. Kopetz, "Resilient Real-Time Systems," in *Resilient Computer Systems*, T. Anderson, Ed, Wiley (1985), 91–101.

H. Kopetz, H. Kantz, G. Grünsteidl, P. Puschner and J. Reisinger, "Tolerating Transient Faults in Mars," *Fault-Tolerant Computing Symp.* (June 1990), 466–473.

W. Edward Koss, "Software-Reliability Metrics for Military Systems," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1988), 190–193.

C. M. Krishna and Kang G. Shin, "Performance Measures for Control Computers," *IEEE Trans. Automatic Control* **32**, 6 (June 1987), 467–473.

C. M. Krishna, Kang G. Shin and Inderpal S. Bhandari, "Processor Tradeoffs in Distributed Real-Time Systems," *IEEE Trans. Comp.* **36**, 9 (September 1987), 1030–1040.

Gregory A. Kruger, "Project Management Using Software Reliability Growth Models," *Hewlett-Packard J.* **39**, 3 (June 1988), 30–35.

Gregory A. Kruger, "Validation and Further Application of Software Reliability Growth Models," *Hewlett-Packard J.* **40**, 2 (April 1989), 75–79.

Peter Kubat and Harvey S. Koch, "Managing Test Procedures to Achieve Reliable Software," *IEEE Trans. Rel.* **32**, 3 (August 1983), 299–303.

Peter Kubat, "Estimation of Reliability for Communication/Computer Networks—Simulation/Analytic Approach," *IEEE Trans. Comm.* **37**, 9 (September 1989), 927–933.

V. K. Prasanna Kumar, Salim Hariri and C. S. Raghavendra, "Distributed Program Reliability Analysis," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 42–50.

Reino Kurki-Suonio, "Stepwise Design of Real-Time Systems," *ACM Sigsoft Conf. on Soft. for Critical Systems* (December 1991), 120–131.

Jaynarayan H. Lala, Richard E. Harper and Linda S. Alger, "A Design Approach for Ultrareliable Real-Time Systems," *IEEE Computer* **24**, 5 (May 1991), 12–22.

M. La Manna, "A Robust Database for Safe Real-Time Systems," *Safety of Computer Control Systems (Safecomp)* (October 1986), 67–71.

Naftali Langberg and Nozer D. Singpurwalla, "Some Foundational Considerations in Software Reliability Modeling and a Unification of Some Software Reliability Models," in Serra, A., and R. E. Barlow, Ed, *Theory of Reliability* (1984), 379–394.

Naftali Langberg and Nozer D. Singpurwalla, "A Unification of Some Software Reliability Models," *Siam J. Sci. Stat. Comput.* **6**, 3 (July 1985), 781–790.

Jean-Claude Laprie, "Dependability Evaluation of Software Systems in Operation," *IEEE Trans. Soft. Eng.* **10**, 6 (November 1984), 701–714.

Jean-Claude Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Annual Int'l Symp. on Fault-Tolerant Computing* (1985), 2–11.

Jean-Claude Laprie, Christian Béounes, M. Kaâniche and Karama Kanoun, "The Transformation Approach to the Modeling and Evaluation of the Reliability and Availability Growth," *Int'l Symp. for Fault-Tolerant Computing* (June 1990), 364–371.

Jean-Claude Laprie, Jean Arlat, Christian Béounes and Karama Kanoun, "Definition and Analysis of Hardware- and Software-Fault Tolerant Architectures," *IEEE Computer* **23**, 7 (July 1990), 39–51.

Jean-Claude Laprie, Karama Kanoun, Christian Béounes and Mohamed Kaâniche, "The Kat (Knowledge-Action-Transformation) Approach to the Modeling and Evaluation of Reliability and Availability Growth," *IEEE Trans. Soft. Eng.* **17**, 4 (April 1991), 370–382.

Jean-Claude Laprie and Karama Kanoun, "X-Ware Reliability and Availability Modeling," *IEEE Trans. Soft. Eng.* **18**, 2 (February 1992), 130–147.

Jean-Claude Laprie, "for a Product-in-a-Process Approach to Software Reliability Evaluation," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 134–139.

Khiem V. Le and Victor O. K. Li, "Modeling and Analysis of Systems With Multimode Component and Dependent Failures," *IEEE Trans. Rel.* **38**, 1 (April 1989), 68–75.

Philippe R. Leclerq, "A Software-Reliability Assessment Model," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 294–298.

Pierre L'Ecuyer and Jacques Malenfant, "Computing Optimal Checkpointing Strategies for Rollback and Recovery Schemes," *IEEE Trans. Comp.* **37**, 4 (April 1988), 491–496.

Gilbert Le Gall, Marie-Françoise Adam, Henri Derriennic, Bernard Moreau and Nicole Valette, "Studies on Measuring Software," *IEEE J. Sel. Areas in Comm.* **8**, 2 (February 1990), 234–245.

Kang W. Lee, Frank A. Tillman and James J. Higgins, "A Literature Survey of the Human Reliability Component in a Man–Machine System," *IEEE Trans. Rel.* **37**, 1 (April 1988), 24–34.

Pen-Lan Lee and Aderbad Tamboli, "Concurrent Correspondent Modules: A Fault Tolerant Implementation," *Annual Int'l Phoenix Conf. on Comp. and Comm.* (March 1989), 300–304.

Inhwan Lee and Ravishankar K. Iyer, "Analysis of Software Halts in the Tandem Guardian Operating System," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 227–236.

Ann Marie Leone, "Selecting An Appropriate Model for Software Reliability," *Proc. Annual Rel. and Maint. Symp.* (January 1988), 208–213.

Ytzhak Leveldel, "Reliability Analysis of Large Software Systems: Defect Data Modeling," *IEEE Trans. Soft. Eng.* **16**, 2 (February 1990), 141–152.

Nancy G. Leveson, "Software Safety: A Definition and Some Preliminary Thoughts," Tech. Rep. 174, Dept. of Information and Computer Science, Univ. of Cal., Irvine (April 1981).

Nancy G. Leveson and Peter R. Harvey, "Analyzing Software Safety," *IEEE Trans. Soft. Eng.* **9**, 5 (September 1983), 569–579.

Nancy G. Leveson, "Verification of Safety," *Safety of Computer Control Systems (Safecomp)* (September 1983), 167–174.

Nancy G. Leveson, "Software Safety," in *Resilient Computing Systems*, T. Anderson, Ed, Wiley (1985), 122–143.

Nancy G. Leveson, "Software Safety: Why, What and How," *ACM Comp. Surveys* **18**, 2 (June 1986), 125–163.

Nancy G. Leveson, "An Outline of a Program to Enhance Software Safety," *Safety of Computer Control Systems (Safecomp)* (October 1986), 129–135.

Nancy G. Leveson and Janice L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Trans. Soft. Eng.* **13**, 3 (March 1987), 386–397.

Nancy G. Leveson, S. S. Cha, John C. Knight and T. J. Shimeall, "The Use of Self Checks and Voting in Software Error Detection: An Empirical Study," *IEEE Trans. Soft. Eng.* **16**, 4 (April 1990), 432–443.

Nancy G. Leveson, "Software Safety in Embedded Computer Systems," *Comm. ACM* **34**, 2 (February 1991), 34–46.

Nancy G. Leveson, "Safety Assessment and Management Applied to Software," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 377–382.

Nancy G. Leveson, "Safety," in *Aerospace Software Engineering*, Christine Anderson and Merlin Dorfman, Ed, Aiaa (1991), 319–336.

Nancy G. Leveson, Stephen S. Cha and Timothy J. Shimeall, "Safety Verification of Ada Programs Using Software Fault Trees," *IEEE Software* (July 1991), 48–59.

Nancy G. Leveson and Clark S. Turner, "An Investigation of the Therac-25 Accidents," UCI Tech. Report 92–108, Information and Computer Science Dept., University of California, Irvine (November 1992).

Nancy G. Leveson, Mats P. E. Heimdahl, Holly Hildreth and Jon D. Reese, "Requirements Specification for Process-Control Systems," Tech. Report 92–106, Information and Computer Science Dept., University of California, Irvine (November 10 1992).

Ken S. Lew, K. E. Forword and T. S. Dillon, "The Impact of Software Fault Tolerant Techniques on Software Complexity in Real Time Systems," *Safety of Computer Control Systems (Safecomp)* (September 1983), 67–73.

Ken S. Lew, Tharam S. Dillon and Kevin E. Forword, "Software Complexity and Its Impact on Software Reliability," *IEEE Trans. Soft. Eng.* **14**, 11 (November 1988), 1645–1655.

Deron Liang, Ashok Agrawala, Daniel Mosse and Yiheng Shi, "Designing Fault Tolerant Applications in Maruti," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 264–273.

Burt H. Liebowitz and John H. Carson, *Multiple Processor Systems for Real-Time Applications*, Prentice-Hall (1985).

Hsin-Hui Lin and Way Kuo, "Reliability Cost in Software Life-Cycle Models," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1987), 364–368.

Bev Littlewood, "Theories of Software Reliability: How Good Are They and How Can They Be Improved?" *IEEE Trans. Soft. Eng.* **6**, 5 (September 1980), 489–500.

Bev Littlewood, "Stochastic Reliability-Growth: A Model for Fault-Removal in Computer Programs and Hardware Designs," *IEEE Trans. Rel.* **30**, 4 (October 1981), 313–320.

Bev Littlewood and John L. Verrall, "Likelihood Function of a Debugging Model for Computer Software Reliability," *IEEE Trans. Rel.* **30**, 2 (June 1981), 145–148.

Bev Littlewood, "Rationale for a Modified Duane Model," *IEEE Trans. Rel.* **33**, 2 (June 1984), 157–159.

Bev Littlewood, "Software Reliability Prediction," in *Resilient Computing Systems*, T. Anderson, Ed, Wiley (1985), 144–162.

Bev Littlewood and Ariela Sofer, "A Bayesian Modification to the Jelinski-Moranda Software Reliability Growth Model," *Soft. Eng. J.* **2**, 2 (March 1987), 30–41.

Bev Littlewood and Douglas R. Miller, "Conceptual Modeling of Coincident Failures in Multiversion Software," *IEEE Trans. Soft. Eng.* **15**, 12 (December 1989), 1596–1614.

Bev Littlewood and Lorenzo Strigini, "The Risks of Software," *Scientific American* (November 1992), 62–75.

Bev Littlewood and Lorenzo Strigini, "Validation of Ultra-High Dependability for Software-Based Systems," Technical Report (1992).

Bev Littlewood, "Sizewell PPS: Software Concerns," *ACSNImeeting* (June 1992).

D. K. Lloyd and M. Lipow, *Reliability: Management, Methods and Mathematics*, Prentice-Hall (1977).

Roy Longbottom, *Computer System Reliability*, Wiley (1980).

Gan Luo, Gregor V. Bochmann, Behcet Sarikaya and Michel Boyer, "Control-Flow Based Testing of Prolog Programs," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 104–113.

Michael R. Lyu, "Measuring Reliability of Embedded Software: An Empirical Study With JPL Project Data," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 493–500.

Michael R. Lyu and Allen Nikora, "Applying Reliability Models More Effectively," *IEEE Software* **9**, 4 (July 1992), 43–52.

Michael R. Lyu, "Software Reliability Measurements in N-Version Software Execution Environment," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 254–263.

Saeed Maghsoodloo, David B. Brown and Chien-Jong Lin, "A Reliability and Cost Analysis of An Automatic Prototype Generator Test Paradigm," *IEEE Trans. Rel.* **41**, 4 (December 1992), 547–553.

Maria T. Mainini and Luc Billot, "Perfide: An Environment for Evaluation and Monitoring of Software Reliability Metrics During the Test Phase," *Soft. Eng. J.* **5**, 1 (January 1990), 27–32.

S. V. Makam and A. Avizienis, "Aries 81: A Reliability and Life-Cycle Evaluation Tool for Fault-Tolerant Systems," *Proc. IEEE Int'l. Symp. on Fault-Tolerant Computing* (1982), 267–274.

Yashwant K. Malaiya, Nachimuthu Karunanithi and Pradeep Verma, "Predictability of Software-Reliability Models," *IEEE Trans. Rel.* **41**, 4 (December 1992), 539–546.

Yashwant K. Malaiya, A. Von Mayrhauser and P. K. Srimani, "The Nature of Fault Exposure Ratio," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 23–32.

Derek P. Mannering and Bernard Cohen, "The Rigorous Specification and Verification of the Safety Aspects of a Real-Time System," *Computer Assurance (Compass)* (June 1990), 68–85.

Giancarlo Martella, Barbara Pernici and Fabio A. Schreiber, "An Availability Model for Distributed Transaction Systems," *IEEE Trans. Soft. Eng.* **11**, 5 (May 1985), 483–491.

M. R. Bastos Martini, Karama Kanoun and J. Moreira De Souza, "Software Reliability Evaluation of the Tropico-R Switching System," *IEEE Trans. Rel.* **39**, 3 (August 1990), 369–379.

Francis P. Mathur, "Automation of Reliability Evaluation Procedures Through CARE—the Computer-Aided Reliability Estimation Program," *Afips Conf. Proc. Fall Joint Comp. Conf.* (1972).

R. A. Maxion, "Toward Fault-Tolerant User Interfaces," *Safety of Computer Control Systems (Safecomp)* (October 1986), 117–122.

R. A. Maxion, D. P. Siewiorek and S. A. Elkind, "Techniques and Architectures for Fault-Tolerant Computing," *Annual Review Computer Science* (1987), 469–520.

Annelise Von Mayrhauser and John A. Teresinski, "The Effects of Status Code Metrics on Dynamic Software Reliability Models," *Workshop on Software Reliability*, IEEE Comp. Soc. Tech. Comm. on Soft. Rel. Eng. (April 13, 1990).

Thomas A. Mazzuchi and Refik Soyer, "A Bayes Empirical-Bayes Model for Software Reliability," *IEEE Trans. Rel.* **37**, 2 (June 1988), 248–254.

Roger L. McCarthy, "Present and Future Safety Challenges of Computer Control," *Computer Assurance (Compass)* (1988), 1–7.

N. J. McCormick, *Reliability and Risk Analysis*, Academic Press (1981).

John A. McDermid, "Issues in Developing Software for Safety Critical Systems," *Rel. Eng. and System Safety* **32**, 1 (1991), 1–24.

John A. McDermid, "Safety Arguments, Software and System Reliability," *Int'l Symp. on Soft. Rel. Eng.* (1991), 43–50.

Gerald W. McDonald, "Why There Is a Need for a Software-Safety Program," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 30–34.

Archibald McKinlay, "Software Safety Handbook," *Computer Assurance (Compass)* (1989), 14–17.

Archibald McKinlay, "The State of Software Safety Engineering," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 369–376.

Barry T. McKinney, "FMECA the Right Way," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 253–259.

G. R. McNichols, "Software Development Cost Models," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 145–163.

Richard B. Mead, "The Use of Ada PDL as the Basis for Validating a System Specified By Control Flow Logic," *Computer Assurance (Compass)* (June 1992), 77–94.

P. Mellor, "Software Reliability Data Collection: Problems and Standards," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 165–181.

Douglas R. Miller, "Exponential Order Statistic Models of Software Reliability Growth," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 12–24.

Douglas R. Miller and A. Sofer, "A Non-Parametric Approach to Software Reliability, Using Complete Monotonicity," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 183–195.

Keith W. Miller, Larry J. Morell, Robert E. Noonan, Stephen K. Park, David M. Nicol, Branson W. Murrill and Jeffrey M. Voas, "Estimating the Probability of Failure When Testing Reveals No Failures," *IEEE Trans. Soft. Eng.* **18**, 1 (January 1992), 33–43.

Ministry of Defence, "Requirements for the Procurement of Safety Critical Software in Defence Equipment," Draft Defence Standard 00–55, Ministry of Defence, Glasgow (May 1989).

Jean Mirra, Frank McNolty, and William Sherwood, "A General Model for Mixed Exponential Failure," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1987), 176–180.

M. Granger. Morgan, "Probing the Question of Technology-Induced Risk," *IEEE Spectrum* **18**, 11 (November 1981), 58–64 and 18, 12 (December 1983), 53–60.

M. Morganti, "Reliable Communications," *Resilient Computing Systems*, T. Anderson (Ed), Wiley (1985), 78–90.

Louise E. Moser and P. M. Melliar-Smith, "Formal Verification of Safety-Critical Systems," *Soft. Prac. and Exper.* **20**, 8 (August 1990), 799–821.

M. Mulazzani, "Reliability Versus Safety," *Safety of Computer Control Systems (Safecomp)* (October 1985), 141–145.

M. Mulazzani and K. Trivedi, "Dependability Prediction: Comparison of Tools and Techniques," *Safety of Computer Control Systems (Safecomp)* (October 1986), 171–178.

Jogesh K. Muppala, Steven P. Woolet and Kishor S. Trivedi, "Real-Time Systems Performance in the Presence of Failures," *IEEE Computer* **24**, 5 (May 1991), 37–47.

John D. Musa, "The Measurement and Management of Software Reliability," *Proc. IEEE* **68**, 9 (September 1980), 1131–1143.

John D. Musa and Kazuhira Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," *Proc Int'l Conf. Soft. Eng.* (March 1984), 230–238.

John D. Musa, Anthony Iannino and Kazuhira Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw Hill (1987).

John D. Musa, "Tools for Measuring Software Reliability," *IEEE Spectrum* (February 1989), 39–42.

John D. Musa and A. Frank Ackerman, "Reliability," in *Aerospace Software Engineering*, Christine Anderson and Merlin Dofrman, Ed, Aiaa (1991), 289–317.

John D. Musa, "Software Reliability Engineering: Determining the Operational Profile," AT&T Bell Laboratories Tech. Rpt. (October 1992).

John D. Musa, "The Operational Profile in Software Reliability Engineering: An Overview," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 140–154.

Yutaka Nakagawa and Shuetsu Hanata, "An Error Complexity Model for Software Reliability Measurement," *Proc. Int'l Conf. Soft. Eng.* (1989), 230–236.

Takeshi Nakajo, Katsuhiko Sasabuchi and Tadashi Akiyama, "A Structured Approach to Software Defect Analysis," *Hewlett-Packard J.* **40**, 2 (April 1989), 50–56.

Kyoichi Nakashima, "Redundancy Technique for Fail-Safe Design of Systems," *Probabilistic Safety Assessment and Management* , G. Apostolakis, Ed (1991), 735–740.

Fares A. Nassar and Dorothy M. Andrews, "A Methodology for Analysis of Failure Prediction Data," *Proc. Real Time Systems Symp.* (December 1985), 160–166.

Tapan K. Nayak, "Software Reliability: Statistical Modeling and Estimation," *IEEE Trans. Rel.* **35**, 5 (December 1986), 566–570.

Tapan K. Nayak, "Estimating Population Size By Recapture Sampling," *Biometrika* **75**, 1 (1988), 113–120.

Victor A. Nelson and Bill D. Carroll, "Introduction to Fault-Tolerant Computing," in *Tutorial: Fault-Tolerant Computing*, Nelson and Carroll, Ed (1987), 1–4.

Victor P. Nelson, "Fault-Tolerant Computing: Fundamental Concepts," *IEEE Comp.* **23**, 7 (July 1990), 19–25.

Peter G. Neumann, "The Computer-Related Risk of the Year: Computer Abuse," *Computer Assurance (Compass)* (1988), 8–12.

Peter G. Neumann, "The Computer-Related Risk of the Year: Misplaced Trust in Computer Systems," *Computer Assurance (Compass)* (1989), 9–13.

Peter G. Neumann, "The Computer-Related Risk of the Year: Distributed Control," *Computer Assurance (Compass)* (1990), 173–177.

Ying-Wah Ng and Algirdas Avizienis, "A Model for Transient and Permanent Fault Recovery in Closed Fault-Tolerant Systems," *Proc. Int'l Symp. Fault-Tolerant Comp.* (1976).

Ying W. Ng and Algirdas A. Avizienis, "A Unified Reliability Model for Fault-Tolerant Computers," *IEEE Trans. Comp.* **29**, 11 (November 1980), 1002–1011.

Victor F. Nicola and F. J. Kylstra, "A Model of Checkpointing and Recovery With a Specified Number of Transactions Between Checkpoints," *Performance '83* (1983), 83–99.

Victor F. Nicola, V. G. Kulkarni and Kishor S. Trivedi, "Queuing Analysis of Fault-Tolerant Computer Systems," *IEEE Trans. Soft. Eng.* **13**, 3 (March 1987), 363–375.

Victor F. Nicola and Ambuj Goyal, "Modeling of Correlated Failures and Community Error Recovery in Multiversion Software," *IEEE Trans. Soft. Eng.* **16**, 3 (March 1990), 350–359.

H. P. Nourbakhsh and E. G. Cazzoli, "A Simplified Approach for Predicting Radionuclide Releases From Light Water Reactor Accidents," *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 693–698.

Mitsuru Ohba and Xioa-Mei Chou, "Does Imperfect Debugging Affect Software Reliability Growth?" *Proc. Int'l Conf. Soft. Eng.* (1989), 237–244.

Hiroshi Ohtera and Shigeru Yamada, "Optimal Allocation and Control Problems for Software Testing Resources," *IEEE Trans. Rel.* **39**, 2 (June 1990), 171–176.

Kazuhira Okumoto, "A Statistical Method for Software Quality Control," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1424–1430.

J. S. Ostroff, "Verifying Finite State Real-Time Discrete Event Processes,' *Proc. Int'l Conf. Dist. Sys.* (1989), 207–216.

Martyn A. Ould, "Testing—A Challenge to Method and Tool Developers," *Soft. Eng. J.* **6**, 2 (March 1991), 59–64.

A. Pages and M. Gondran, *System Reliability Evaluation and Prediction in Engineering*, Springer-Verlag (1986).

Christopher J. Palermo, "Software Engineering Malpractice and Its Avoidance," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 41–50.

D. B. Parkinson, "Reliability Bounds for Dependent Failures," *IEEE Trans. Rel.* **37**, 1 (April 1988), 54–56.

David L. Parnas, A. John Van Schouwen and Shu Po Kwan, "Evaluation of Safety-Critical Software," *Comm. ACM* **33**, 6 (June 1990), 636–648.

David L. Parnas, G. J. K. Asmis and J. Madey, "Assessment of Safety-Critical Software in Nuclear Power Plants," *Nuclear Safety* **32**, 2 (April-June 1991), 189–198.

V. B. Patki, A. B. Patki and B. N. Chatterji, "Reliability and Maintainability Considerations in Computer Performance Evaluation," IEEE Trans. Reliability R-32, 5 (December 1983), 433–436.

Charles C. Perrow, *Normal Accidents: Living With High Risk Technologies*, Basic Books (1984).

Ivars Peterson, "Finding Fault: The formidable Task of Eradicating Software Bugs," *Science News* **139** (February 16, 1991), 104–106.

S. Petrella, P. Michael, W. C. Bowman and S. T. Lim, "Random Testing of Reactor Shutdown System Software," in *Probabilistic Safety Assessment and Management*, G. Apostolakis, Ed (1991), 681–686.

Yannis A. Phillis, Henry D'Angelo, and Gordon C. Saussy, "Analysis of Series-Parallel Production Networks Without Buffers," *IEEE Trans. Rel.* **35**, 2 (June 1986), 179–184.

Patrick R. H. Place, William G. Wood and Mike Tudball, *Survey of Formal Specification Techniques for Reactive Systems*, Tech. Rpt. CMU/SEI-90–TR-5 (May 1990).

D. J. Pradhan, Ed, *Fault Tolerant Computing: Theory and Practice*, Prentice-Hall (1986).

Kevin H. Prodromides and William H. Sanders, "Performability Evaluation of CSMA/CD and CSMA/DCR Protocols Under Transient Fault Conditions," *Tenth Symp. on Reliable Distributed Systems* (September-October 1991), 166–176.

Geppino Pucci, "On the Modeling and Testing of Recovery Block Structures," *Int'l Symp. Fault-Tolerant Comp.* (June 1990), 356–363.

Geppino Pucci, "A New Approach to the Modeling of Recovery Block Structures," *IEEE Software* **9**, 4 (July 1992), 159–167.

James M. Purtilo and Jankaj Jalote, "An Environment for Developing Fault-Tolerant Software," *IEEE Trans. Soft. Eng.* **17**, 2 (February 1991), 153–159.

W. J. Quirk, "Engineering Software Safety," *Safety of Computer Control Systems (Safecomp)* (October 1986), 143–147.

C. S. Raghavendra and S. V. Makam, "Reliability Modeling and Analysis of Computer Networks," *IEEE Trans. Rel.* **35**, 2 (June 1986), 156–160.

C. S. Raghavendra, V. K. P. Kumar and S. Hariri, "Reliability Analysis in Distributed Systems," *IEEE Trans. Comp.* **37**, 3 (March 1988), 352–358.

Dev G. Raheja, "Software Reliability Growth Process—A Life Cycle Approach," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 52–55.

Chandrashekar Rajaraman and Michael R. Lyu, "Reliability and Maintainability Related Software Coupling Metrics in C++ Programs," Third Int'l Symp. on Soft. Rel. Eng. (October 1992), 303–311.

C. S. Ramanjaneyulu, and V. V. S. Sarma, "Modeling Server-Unreliability in Closed Queuing Networks," *IEEE Trans. Rel.* **38**, 1 (April 1989), 90–95.

C. V. Ramamoorthy, A. Prakash, W. T. Tsai and Y. Usuda, "Software Reliability: Its Nature, Models and Improvement Techniques," in Serra, A., and R. E. Barlow, Ed, *Theory of Reliability*, North Holland, (1984), 287–320.

Brian P. A. L. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Soft. Eng.* **1**, 2 (June 1975), 220–232.

Brian P. A. L. Randell and P. C. Treleaven, "Reliability Issues in Computing System Design," *ACM Comp. Surv.* **10**, 2 (June 1978), 123–165.

D. M. Rao and S. Bologna, "Verification and Validation Program for a Distributed Computer System for Safety Applications," *Safety of Computer Control Systems (Safecomp)* (October 1985), 39–46.

Christophe Ratel, Nicolas Halbwachs and Pascal Raymond, "Programming and Verifying Critical Systems By Means of the Synchronous Data-Flow Language Lustre," *ACM Sigsoft Conf. on Soft. for Critical Systems* (December 1991), 112–119.

Steve Rayner, "Disagreeing About Risk: The Institutional Cultures of Risk Management and Planning for Future Generations," in *Risk Analysis, Institutions and Public Policy*, Susan G. Hadden, Ed., Associated Faculty Press (1984), 150–178.

Steve Rayner, "Risk and Relativism in Science for Policy," in *the Social and Cultural Construction of Risk*, B. B. Johnson and V. T. Covello, Ed, D. Reidel Pub (1987), 5–23.

Steve Rayner, "Muddling Through Metaphors to Maturity: A Commentary on Kasperson Et Al, the Social Amplification of Risk," *Risk Analysis* **8**, 2 (1988), 201–204.

Steve Rayner, "Opening Remarks," *Symp. on Global Environmental Change and the Public, First Int'l Conf. on Risk Comm.* (October 20, 1988).

Steve Rayner, "Equity Issues in Global Risk Management: The Case of Climate Change," *Amer. Anthro. Assoc. Annual Meeting* (November1988).

Steve Rayner, "Risk in Cultural Perspective," Oak Ridge National Laboratory (January 1989).

F. J. Redmill, Ed, *Dependability of Critical Computer Systems 1*, Elsevier (1988). Guidelines Produced By the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7).

F. J. Redmill, Ed, *Dependability of Critical Computer Systems 2* Elsevier (1989). Guidelines Produced By the European Workshop on Industrial Computer Systems, Technical Committee 7 (EWICS TC7).

Andrew L. Reibman and Malathi Veeraraghavan, "Reliability Modeling: An Overview for System Designers, " *IEEE Computer* **24**, 4 (April 1991), 49–56.

David A. Rennels, "Fault-Tolerant Computing—Concepts and Examples," *IEEE Trans. Comp.* **33**, 12 (December 1984), 1116–1129.

J. Luis Roca, "A Method for Microprocessor Software Reliability Prediction," *IEEE Trans. Rel.* **37**, 1 (April 1988), 88–91.

Jorge L. Romeu and Kieron A. Dey, "Classifying Combined Hardware/Software R Models," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1984), 282–287.

Sheldon M. Ross, "Statistical Estimation of Software Reliability," *IEEE Trans. Soft. Eng.* **11**, 5 (May 1985), 479–483.

Sheldon M. Ross, "Software Reliability: The Stopping Rule Problem," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1472–1476.

B. Runge, "Quantitative Assessment of Safe and Reliable Software," *Safety of Computer Control Systems (Safecomp)* (October 1986), 7–11.

John Rushby and Friedrich Von Henke, "Formal Verification of Algorithms for Critical Systems," *ACM Sigsoft Conf. on Soft. for Critical Systems* (December 1991), 1–15.

John Rushby, *Formal Methods for Software and Hardware in Digital Flight Control Systems*, Computer Science Laboratory, Sri International (July 1992 Draft).

Krishan K. Sabnani, Aleta M. Lapone and M. Umit Uyar, "An Algorithmic Procedure for Checking Safety Properties of Protocols," *IEEE Trans. Comm.* **37**, 9 (September 1989), 940–948.

F. Sagietti and W. Ehrenberger, "Software Diversity—Some Considerations About Its Benefits and Its Limitations," *Safety of Computer Control Systems (Safecomp)* (October 1986), 27–34.

J. S. Sagoo and D. J. Holding, "The Use of Temporal Petri Nets in the Specification and Design of Systems With Safety Implications," *Algorithms and Arch. for Real-Time Control Workshop* (September 1991), 231–236.

Robin A. Sahner and Kishor S. Trivedi, "A Hierarchical, Combinatorial-Markov Method of Solving Complex Reliability Models," *Proc. Fall Joint Comp. Conf.*, (1986), 817–825.

R. A. Sahner and Kishor S. Trivedi, "Reliability Modeling Using Sharpe," *IEEE Trans. Rel.* **36**, 2 (June 1987), 186–193.

R. A. Sahner and Kishor S. Trivedi, "Performance and Reliability Analysis Using Directed Acyclic Graphs," *IEEE Trans. Soft. Eng.* **13**, 10 (October 1987), 1105–1114.

C. Sayet and E. Pilaud, "An Experience of a Critical Software Development," *Int'l Symp. Fault-Tolerant Comp.* (June 1990), 36–45.

Richard D. Schlichting, "A Technique for Estimating Performance of Fault-Tolerant Programs," *Proc. IEEE Symp. on Rel. of Dist. Soft. and Database Systems* (October 1984), 62–74.

Fred B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comp. Surv.* **22**, 4 (December 1990), 299–319.

Norman F. Schneidewind and Ted W. Keller, "Applying Reliability Models to the Space Shuttle," *IEEE Software* **9**, 4 (July 1992), 28–33.

Norman F. Schneidewind, "Minimizing Risk in Applying Metrics on Multiple Projects," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 173–182.

F. W. Scholz, "Software Reliability Modeling and Analysis," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 25–31.

Werner Schütz, "On the Testability of Distributed Real-Time Systems," *Symp. on Rel. Dist. Syst.* (September-October 1991), 52–61.

Karsten Schwan, Thomas E. Bihari and Ben A. Blake, "Adaptive Reliable Software for Distributed and Parallel Real-Time Systems," *Proc. IEEE/ACM Symp. Rel. in Dist. Soft. and Database Systems* (March 1987), 32–42.

R. Keith Scott, James W. Gault and David F. McAllister, "Modeling Fault-Tolerant Software Reliability," *Proc. IEEE Symp. Rel. in Dist. Soft. and Database Systems* (October 1983), 15–27.

R. Keith Scott, James W. Gault, David F. McAllister and Jeffrey Wiggs, "Investigating Version Dependence in Fault-Tolerant Software," *Avionics Panel Symp.* (May 1984), 21.1–21.11

R. Keith Scott, James W. Gault and David F. McAllister, "Fault-Tolerant Software Reliability Modeling," *IEEE Trans. Soft. Eng.* **13**, 5 (May 1987), 582–592.

Richard W. Selby, "Empirically Based Analysis of Failures in Software Systems," *IEEE Trans. Rel.* **39**, 4 (October 1990), 444–454.

A. Serra and R. E. Barlow, Ed, *Theory of Reliability*, North-Holland (1984).

Alexander N. Shabalin, "Generation of Models for Reliability Growth," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 299–302.

Sol M. Shatz and Jia-Ping Wang, "Models and Algorithms for Reliability-Oriented Task-Allocation in Redundant Distributed Computer Systems," *IEEE Trans. Rel.* **38**, 1 (April 1989), 16–27.

Paul V. Shebalin, Sang H. Son and Chun-Hyon Chang, "An Approach to Software Safety Analysis in a Distributed Real-Time System," *Computer Assurance (Compass)* (1988), 29–43.

Frederick T. Sheldon, Krishna M. Kavi, Robert C. Tausworthe, James T. Yu, Ralph Brettschneider and William W. Everett, "Reliability Measurement: From Theory to Practice," *IEEE Software* **9**, 4 (July 1992), 13–20.

Vincent Y. Shen, Tze-Jie Yu, Stephen M. Thebaut and Lorri R. Paulsen, "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. Soft. Eng.* **11**, 4 (April 1985), 317–323.

Y. S. Sherif and N. A. Kheir, "Reliability and Failure Analysis of Computing Systems," *Comp. & Elect. Eng.* **11**, 2/3 (1984), 151–157.

Yuan-Bao Shieh, Dipak Ghosal, Prasad R. Chintamaneni and Satish K. Tripathi, "Application of Petri Net Models for the Evaluation of Fault-Tolerant Techniques in Distributed Systems," *Proc Int'l Conf. Dist. Syst.* (1989), 151–159.

Yuan-Bao Shieh, Dipak Ghosal, Prasad R. Chintamaneni and Satish K. Tripathi, "Modeling of Hierarchical Distributed Systems With Fault-Tolerance," *IEEE Trans. Soft. Eng.* **16**, 4 (April 1990), 444–457.

Timothy J. Shimeall and Nancy G. Leveson, "An Empirical Comparison of Software Fault Tolerance and Fault Elimination," *IEEE Trans. Soft. Eng.* **17**, 2 (February 1991), 173–182.

Kang G. Shin and James W. Dolter, "Alternative Majority-Voting Methods for Real-Time Computing Systems," *IEEE Trans. Rel.* **38**, 1 (April 1989), 58–64.

Kang G. Shin, "Harts: A Distributed Real-Time Architecture," *IEEE Computer* **24**, 5 (May 1991), 25–35.

Martin L. Shooman, "Software Reliability: A Historical Perspective," *IEEE Trans. Rel.* **33**, 1 (April 1984), 48–55.

Martin L. Shooman, "A Class of Exponential Software Reliability Models," *Workshop on Soft. Rel.*, IEEE Comp. Soc. Tech. Comm. on Soft. Rel. Eng. April 13, 1990).

Andrew M. Shooman and Aaron Kershenbaum, "Methods for Communication-Network Reliability Analysis: Probabilistic Graph Reduction," *IEEE Annual Rel. and Maint. Symp.* (January 1992), 441–448.

Santosh K. Shrivastava, "Structuring Distributed Systems for Recoverability and Crash Resistance," *IEEE Trans. Soft. Eng.* **7**, 4 (July 1981), 436–447.

Santosh K. Shrivastava, "Robust Distributed Programs," in *Resilient Computing Systems*, T. Anderson, Ed, Wiley (1985), 102–121.

David M. Siefert and George E. Stark, "Software Reliability Handbook: Achieving Reliable Software," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 126–130.

Kyle Siegrist, "Reliability of Systems With Markov Transfer of Control," *IEEE Trans. Soft. Eng.* **14**, 7 (July 1988), 1049–1053.

Kyle Siegrist, "Reliability of Systems With Markov Transfer of Control II," *IEEE Trans. Soft. Eng.* **14**, 10 (October 1988), 1478–1481.

Daniel P. Siewiorek and R. S. Swarz, *the Theory and Practice of Reliable System Design*, Digital Press (1982).

Daniel P. Siewiorek, "Architecture of Fault-Tolerant Computers," in *Fault-Tolerant Computing*, D. K. Pradhan (Ed), Prentice Hall (1986), 417–466.

Daniel P. Siewiorek, "Fault Tolerance in Commercial Computers," *IEEE Computer* **23**, 7 (July 1990), 26–37.

Nozer D. Singpurwalla and Refik Soyer, "Assessing (Software) Reliability Growth Using a Random Coefficient Autoregressive Process and Its Ramifications," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1456–1464.

Nozer D. Singpurwalla, "Determining An Optimal Time Interval for Testing and Debugging Software," *IEEE Trans. Soft. Eng.* **17**, 4 (April 1991), 313–319.

Paul Slovic, "Perception of Risk," *Science* **236** (April 17, 1987), 280–285.

David J. Smith, *Reliability Engineering*, Pitnam (1972).

R. M. Smith and Kishor S. Trivedi, "A Performability Analysis of Two Multi-Processor Systems," *Proc. Int'l. Symp. on Fault Tolerant Computing* (July 1987), 224–229.

R. M. Smith, Kishor S. Trivedi and A. V. Ramesh, "Performability Analysis: Measures, an Algorithm, and a Case Study," *IEEE Trans. Comp.* **37**, 4 (April 1988), 406–417.

Mark Smotherman, Robert M. Geist and Kishor S. Trivedi, "Provably Conservative Approximations to Complex Reliability Models," *IEEE Trans. Comp.* **35**, 4 (April 1986), 333–338.

Ariela Sofer and Douglas R. Miller, "A Nonparametric Software-Reliability Growth Model," *IEEE Trans. Rel.* **40**, 3 (August 1991), 329–337.

R. Soyer, "Applications of Time Series Models to Software Reliability Analysis," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 197–207.

Debra Sparkman, "Standards and Practices for Reliable Safety-Related Software Systems," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 318–328.

Pradip K. Srimani and Yashwant K. Malaiya, "Steps to Practical Reliability Measurement," *IEEE Software* **9**, 4 (July 1992), 10–12.

George E. Stark, "Dependability Evaluation of Integrated Hardware/Software Systems," *IEEE Trans. Rel.* **36**, 4 (October 1987), 440–444.

George E. Stark, "The AIAA Software Reliability Recommended Practice," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 131–132.

Susan Stepney and Stephen P. Lord, "Formal Specification of An Access Control System," *Soft. Prac. and Exper.* **17**, 9 (September 1987), 575–593.

J. J. Stiffler, "Computer-Aided Reliability Estimation," in *Fault-Tolerant Computing, Theory and Techniques*, D. K. Pradhan, Ed, Prentice Hall (1985), 633–657.

L. Strigini and A. Avizienis, "Software Fault-Tolerance and Design Diversity: Past Experience and Future Evolution," *Safety of Computer Control Systems (Safecomp)* (October 1985), 167–172.

Byung-Hoon Suh, John Hudak, Dan Siewiorek and Zary Segall, "Development of a Benchmark to Measure System Robustness: Experiences and Lessons Learned," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 237–245.

Gregory F. Sullivan and Gerald M. Masson, "Using Certification Trails to Achieve Software Fault Tolerance," *Int'l Symp. for Fault-Tolerant Comp.* (June 1990), 423–431.

Ushio Sumita and Yasushi Masuda, "Analysis of Software Availability/Reliability Under the Influence of Hardware Failures," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 32–41.

Arnold L. Sweet, Yungtat Lam, and Julian Bott, "Computation of Mean Time to Failure for a System With Simple Redundancy," *IEEE Trans. Rel.* **35**, 5 (December 1986), 539–540.

K. C. Tai, "Predicate-Based Test Generation for Computer Programs," Computer Science Dept., North Carolina State Univ. (September 1992).

Andrew S. Tanenbaum, "Reliability Issues in Distributed Operating Systems," *Proc. IEEE/ACM Symp. on Rel. in Dist. Soft. & Database Systems* (March 1987), 3–11.

Dong Tang, Ravishankar K. Iyer and Sujatha S. Subramani, "Failure Analysis and Modeling of a Vaxcluster System," *Int'l Symp. Fault-Tolerant Computing* (June 1990), 244–251.

Dong Tang and Ravishankar K. Iyer, "Analysis of the VAX/VMS Error Logs in Multicomputer Environments—A Case Study of Software Dependability," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 216–226.

David J. Taylor, "Concurrency and Forward Recovery in Atomic Actions," *IEEE Trans. Soft. Eng.* **12**, 1 (January 1986), 69–78.

David J. Taylor, "Error Models for Robust Storage Structures," *Int'l Symp. Fault-Tolerant Computing* (June 1990), 416–422.

Philip Thambidurai, You-Keun Park and Kishor S. Trivedi, "On Reliability Modeling of Fault-Tolerant Distributed Systems," *Proc. International Conf. Distributed Systems* (1989), 136–142.

W. A. Thompson, "On the Foundations of Reliability," *Technometrics* **23**, 1 (February 1981), 1–13.

Jianhui Tian, Adam Porter and Marvin V. Zelkowitz, "An Improved Classification Tree Analysis of High Cost Modules Based Upon An Axiomatic Definition of Complexity," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 164–172.

Yoshihiro Tohma, Kenshin Tokunaga, Shinji Nagase and Yukihisa Murata, "Structural Approach to the Estimation of the Number of Residual Software Faults Based on the Hypergeometric Distribution," *IEEE Trans. Soft. Eng.* **15**, 3 (March 1989), 345–355.

Yoshihiro Tohma, Hisashi Yamano, Morio Ohba and Raymond Jacoby, "The Estimation of Parameters of the Hypergeometric Distribution and Its Application to the Software Reliability Growth Model," *IEEE Trans. Soft. Eng.* **17**, 5 (May 1991), 483–489.

Martin Trachtenberg, "The Linear Software Reliability Model and Uniform Testing," *IEEE Trans. Rel.* **34**, 1 (April 1985), 8–16.

Martin Trachtenberg, "A General Theory of Software-Reliability Modeling," *IEEE Trans. Rel.* **39**, 1 (April 1990), 92–96.

Martin Trachtenberg, "Why Failure Rates Observe Zipf's Law in Operational Software," *IEEE Trans. Rel.* **41**, 3 (September 1992), 386–389.

A. M. Traverso, "A Tool for Specification Analysis: 'Complete' Decision Tables," *Safety of Computer Control Systems (Safecomp)* (October 1985), 53–56.

Kishor S. Trivedi and Robert M. Geist, "Decomposition in Reliability Analysis of Fault-Tolerant Systems," *IEEE Trans. Rel.* **32**, 5 (December 1983), 463–468.

Kishor Trivedi, Joanne Bechta Dugan, Robert Geist and Mark Smotherman, "Hybrid Reliability Modeling of Fault-Tolerant Computer Systems," *Comput. & Elect. Eng.* **11**, 2/3 (1984), 87–108.

Kishor Trivedi, Joanne Bechta Dugan, Robert Geist and Mark Smotherman, "Modeling Imperfect Coverage in Fault-Tolerant Systems," *Int'l. Conf. Fault Tolerant Computing* (June 1984), 77–82.

Robert Troy and Ramadan Moawad, "Assessment of Software Reliability Models," *IEEE Trans. Soft. Eng.* **11**, 9 (September 1985), 839–849.

K. S. Tso, A. Avizienis and J. P. J. Kelly, "Error Recovery in Multi-Version Software," *Safety of Computer Control Systems (Safecomp)* (October 1986), 35–40.

U.S. House of Representatives, *Bugs in the Program: Problems in Federal Government Computer Software Development and Regulation*, Staff Study By the Subcommittee on Investigations and Oversight, Committee on Science, Space and Technology, (September 1989).

S. Utena, T. Suzuki, H. Asano and H. Sakamoto, "Development of the BWR Safety Protection System With a New Digital Control System," *Int'l Symp. on Nuclear Power Plant Instr. and Control* (May 1992).

Véronique Valette and Frédérique Vallée, "Software Quality Metrics in Space Systems," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 296–302.

Anujan Varma and C. S. Raghavendra, "Reliability Analysis of Redundant-Path Interconnection Networks," *IEEE Trans. Rel.* **38**, 1 (April 1989), 130–137.

Robert L. Vienneau, "The Cost of Testing Software," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 423–427.

Jeffrey M. Voas and Keith W. Miller, "Improving the Software Development Process Using Testability Research," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 114–121.

Mladen A. Vouk, "On Back-to-Back Testing," *Computer Assurance (Compass)* (1988), 84–91.

Dolores R. Wallace, Laura M. Ippolito and D. Richard Kuhn, "High Integrity Software Standards and Guidelines," NIST Sp. Pub. 500–204, National Inst. of Std. and Tech. (September 1992).

Dolores R. Wallace, D. Richard Kuhn and Laura M. Ippolito, "An Analysis of Selected Software Safety Standards," *Computer Assurance (Compass)* (June 1992), 123–136.

L. A. Walls and A. Bendell, "An Exploratory Approach to Software Reliability Measurement," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 209–227.

Chris J. Walter, "Identifying the Cause of Detected Errors," *Int'l Symp. for Fault-Tolerant Comp.* (June 1990), 48–54.

Benjamin C. Wei, "A Unified Approach to Failure Mode, Effects and Criticality Analysis (FMECA)," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1991), 260–271.

Anne S. Wein and Archana Sathaye, "Validating Complex Computer System Availability Models," *IEEE Trans. Rel.* **39**, 4 (October 1990), 468–479.

Stewart N. Weiss and Elaine J. Weyuker, "An Extended Domain-Based Model of Software Reliability," *IEEE Trans. Soft. Eng.* **14**, 10 (October 1988), 1512–1524.

John H. Wensley, "Fault Tolerance in a Local Area Network Used for Industrial Control," *Proc. Real-Time Systems Symp.* (December 1983), 113–118.

John H. Wensley, "The Man-Machine Interface for a Fault Tolerant Control System," *Safety of Computer Control Systems (Safecomp)* (September 1983), 95–99.

B. A. Wichmann, "A Note on the Use of Floating Point in Critical Systems," *the Comp. J.* **35**, 1 (January 1992), 41–44.

D. W. Wightman and A. Bendell, "Proportional Hazards Modeling of Software Failure Data," in *Software Reliability State of the Art Report* **14**, 2, Pergamon Infotech (1986), 229–242.

Antonin Wild, "What Is a System Approach to Prediction?" *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1989), 342–346.

Claes Wohlin and Per Runeson, "A Method Proposal for Early Software Reliability Estimation," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 156–163.

Denise M. Woit, "Realistic Expectations of Random Testing," CRL Rpt 246, Telecommunications Research Institute of Ontario, McMaster University, Hamilton, Ontario (May 1992).

Denise M. Woit, "An Analysis of Black-Box Testing Techniques," CRL Rpt 245, Telecommunications Research Institute of Ontario, McMaster University, Hamilton, Ontario (May 1992).

Michael H. Woodbury and Kang G. Shin, "Measurement and Analysis of Workload Effects on Fault Latency in Real-Time Systems," *IEEE Trans. Soft. Eng.* **16**, 2 (February 1990), 212–216.

M. Xie and M. Zhao, "The Schneidewind Software Reliability Model Revisited," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 184–192.

Y. W. Yak, T. S. Dillon and K. E. Forword, "Incorporation of Recovery and Repair Time in the Reliability Modeling of Fault-Tolerant System," *Safety of Computer Control Systems (Safecomp)* (September 1983), 45–52.

Y. W. Yak, K. E. Forword and T. S. Dillon, "Modeling the Effect of Transient Faults in Fault Tolerant Computer System," *Safety of Computer Control Systems (Safecomp)* (October 1985), 129–133.

Y. W. Yak, T. S. Dillon and K. E. Forword, "The Effect of Incomplete and Deleterious Periodic Maintenance on Fault-Tolerant Computer Systems," *IEEE Trans. Rel.* **35**, 1 (April 1986), 85–90.

Shigeru Yamada, Mitsuru Ohba and Shunji Osaki, "S-Shaped Software Reliability Growth Models and Their Applications," *IEEE Trans. Rel.* **33**, 4 (October 1984), 289–292.

Shigeru Yamada and Shunji Osaki, "Software Reliability Growth Modeling: Models and Applications," *IEEE Trans. Soft. Eng.* **11**, 12 (December 1985), 1431–1437.

Shigeru Yamada, Hiroshi Ohtera and Miroyuki Narihisa, "Software Reliability Growth Models With Testing Effort," *IEEE Trans. Rel.* **35**, 1 (April 1986), 19–23.

Raif M. Yanney and John P. Hayes, "Distributed Recovery in Fault-Tolerant Multiprocessor Networks," *IEEE Trans. Comp.* **35**, 10 (October 1986), 871–879.

Wilson D. Yates and David A. Shaller, "Reliability Engineering as Applied to Software," *Proc. IEEE Annual Rel. and Maint. Symp.* (January 1990), 425–429.

Jong P. Yoon, "Techniques for Data and Rule Validation in Knowledge-Based Systems," *Computer Assurance (Compass)* (1989), 62–70.

Michal Young and Richard N. Taylor, "Rethinking the Taxonomy of Fault Detection Techniques," *Proc. Int'l Conf. Soft. Eng.* (1989), 53–62.

W. Y. Yun and D. S. Bai, "Optimum Software Release Policy With Random Life Cycle," *IEEE Trans. Rel.* **39**, 2 (June 1990), 167–170.

Fatemeh Zahedi and Noushin Ashrafi, "Software Reliability Allocation Based on Structure, Utility, Price and Cost," *IEEE Trans. Soft. Eng.* **17**, 4 (April 1991), 345–356.

M. Zhao and M. Xie, "On the Log-Power NHPP Software Reliability Model," *Third Int'l Symp. on Soft. Rel. Eng.* (October 1992), 14–22.

Kathlean C. Zinnel, "Using Software Reliability Growth Models to Guide Release Decisions," *Workshop on Software Reliability*, IEEE Comp. Soc. Tech. Comm. on Soft. Rel. Eng. (April 13, 1990).

DATE

FILMED

11/13/94

END