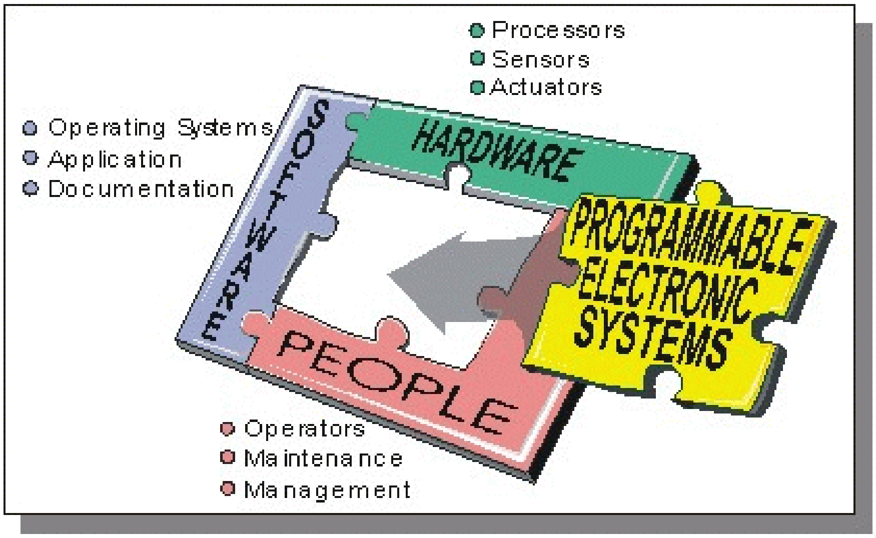
SYSTEM SAFETY EVALUATION PROGRAM

Programmable Electronic Mining Systems: Best Practice Recommendations (In Nine Parts)



Part 3: 2.2

Software Safety

Mine Safety and Health Administration
Approval and Certification Center
Electrical Safety Division
Triadelphia, West Virginia
October 2001

U.S. Department of Labor
MSHA
Mine Safety & Health Administration

Department of Health and Human Services
Centers for Disease Control and Prevention
National Institute for Occupational Safety and Health
Pittsburgh Research Laboratory
Pittsburgh, Pennsylvania

NIOSH

CONTENTS

Page

ILLUSTRATIONS

TABLES

PROGRAMMABLE ELECTRONIC MINING SYSTEMS: BEST PRACTICE RECOMMENDATIONS (In Nine Parts)

Part 3: 2.2 Software Safety

By Edward F. Fries,[1] Thomas J. Fisher,[2] and Christopher C. Jobes, Ph.D.[3]

ABSTRACT

This report (Software Safety 2.2) is the third in a nine-part series of recommendations addressing the functional safety of processor-controlled mining equipment. It is part of a risk-based system safety process encompassing hardware, software, humans, and the operating environment for the equipment's life cycle. Figure 1 shows a safety framework containing these recommendations. The reports in this series address the various life cycle stages of inception, design, approval and certification, commissioning, operation, maintenance, and decommissioning. These recommendations were developed as a joint project between the National Institute for Occupational Safety and Health and the Mine Safety and Health Administration. They are intended for use by mining companies, original equipment manufacturers, and aftermarket suppliers to these mining companies. Users of these reports are expected to consider the set in total during the design cycle.

• 1.0 *Safety Introduction*.—This is an introductory report for the general mining industry. It provides basic system/software safety concepts, discusses the need for mining to address the functional safety of programmable electronics (PE), and includes the benefits of implementing a system/software safety program.

• 2.1 *System Safety* and 2.2 *Software Safety*.—These reports draw heavily from International Electrotechnical Commission (IEC) standard IEC 61508 [IEC 1998a,b,c,d,e,f,g] and other standards. The scope is "surface and underground safety-related mining systems employing embedded, networked, and nonnetworked programmable electronics." System safety seeks to design safety into all phases of the entire system. Software is a subsystem; thus, software safety is a part of the system's safety.

• 3.0 *Safety File*.—This report contains the documentation that demonstrates the level of safety built into the system and identifies limitations for the system's use and operation. In essence, it is a "proof of safety" that the system and its operation meets the appropriate level of safety for the intended application. It starts from the beginning of the design, is maintained during the full life cycle of the system, and provides administrative support for the safety program of the full system.

[1] Supervisory general engineer.

[2] Senior research physical scientist.

[3] Mechanical engineer.

Pittsburgh Research Laboratory, National Institute for Occupational Safety and Health, Pittsburgh, PA.

• 4.0 *Safety Assessment*.—The independent assessment of the Safety File is addressed. It establishes consistent methods to determine the completeness and suitability of safety evidence and justifications. This assessment could be conducted by an independent third party.

• 5.0 *Safety Framework Guidance*.—It is intended to supplement the safety framework reports with guidance providing users with additional information. The purpose is to assist users in applying the concepts presented. In other words, the safety framework is *what needs to be done* and the guidance is the *how it can be done*. The guidance information reinforces the concepts, describes various methodologies that can be used, and gives examples and references. It also gives information on the benefits and drawbacks of various methodologies. The guidance reports are not intended to promote a single methodology or to be an exhaustive treaty of the subject material. They provide information and references so that the user can more intelligently choose and implement the appropriate methodologies given the user's application and capabilities.

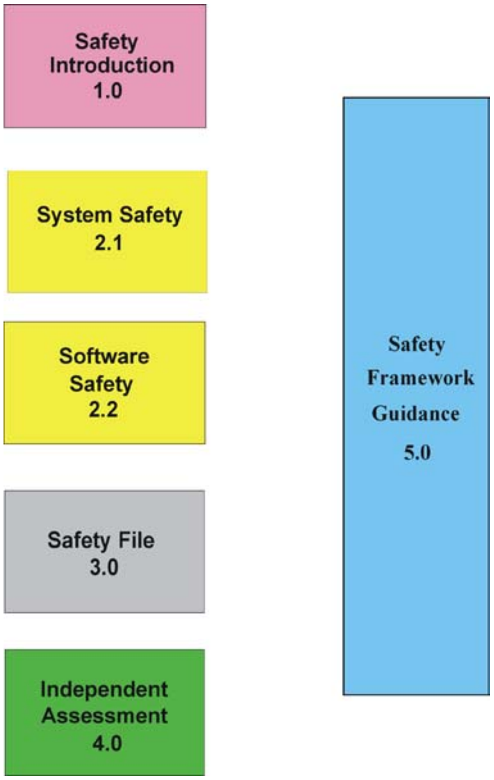


Figure 1.—The safety framework and associated guidance.

ACKNOWLEDGMENTS

The authors thank David C. Chirdon, Gerald D. Dransite, and Chad Huntley with the Mine Safety and Health Administration's (MSHA) Approval and Certification Center, Triadelphia, WV, for their assistance in developing this series of reports.

BACKGROUND

The mining industry is using programmable electronic (PE) technology to improve safety, increase productivity, and improve mining's competitive position. It is an emerging technology for mining that is growing in diverse areas including longwall mining systems, automated haulage, mine monitoring systems, and mine processing equipment. Although PE provides many benefits, it adds a level of complexity that, if not properly considered, may adversely affect worker safety [Sammarco et al. 1997]. This emerging technology can create new hazards or worsen existing ones. PE technology has unique failure modes that are different from mechanical systems or hard-wired electronic systems traditionally used in mining. PE includes microprocessors, embedded controllers, programmable logic controllers (PLCs), and the associated software.

The use of a safety life cycle helps to ensure that safety is applied in a systematic manner for all phases of the system; thus reducing the potential for systematic errors. It enables safety to be "designed in" *early* rather than being addressed after the system's design is completed. Early identification of hazards makes it easier and less costly to address them. The life cycle concept is applied during the entire life of the system since hazards can become evident at later stages or new hazards can be introduced by system modifications. The safety life cycle for mining is an adaptation of the safety life cycle in Part 1 of IEC 61508 [IEC 1998a].

System safety activities include identifying hazards, analyzing the risks, designing to eliminate or reduce hazards, and using this approach over the entire system life cycle. These system safety activities start at the system level and flow down to the subsystems and components. More detailed information on the fundamentals of system safety is presented by Sammarco et al. [1999].

This report incorporates some of the "best practices" for safety in the world and some of the latest international thinking on safety for PE. It uses a key group of standards selected from about 200 safety standards pertaining to PE. These key standards are listed in table 3.

Existing safety standards are built on collections of expertise and experiences (lessons learned) involving fatalities, injuries, and near misses of systems using PE. In general, standards also provide uniform, systematic approaches. History has shown standards to be an effective tool for safety [Leveson 1992]. Thus, by adapting existing standards, mining can build upon the valuable information captured in these standards documents.

1.0 Introduction

1.1 Document Conventions

This report follows a general format where major sections consist of an objective and associated recommendations. The formats are as follows:

Objective(s):
Recommendation(s):
NOTE:

The **NOTES** give brief clarification, reasoning, or guidance. More in-depth information is found in supplemental guidance documents.

1.2 Scope

The scope is "surface and underground safety mining systems employing embedded, networked, and nonnetworked programmable electronics." Software is a subsystem; thus, software safety is a part of the system's safety. Information on background, introduction, key documents, and additional definitions not covered in this document can be found in the System Safety document 2.1 [Sammarco and Fisher 2001].

1.3 General

1.3.1 These recommendations do not supersede Federal or State laws and regulations.

1.3.2 These recommendations are not equipment- or application-specific.

1.3.3 These recommendations do not serve as a compliance document.

1.3.4 These recommendations apply to the entire life cycle for the mining system.

1.3.5 These recommendations apply primarily to the safety-related parts of the system. However, many of the recommendations can also be applied to the basic system.

1.4 Software Safety Plan Framework

The concepts of safety life cycle and safety integrity level (SIL) are presented and developed in the System Safety document 2.1 [Sammarco and Fisher 2001]. The software safety plan framework builds on these concepts and provides specific recommendations for these requirements as related to software. A hazard analysis is required at the system level of mining systems employing programmable electronics to identify hazards or hazardous events. A risk analysis is then conducted to determine what should be controlled and to what degree (safety integrity level). This determination of an SIL at the system level leads in turn to an SIL value requirement for the programmable component of the system.

Each safety integrity level signifies a specific level of risk. The level of risk associated with a system determines which SIL is appropriate for that system. The determination of system-level risk is in turn based on a determination of the level of risk associated with each of the hazards identified for the system. The SIL framework specifies a set of requirements, procedures, methods, and techniques for each level commensurate with the specific level of risk addressed.

Three SILs comprise the Software Safety Plan (SWSP) framework. Three levels allow a range of minor, moderate, and major levels of risk to be addressed. This is a relatively easy classification to apply. As experience with the application of the framework to the mining industry grows, it may be desirable to break the range into more finely discriminated levels. The key is to not create more sharply differentiated levels than existing verification methods can support. In this framework, the range of requirements, procedures, methods, and techniques is specified in terms of which of the elements of the SWSP are required and the formality or rigor of the methods associated with the activities in each of the elements. For example, testing is a verification and validation activity required at each SIL. However, the testing objectives, the types of testing required, and the level of coverage required vary for each level.

Table 1 summarizes the framework by listing the elements of the SWSP required for each SIL. Table 2 further characterizes the framework by specifying the verification and validation (V&V) methods that can be used. The purpose of verification and validation is to demonstrate that the software is correct and complete with respect to the system and software design specifications and safety requirements. Table 2 lists types of verification and validation methods that can be used to accomplish the objectives at each framework level. Methods must be selected based on the objectives at each level and the kinds of software defects associated with those objectives, because some methods are more effective than others at detecting particular kinds of defects in the software with respect to design and safety requirements. The verification and validation of methods presented in table 2 are based on the Rome Laboratory Certification Framework [Scheper 1993, 1996] and IEC 61508 Parts 3 and 7 [IEC c,d,e,f,g]. See section 5 of this report for more information on verification and validation methods.

Table 1.—Software safety plan framework

SWSP element	SIL 1 (minor risk)	SIL 2 (moderate risk)	SIL 3 (major risk)
Software Development Plan	Statement of approach; life cycle requirements.	All requirements	All requirements.
Risk Analysis	All requirements	All requirements	All requirements.
Product Summary	All requirements	All requirements	All requirements.
System Architecture	All requirements	All requirements	All requirements.
System and Software Requirements	Address all requirements	Address all requirements; all requirements are traceable to V&V.	Address all requirements; all requirements traceable to V&V; all requirements validated.
System and Software Design Specification	Document	Document and trace to V&V	Document and trace to V&V; validate.
System and Software Implementation	Document	Document and verify	Document and validate.
Verification and Validation (V&V)	See table 2	See table 2	See table 2.
Management of Change	Required for production software.	Required for software under development and production software.	Required for software under development, production software, and maintenance.
Off-the-Shelf Software	Document	Document and verify	Document and validate.
Management Responsibility	All requirements	All requirements	All requirements.

Training	All requirements	All requirements	All requirements.
Maintenance	All requirements	All requirements	All requirements.

Table 2.—Verification and validation (V&V) methods for each SIL

V&V Method	SIL 1	SIL 2	SIL 3
Static Analysis:			
Requirements Review	Required	Required	Required.
Design Review	Required	Required	Required.
Code Review	Required	Required	Required.
Structure Analysis	Not required	Required	Required.
Boundary Value Analysis	Required	Required	Required.
Control Flow Analysis	Not required	Required	Required.
Data Flow Analysis	Not required	Not required	Required.
Error Anomaly Analysis	Not required	Not required	Required.
Functional Testing:			
Equivalence Partitions	Required	Required	Required.
Boundary Values	Required	Required	Required.
Special Values	Required	Required	Required.
Random Testing	Not required	Not required	Required.
Structural Testing:			
Statement Coverage	Required	Required	Required.
Branch Coverage	Recommended	Required	Required.
Multiple Condition Coverage	Not required	Required	Required.
Decision-to-Decision Path	Not required	Not required	Required.
LCSAJ Measures	Not required	Not required	Required.
Data Flow Coverage	Not required	Recommended	Required.
Performance Testing	Required	Required	Required.
Failure Mode and Stress Testing	Recommended	Required	Required.
Modeling and Simulation:			
Structure Diagrams	Not required	Required	Required.
Data Flow Diagrams	Not required	Required	Required.
Finite State Machines	Not required	Required	Required.
Formal Models	Not required	Not required	Recommended.
Performance Models	Not required	Not required	Required.
State Models	Not required	Required	Required.
Prototypes	Not required	Not required	Required.
Symbolic Execution	Not required	Not required	Recommended.
Formal Analysis	Not required	Not required	Recommended.
Interface Testing	Required	Required	Required.

2.0 Key Documents

2.1 This recommendation document is based on information and concepts from the documents listed in table 3. References for these standards can be found in the System Safety document 2.1 [Sammarco and Fisher 2001].

Table 3.—Key documents used for these recommendations

Standard identification	Title
IEC 61508 Parts 1-7	Functional safety of electrical/electronic/programmable electronic safety-related systems.
ANSI/ISA S84.01	Application of safety instrumented systems for the process industries.
ISA Draft Technical Report and TR84.0.02 - Parts 1-5	Safety Instrumented Systems (SIS); Safety Integrity Level (SIL) Evaluation Techniques.
MIL-STD-882C	Standard practice for systems safety program requirements.
UK Def Stan 00-58	HAZOP studies on systems containing programmable electronics.
ANSI/UL 1998, 2nd edition	Software in programmable components.

3.0 Definitions

The definitions are directly from IEEE Standard 610.12-1990 IEEE Standard Glossary of Software Engineering Technology; IEC 61508; and the *System Safety Analysis Handbook* [Stephans and Talso 1997]. A few definitions are adaptations or newly formed definitions specific to mining.

Critical Software - Computer software components and units whose errors can result in a potential hazard or in loss of predictability or control of a system.

Deliverable Software Safety Data - Data that are contractually required to be delivered to the customer.

Fail-Operational - Pertaining to a system that works with limited capacity and with known safety risk.

Fail-Safe - Pertaining to a system or component that automatically places itself in a safe operating mode in the event of a failure, e.g., a traffic light that reverts to blinking red in all directions when normal operation fails.

Management of Change - Discipline of identifying the components of an evolving system for the purposes of controlling changes to those components and maintaining continuity and traceability throughout the life cycle.

Noncritical Software - Software whose failure would not have an impact on safety, or would not cause large financial or social loss.

Nondeliverable Software Safety Data - All other data used to complete or support the safety effort, but not contractually required to be delivered to the customer. However, it may be required to be maintained on file and available for customer review (e.g., shop audit data and document review logs).

Reasonably Foreseeable Misuse - Use of a product, process, or service under conditions or for purposes not intended by the supplier, but which can happen, induced by the product, process, or service in combination with, or as a result of, common human behavior.

Risks-Addressed (RA) State - A state that is characterized by all reasonable foreseeable risks associated with the intended use of the product being addressed, such that the risk is reduced to a tolerable level.

Software - Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system.

Software Safety Integrity - Measure that signifies the likelihood of software in a programmable electronic system achieving its safety functions under all stated conditions within a stated period of time.

Software Safety Integrity Level - One of three discrete levels for specifying the safety integrity of software in a safety system.

Software Safety Validation - To ensure that the integrated system complies with the specified requirements for software safety at the intended safety integrity level.

Software Verification - To the extent required by the safety integrity level, to test and evaluate the outputs from a given software safety life cycle phase to ensure correctness and consistency with respect to the outputs and standards provided as inputs to that phase.

Supervisory Software - A computer program, usually part of an operating system, that controls the execution of other computer programs and regulates the flow of work in a computer system.

Tolerable Risk Level - Risk that is accepted in a given context based on the current values of society.

4.0 Software Safety

4.1 Product Summary

Objective:

4.1.1 To provide an overview of the programmable electronic equipment safety features, listing its capabilities and characteristics that are attributable to software.

Recommendations:

4.1.2 Sample operational scenarios and applications should be provided here.

4.1.3 Detailed instructions on how to use the equipment, including interfacing with the software and/or hardware, should also be provided here.

4.2 Software Safety Plan

NOTE 1: The Software Safety Plan (SWSP) specifies the safety requirements for the software in a programmable electronic system and describes how the requirements will be met and how it will be demonstrated that the requirements have been met. The SWSP also provides a description of the functionality and safety features of the software and describes the operating modes of the software. In describing the safety features of the software, the SWSP addresses how the software operating modes correspond to the system operating modes and how to specify for each software operating mode whether the corresponding system mode is a Risks-Addressed (RA) state. A risks-addressed state is characterized by all reasonably foreseeable risks associated with the intended use of the product being addressed, such that the risk is reduced to a tolerable level. The SWSP also addresses how the software triggers transitions between the system modes and identifies any of these transitions that are capable of resulting in a risk.

Objectives:

4.2.1 To plan and document the design, development, and verification of safety software.

4.2.2 To specify design, development, and verification processes and procedures (for example, see section 5.3.4 for a Software Checklist).

4.2.3 To specify safety requirements for software and microelectronic hardware components.

4.2.4 To specify and document engineering methods and techniques used in design, development, and verification.

Recommendations:

4.2.5 A statement of the methodology and approach to the software development, including design rationale, metrics to be collected, applicable standards, and the engineering methods and techniques employed should be developed.

4.2.6 A statement of how relationships and lines of communication are set up between organizational functions that may impact or have responsibility for tasks with system implications should be developed.

4.2.7 A qualifications summary of personnel involved in the development of software, including a statement of the minimum qualifications criteria required of members of the software development team, should be developed.

4.2.8 A statement of the level of authority developers have to implement the tasks necessary to complete the project should be developed.

4.2.9 A description of the relationship between safety and other functional elements of the system should be developed.

4.2.10 A description of the mechanisms by which concerns are or can be brought to light by project personnel should be developed.

4.2.11 A clear statement of the documents produced and the activities undertaken as part of the software development life cycle should be developed.

4.3 Software Life Cycle

NOTE 2: These processes are a subset of the overall system life cycle processes and activities. It is important that the development of requirements (including safety requirements), the design of the system to meet those requirements, and the verification points at the transition between requirements and design and between design and implementation be identified as key activities.

NOTE 3: The use of a software life cycle is required to ensure that safety is applied in a systematic manner, thus reducing the potential for systematic errors. It enables safety to be "designed in" early rather than being addressed after the software's design is completed. Early identification of hazards makes it easier and less costly to address them. The software life cycle concept is applied during the entire life of the software since hazards can become evident at later stages or new hazards can be introduced by software modifications (see table 4).

Table 4.—Example documentation structure for information related to the software safety life cycle

Software safety life cycle phase	Information
Software safety requirements	Specification (software safety requirements, comprising software safety functions and software safety integrity).
Software validation planning	Plan (software safety validation).
Software design and development:	
Software architecture	Description (software architecture design); Specification (software architecture integration tests); Specification (programmable electronic and software integration tests); Instruction (development tools and coding manual).
Software system design	Description (software system design); Specification (software system integration tests).
Software module design	Specification (software module design); Specification (software module tests).

Coding	List (source code);
	Report (software module test);
	Report (code review).

Software module testing	Report (software module test).
Software integration	Report (software module integration test); Report (software system integration test); Report (software architecture integration test).
Programmable electronic integration	Report (programmable electronic and software integration test).
Software operation and maintenance procedures	Instruction (user); Instruction (operation and maintenance).
Software safety validation	Report (software safety validation).

Software modification	Instruction (software modification procedures);
	Request (software modification);
	Report (software modification impact analysis);
	Log (software modification).

Concerning all phases	Plan (software safety);
	Plan (software verification);
	Report (software verification);
	Plan (software functional safety assessment);
	Report (software functional safety assessment).

Objective:

4.3.1 To define processes and activities that should be considered when developing software for use in mining systems or equipment.

Recommendations:

4.3.2 The software process should begin with a risk analysis.

4.3.3 All software process activities should be described and documented, including a description of the safety analyses and processes conducted during each life cycle.

4.3.4 All software life cycle activities should be associated with the system life cycle activities.

4.3.5 The software process activities should be identified with distinct entry points, exit points, and criteria for transitioning among activities.

4.3.6 Safety requirements for the programmable component should be met before process transitions occur.

4.3.7 Safety requirements for the programmable component should be traceable throughout the software process activities.

4.3.8 Work products (e.g., meeting minutes, analysis and test results, formal documentation, etc.) should be identified and associated with software process activities.

4.3.9 All software process activities should support the communication of issues that could impact the safety functioning of the programmable component, and safety issues should be communicated between staff and across life cycle phases.

4.3.10 The verification, validation, and testing activities in the software process should address errors at their source.

4.3.11 Staff responsibilities and sign-off authority should be described.

4.4 Risk Management

Objective:

4.4.1 To ensure that the safety considerations are consistently addressed by human (including administrative, procedural, and operational mechanisms), electromechanical (including electronics and traditional hardware), and software (including firmware) protective measures.

NOTE 4: The risk management activities should be specified in the system safety plan. This section extends those activities into the software development both by the identification of hazards at the system level that are related to software and by additional analysis of the programmable component and its software to identify any further software-related risks and to determine failure modes in the programmable component and software that could lead to the identified risks.

Recommendations:

4.4.2 A risk analysis for the programmable component should be conducted to determine what the risks are and the ways in which the software and hardware address or impact the identified risks. For each identified risk, a description of the severity, probability, and operational constraints should include—

- An analysis of provisions or alternatives that exist to eliminate or reduce the associated risks
- An evaluation of installation considerations and their possible impact on safety
- Identification of requirements to eliminate or reduce risks

- How software and hardware address or impact the identified risks
- Identification of the critical, noncritical, and supervisory sections of the software
- Identification of states or transitions that can result in a risk

4.4.3 The risk analysis should be based on the safety requirements for the equipment under control.

4.4.4 The analysis should identify the critical, noncritical, and supervisory sections of the software and should identify states or transitions that are capable of resulting in a risk.

4.4.5 The methods used to identify risks should be documented.

NOTE5: Examples of methods include brainstorming activity with software and hardware engineers, user experts, sales representatives, purchasers, and other personnel with specific expertise; the use of reference standards; and the use of field history and databases.

4.4.6 The mitigation methods should be described. Identified hazards should be eliminated, controlled, reduced, and/or warned against.

NOTE 6: Mitigation must not introduce an additional hazard into the system, such as operator warning messages that are difficult to understand or require a manual to decipher. Mitigation methods depend on the severity level and risk/benefit of residual risk or risk associated with the essential performance of the system. If a hazard is identified as credible, assume that it may occur and must be mitigated to achieve a level of risk as low as reasonably practical. A method of tracking the mitigation of a hazard throughout the life cycle of the software, such as a hazard traceability matrix, should be specified and documented.

4.5 System Architecture

Objective:

4.5.1 To describe the computerized system, the computer hardware architecture, the top-level software architecture, and the mapping of the software to the hardware.

Recommendations:

4.5.2 The architectural description should include the possible topologies for which the system can be configured (e.g., buses, LANs/WANs/CANs).

4.5.3 Block diagrams of the system showing a high-level view of the hardware architecture and the system interfaces (e.g., software-to-software interfaces and hardware-to-software interfaces) should also be included.

4.5.4 Information to be included in the description of the system architecture should include—

- References to the detailed design
- Top-level design (logical/functional), including external and internal interfaces
- Logical data scheme, the rationale for the architecture

- Design rationale and tradeoffs
- Relationships and interfaces between the levels of implementation, the allocation of software requirements to lower levels
- System architecture overview
- Software architecture
- Hardware architecture

4.6 System and Software Requirements

NOTE 7: This section specifies system and software requirements. This includes functional, performance, interface, and safety requirements.

Objectives:

4.6.1 To specify the functional, performance, interface, and safety requirements of the system as a whole.

4.6.2 To specify the functional, performance, interface, and safety requirements of the software.

4.6.3 To specify all interfaces between safety systems and any other system.

4.6.4 To include all modes of operation and maintenance.

4.6.5 To document the significance of hardware/software interactions.

4.6.6 To document the failure behavior and the required response.

4.6.7 To examine environmental conditions necessary to achieve functional safety.

4.6.8 To ensure that requirements are traceable to the lowest level of implementation.

4.6.9 To specify that partitioning analysis is provided in order to show the separation of the risk- and non-risk-related software functions.

Recommendations:

4.6.10 **Functional Requirements** address how the software is organized and what it does to accomplish the functional goals of the programmable component and, ultimately, of the mining equipment of which it is a component. Items addressed by the functional requirements should include, where appropriate—

- Inputs and outputs
- Functional decomposition, i.e., how the software is partitioned into subunits that address individual functional components

- Partitioning of software units across different computing hardware resources, such as parallel processors or different memory devices
- Distributed functions
- Safety and other critical functions
- Fault tolerance mechanisms (i.e., fault detection, fault handling, safe shutdown, and fail-operational mechanisms)
- Algorithms used to accomplish the assigned functionality
- Computations and data handling

NOTE 8: The equipment functional requirements are allocated to all components of the equipment, including the programmable component. Within the programmable component, functionality may be distributed between hardware and software components, in which case the distribution should be clearly defined.

4.6.11 Performance Requirements address how functional tasks are scheduled and allocated to physical resources. Items that should be addressed by the performance requirements include, where appropriate—

- Processing deadlines, especially real-time deadlines and timing requirements for interrupt services
- Safety timing constraints and deadlines
- Scheduling constraints such as prioritization of tasks
- Interrupt services timing requirements
- Specification of available processing resources, such as memory size and processor speed
- Constraints on resource allocations
- Estimates of processing requirements for application functions
- Effects of different mappings of software components to architectural resources
- Sensitivity of system performance to architectural parameters and resource allocations
- Resource utilizations by application function
- Processing workload throughput
- I/O and interprocessor communication throughput
- Effect of contention for resources on throughput and utilization
- Potential for bottlenecks and hot spots
- Execution times and performance overheads of operating system mechanisms
- Execution times and performance overheads of fault detection mechanisms
- Performance characteristics of software functions, including—
  a. Data dependencies and memory addressing patterns
  b. Data flow timing
  c. Real-time processing constraints, data rates, transport delays, and process update rates
  d. Control, scheduling, synchronization, and communications
  e. Throughput, memory, and bandwidth requirements
  f. Locality and interaction of references
  g. Maximum computing delays
  h. Maximum allowable performance degradation

NOTE 9: The objective of the performance requirements is to address the timing and resource constraints imposed by application requirements, safety requirements, computational loads, and the hardware characteristics of the programmable component.

4.6.12 Safety Requirements address how faults and errors are prevented, or if not prevented, how they are detected and handled to reduce risk.

A major source of risk derives from faults and errors, such as—

- Design faults, such as incorrect software algorithms or interfaces
- Coding faults, including syntax, incorrect signs, endless loops, etc.
- Timing faults that cause program execution to occur prematurely or late
- Memory faults, such as memory failure, insufficient memory, or memory overlap
- Induced faults caused by hardware failure
- Latent, user, input/output, range, and other faults that are only detectable when a given value or set of conditions occurs
- Failure of the programmable component to perform any function at all

NOTE 10: The objective of the safety requirements is to address risks that occur in the software or in the process used to develop and maintain the software. It is also important to be aware of compatibility problems associated with substitution of different components after the initial design evaluation. For example, if one were to switch memory or microprocessor suppliers, subtle changes could be introduced.

4.6.12.1 General Software Safety Requirements

- Measures to detect failure in software during operation and prevent the occurrence of a risk are required.
- The software will be capable of identifying and responding to states and conditions that can result in a risk. This requires that measures such as initialization and run-time checks, fail-safe and fault-tolerant concepts, and built-in tests be used.
- The software will maintain an RA state when a condition that can result in a risk is detected.
- Where a multitasking system is employed, tasks will be scheduled and resources will be allocated to tasks based on the scheduling frequency required for the tasks, the criticality of the tasks, and the resources used by the tasks in such a way that the principle of the more critical the risk, the higher the response priority is maintained. The impact of each of these factors on the ability to address identified risks will be specified.
- Nonterminating and nondeterministic states and error states, such as undefined branch conditions, zero division, and underflow/overflow, will be prevented, detected, and resolved.
- All variables will be set to initial values before being used by any instruction.

4.6.12.2 Requirements for Critical and Supervisory Sections of Software

Steps should be taken to determine which portions of the software are safety critical and which are not. It is in the best interest of the developer to keep the safety critical portions as small, simple, and concise as possible.

- Software should be initialized to a documented RA state.
- All critical and supervisory sections of the software should be partitioned, both physically and logically from the noncritical sections. If it is not possible to partition the software, all of the software is considered critical.
- Means should be employed to avoid or to detect and recover from memory usage and addressing conflicts.
- A supervisory section or mechanism should maintain control of the execution of the software at all times during the operation of the programmable component.
- A fail-safe or fail-operational procedure should be initiated if a failure occurs in a critical or supervisory section.
- Means such as error checking and/or correcting procedures should be employed to preserve the integrity of data and instructions used by critical and supervisory sections.

4.6.12.3 Requirements for Measures To Address Microelectronic Hardware Failures

Means should be used to address all microelectronic hardware failure modes identified by the risk analysis. Some examples are—

- CPU registers, instruction decoding and execution, program counter, addressing and data paths
- Interrupt handling and execution
- Clock
- Nonvolatile and volatile memory and memory addressing
- Internal data path and data addressing
- External communication and data, addressing, and timing
- Input/output devices, such as analog I/O, D/A, and A/D converters, and analog multiplexers
- Monitoring devices and comparators
- Application-specific integrated circuits (ASICs), gate array logics (GALs), programmable logic arrays (PLAs), and programmable gate arrays (PGAs) hardware
- Clear and concise fault coverage specification will be provided that indicates the faults, the impact of the faults on risks, and the methods used to detect the faults.
- Analysis of possible combinations of microelectronic hardware failures, software faults, and other events that are capable of resulting in a risk will be conducted. This includes, for example, microelectronic hardware failures that cause software faults that are capable of resulting in a risk.
- The microelectronic hardware components should meet the operational life requirements of the equipment.
- The microelectronic hardware components should have an acceptable failure rate over the expected operational life.

- The fault detection and handling mechanisms should provide an acceptable level of coverage given the impact on safety of the identified faults, i.e., the detection likelihood and execution frequency are sufficient to mitigate risk in accordance with the safety impact.

NOTE 11: Some common types of detection methods include replication (redundancy), timing (watchdog), coding (parity, checksum, CRC), and diagnostic (BIT, test pattern). Lists of techniques by fault type detected can be found in tables A.1 to A.19 in IEC 61508 Part 2 and in table A.2.1 in appendix A of the 2nd edition of UL 1998. Definitions of these techniques can be found in annex A of IEC 61508 Part 7 and in section A7 of appendix A of the 2nd edition of UL 1998.

4.6.13 Requirements for the Interface to the Controlled Equipment

- In the event of power interruptions, the software should maintain a documented RA state.
- When initialization is allocated as a software function, the software should initialize the controlled equipment to a documented RA state.
- Whenever the software terminates, the controlled equipment should maintain a documented RA state.
- Any procedure or instruction intended to halt the programmable component should maintain the controlled equipment in an RA state.

4.6.14 Requirements for the User Interface

- Time limits and other parameters of the software should not be changeable by a user in such a way that the intended execution of critical and supervisory sections is adversely affected.
- The time limits and other parameters of the software that are intended to be configured by qualified service personnel should be prevented from being changed to the extent that the intended operation of the critical or supervisory sections of software is adversely affected.
- The software should require two or more user responses to initiate an operation that is capable of resulting in a risk.
- Input commands that are capable of resulting in a risk when executed should not be initiated without operator intervention when those commands are received from an external source and should require two or more user inputs.
- Incorrect input should not adversely affect execution of critical sections of software.
- The software should provide for user cancellation of the current operation and return the programmable component to an RA state.
- User cancellation of the current operation should require a single input by the user.
- Cancellation of processing should leave the software in an RA state.

4.7 System and Software Design Specification

Objective:

4.7.1 To detail the design of the system and software and how the two interact and rely upon each other.

Recommendation:

4.7.2 Information required in the design specification should include—

- Identification of the methods for achieving independence of risk- and non-risk-related functions
- Identification of the techniques necessary to achieve the desired performance
- Inclusion of the overall hardware and software architecture
- Inclusion of the interfaces between software and external users (physical data format, message descriptions, protocols and priorities)
- Documentation of design rationale and tradeoffs
- Inclusion of the mapping between logical or functional design and detailed design units
- Inclusion of the physical design into compilation units, which includes the inputs and outputs, functions, data descriptions and relationships, control and signal flow, error handling and algorithms

4.8 System and Software Implementation

Objective:

4.8.1 To specify implementation requirements.

NOTE 12: These include collecting historical safety data, the deliverable software safety data, making access to the nondeliverable safety data available to the customer, and other record-keeping functions. "Making access" means that the documents are viewable by the customer; it does not imply releasing hard copies to the customer.

Recommendations:

4.8.2 The system developer should use historical safety data from the customer, subcontractors, MSHA, and other corporate knowledge or data. Mine accident data available from the MSHA Web site (www.msha.gov) should be reviewed to identify hazards and risks that could apply to the system. This combined experience should provide a historical safety database for software safety engineers. The data should be used in design reviews, trade studies, determining exposure times, and other safety analyses on the project.

4.8.3 Deliverable software safety data should include a—

- Software Safety Program Plan
- System Safety Program Review (SSPR)

- Hazard Tracking and Risk Resolution
- Safety Review of Engineering Change Protocol (ECP) and Deviations in software specifications
- Top-Level Hazard Analysis
- Detailed Design Hazard Analysis
- Code Level Hazard Analysis
- Software Safety Testing
- Safety Assessment Report

4.8.4 Nondeliverable safety data with customer collaboration could include—

- A hazard log
- System design notes
- Historical safety data summaries
- SSPR action items
- A lessons-learned database

4.8.5 Software integrity after delivery or installation is the responsibility of the customer. The system developer cannot be held responsible for software used outside specified domains for unspecified purposes. These hazards are out of scope. Where special integrity requirements exist to maintain the safety of the software functions, software safety should provide these in the remarks column of the SSHA or software hazard analyses.

4.8.6 Installation hazards should be assessed in the maintenance concept phase or on-site and readdressed as they occur.

4.8.7 When transitioning from old to new software, the system developer should identify specific procedures that will preserve the integrity and safety of the software, if applicable.

4.9 Verification and Validation

Objective:

4.9.1 To describe the plans and procedures used for testing.

NOTE 13: Some of these may be addressed through references to other documents, including coding standards, design standards, etc., that are to be followed during development.

Recommendations:

4.9.2 Information to be provided should include—

- Verification, validation, and testing dates
- Who carries out the verification, validation, and testing
- Identification of relevant modes of operation
- Identification of risk-related systems that need to be validated for each mode

- Technical strategy for verification, validation, and testing (e.g., analytical methods, statistical tests)
- Measures, techniques, and procedures used for confirming that each safety function conforms with the requirements
- The environment in which verification, validation, and testing activities take place. A description of the facilities, equipment, and software used for testing of functions will be provided.
- Pass/fail criteria
- Maintain a checklist
- Traceability matrix
- Policies and procedures for evaluating the results
- Failure resolution policies
- Test objectives, procedures, criteria, expected results for each level of testing (e.g., unit, integration, black-box, regression, and system acceptance testing)

NOTE 14: Software safety personnel work closely with software quality assurance personnel to ensure proper definition of verification assurances. Software safety also works with test engineering to identify safety verification and validation testing and ensure that test procedures and facilities are safe. The generic historical faults test and evaluation (T&E) finds in similar systems should be analyzed for hazard contribution and safety requirement specifications derived. This early analysis by software and system safety should reduce the initial error set that T&E usually deals with and enhance their starting position with better data on system functionality, testability, and hazardous environments.

4.9.3 Software Safety Requirements

Wherever safety requirements are given and safety cannot be shown by analysis, comparison, or planned testing and demonstration, software safety should provide test checklists, vectors, or consultation on what is to be tested, how, and what success criteria are to be used. These inputs should be based on the inferences drawn from other hazard analyses and experience.

4.9.3.1 Verification Requirements To Document Safety

The results of software verification testing should be reviewed for safety impact. Safety deliverables, audits, and meeting minutes should normally suffice as assurances of verification.

4.9.3.2 Hazard Tracking Log

The hazard log should assist in documenting the coverage of all identified hazards by analyses, test, or demonstration. These secure data should be available to project personnel, and procedures should be in place for notification to System and Software Safety Engineering when entries are made.

4.9.4 Test results should provide evidence that the Verification, Validation, and Test Plans and Procedures have been followed and carried out. Information to be provided should include—

- Description of how the plan was carried out
- Summary of results of all analyses and tests conducted, including references to locations of detailed information

- Evidence of adherence to the standards that were established
- Details of test failures and their resolutions
- Evidence of maintaining problem reports for the life cycle of the software that identify and record the resolution of the software's anomalous behavior, process noncompliance with the plan and/or noncompliance with the standard.

4.9.5 Software analysis should document the software design and code analysis. Design and code analysis should demonstrate—

- Correctness and completeness with respect to the safety requirements for the controlled equipment
- Coverage of each decision and function that is capable of involving a risk
- That fail-safe and fail-operational procedures bring the product to an RA state
- That the scheduling requirements are met and safety functions meet the timing constraints specified by the safety requirements for the programmable component
- The integrity of the partitions between supervisory, critical, and noncritical sections of software
- That partition violations caused by occurrences such as data handling errors, control errors, timing errors, and misuse of resources do not occur
- Consistency in the data and control flows across interfaces
- That the critical and supervisory sections only perform those functions that they are intended to perform and do not result in a risk

NOTE 15: The safety or critical components should be identified so that they may be used to assess hazard coverage and, later, proposed changes. Typical identification includes resident hardware component or storage location, system level and software task, software configuration item, software module, data structure, interface, and communication protocol. Each safety software component should be assessed, tracked, analyzed, and tested at the level of rigor commensurate with the assigned level of risk.

NOTE 16: The results of analyses, as a measure of completion of work, are derived from the safety goal of hazard coverage. Each analysis results in a better database of hazards that feeds into the next phase of development, some tests for later stages, recommendations for new or modified safety requirements specifications, and a hazard analysis report.

NOTE 17: Safety requirements should be mapped through the software development. Tracing may stop where analysis, test, demonstration, or examination fulfills the safety requirement. This is to establish and measure hazard coverage and show closed-loop tracking of hazards.

NOTE 18: Safety components should be identified to assess the hazard coverage of interfaces and later software and hardware changes. Dependencies may be used for common cause and zonal effects.

4.9.6 Tests of the software should be conducted and test results documented to evaluate that the software only performs those functions for which it is intended and does not result in a risk.

4.9.7 Test cases should be developed based on the risk analysis, the documented descriptions of the software operation and safety features, and the software analysis. These tests should be conducted to demonstrate—

- Correctness and completeness with respect to the safety requirements for the programmable component

- Coverage of each decision and function that is capable of involving a risk
- That fail-safe and fail-operational procedures bring the product to an RA state
- That the scheduling requirements are met and safety functions meet the timing constraints specified by the safety requirements for the programmable component
- The integrity of the partitions between supervisory, critical, and noncritical sections of software
- That partition violations caused by such occurrences as data handling errors, control errors, timing errors, and misuse of resources do not occur
- Consistency in the data and control flows across interfaces

4.9.8 The outputs that the software generate to control equipment should be tested to determine their effects on the equipment, based on the expected output.

4.9.9 Failure mode and stress testing should be conducted to verify that software responds correctly and without resulting in risks to foreseeable single failures. Failure mode and stress testing should include consideration of the following:

- Operator errors that are capable of adversely affecting the intended operation or the control of the programmable component
- Microelectronic hardware component faults
- Errors in data received from external sensors or other software processes
- Failures associated with the entry into, and execution of, critical and supervisory sections of software
- Negative condition branch
- Other processes and procedures that are capable of adversely affecting the intended operation of the software

4.9.10 Test cases should include the following:

- Out-of-range
- Boundary condition
- Type mismatched values for parameters at which decisions are made

NOTE 19: Some experts are also encouraging use of "free-form" testing. It is unstructured testing at the end of normal testing, where a user (expert as well as novice) tries to break the system.

4.9.11 Failure mode tests should address all foreseeable faults identified in the risk analysis.

NOTE 20: Hazards traced to software should be analyzed and therefore reported in the software hazard analyses. In all cases, software safety testing should be conducted. The analyses or experience should allow some derivation or guidelines for test engineers to identify their own test cases. There may be firmware, ASICs, which are insufficiently test probed, and analysis may be accomplished at the VHDL (which stands for VHSIC Hardware Description Language) level. VHDL is a programming language that has been designed and optimized for describing the behavior of digital circuits and systems.

4.10 Management of Change

Objectives:

4.10.1 To describe how changes to the software and hardware are managed.

4.10.2 To describe how the changes to the system are addressed when the proposed change results in a system risk.

4.10.3 To show that all changes to the system are analyzed and evaluated to determine if there will be an impact on the functioning of the system.

NOTE 21: Management of Change covers all forms of software used throughout the software development life cycle, e.g., development tools (such as compilers, linkers, assemblers, loaders, debuggers, Management of Change tools), analysis tools (such as simulators, CASE tools, and static analysis software), test software, (such as simulators, penetration testing programs, special test data and test report/summary generation). Small changes tend to be error-prone. Therefore, it is even more important that all changes be treated cautiously.

Recommendations:

4.10.4 Information on Management of Change to be provided should include—

- A detailed plan on how changes to the software will be managed, when they will be implemented, how they will be reviewed, and what type of validation and verification will be done
- Records documenting problems, changes made, at what phase of the life cycle, etc., in the Management of Change plan
- A description of the initiation, transmittal, review, disposition, implementation, and tracking of discrepancy reports and change requests
- Proof (i.e., convincing evidence) of historical traceability with respect to the configuration identification scheme
- A description of the configuration identification scheme, responsibilities, and activities used to maintain and control baseline products
- A description of the methods and activities employed to formally control receipt, storage, handling, and release of configurable items

4.10.5 Changes to parameter settings and data should not create a risk or impact a risk that has previously been identified, other than to reduce or eliminate it.

4.10.6 Changes or patches to the software should not create a risk or impact a risk that has previously been identified, other than to reduce or eliminate it.

4.10.7 Documentation should be reviewed to determine that it correctly reflects safety changes that have been made in the software.

4.10.8 There should be procedures to maintain and control software changes to the configuration of the programmable components and critical and supervisory sections of software to facilitate traceability.

4.10.9 Changes to the system configuration or to the supervisory software should be considered as changes to the software.

4.10.10 Software should contain an identifier showing its revision or version and intended application.

4.10.11 The unique identifier should be computed as a function of the critical and supervisory sections of the software.

4.10.12 Each time a change or patch is incorporated in the software, a new unique identifier should be assigned.

4.10.13 Documentation should include sufficient information to identify each item that is investigated with the software.

NOTE 22: For example, identification of software elements should include the version number, release number, and date.

4.11 Off-the-Shelf (OTS) Software

Objective:

4.11.1 To provide a list of items to consider when off-the-shelf (OTS) software is interfaced with the manufacturer-supplied software. These recommendations also apply to contract software, third-party software, and tools.

Recommendations:

The following should all be developed:

4.11.2 The name and version/revision identifier of the OTS software.

4.11.3 The name of the OTS software provider.

4.11.4 A description of the purpose for which the software is being used.

4.11.5 A clear description of the function provided by the software.

4.11.6 For OTS software that performs supervisory section functions or is used by the supervisory section of software, the safety requirements for critical or supervisory software apply. The functionality of the OTS software should be subject to risk analysis.

4.12 Management Responsibility

Objective:

4.12.1 To specify the management activities and processes for software development throughout the system development life cycle.

Recommendation:

4.12.2 The software development activities and processes are derived from those specified for the system in the System Safety Plan, but are tailored for software development. These activities are coordinated by and report to the appropriate system development management structure. See NIOSH System Safety document 2.1 [Sammarco and Fisher 2001] for more information.

4.13 Training

Objective:

4.13.1 To address the issues of both the programmer and operator (user).

Recommendations:

4.13.2 The competency of persons, including subcontractors, involved in the critical safety life cycle and management activities for software needs to be addressed.

NOTE 23: This means specifying the technologies used and the minimum level of training in those technologies that the software development staff is required to have. For example, specifying the number of years of programming experience in a specified computer language might be required. Where new technologies are to be used, or where staff does not meet the minimum requirements, the training should specify the type of training that will be provided to bring staff to the required level.

4.13.3 The competency of persons whose use of the mining equipment requires knowledge of or interaction with the software needs to be addressed.

NOTE 24: This means specifying the kind of training that is required for users of the system to ensure that they can safely operate the equipment. This information should also be included in the system-level training and operation documentation.

4.13.4 New operators should be trained as intensively as the first operators.

NOTE 25: As the system ages and the mechanical parts wear, the characteristics of the system may change.

5.0 Information on Methods and Techniques

5.1 Risk Analysis Methods

Objectives:

5.1.1 To identify system hazards.

5.1.2 To trace system hazards to software.

5.1.3 To trace the safety software threads.

5.1.4 To determine mitigation methods.

NOTE 26: Risk analysis refers to the identification of hazards, the assessment of risk associated with those hazards, and the specification of hazard mitigation techniques. It is composed of four steps: hazard identification, hazard analysis, risk estimation, and risk control. Risk analysis starts at the beginning of product development and is revisited at each stage of development.

Recommendations:

5.1.5 Hazard identification uses brainstorming activities, reference standards, and field history to develop lists of hazards associated with a system.

5.1.6 Hazard analysis determines how a hazard could arise during the operation of the system and identifies the system components that are associated with the hazard.

5.1.7 Risk estimation assigns a measure or ranking to each hazard with the goal of focusing mitigation efforts. The ranking depends on characteristics of the hazard such as its likelihood of occurrence, the severity of its effects, the level of exposure to the hazard, and the balance between the risk and the benefit to the user.

NOTE 27: The level of exposure is based on how many of the systems are produced, who will use it, how many people will use it, and the skill level of the users.

NOTE 28: One method of predicting risk likelihood is to assign probabilities to events in a fault tree. Where software is involved and software fault events are included in the fault trees, probabilities can be assigned to the software events. However, the basis for assigning probabilities to software events is not as well-founded as that for hardware events. One possible basis for assigning probability to software events is the frequency of execution of the particular software components. Thus, a fault in a software component that is executed frequently is assumed to be more likely to be encountered than a fault in a software component that is rarely executed. Given the difficulties of assigning probabilities to software events, the key concern at the software level is to identify where software impacts or is related to a risk or hazard and to identify the specific software component or structure associated with the risk/hazard.

NOTE 29: Risk analysis leads to a safety integrity level (SIL) determination at the system level, which in turn leads to an SIL value requirement for the programmable component. The SIL requirement for the programmable component leads, in effect, to a reliability requirement, a fault coverage requirement, and sometimes a fault tolerance requirement for the hardware, but only to a "development and test rigor" requirement for software. The limitations of, and the lack of general consensus on the validity of, existing methods for predicting software reliability make it problematic for an SIL value to result in a reliability requirement for software.

NOTE 30: The approaches of ANSI/UL 1998-2nd edition and IEC 61508 place requirements on the completeness and rigor of the development and the verification and validation methods for software that are commensurate with the SIL requirement for the system. Thus, the requirements for completeness and rigor in the methods increase with the SIL value required for the system. However, there is not a large body of objective evidence on the relative effectiveness of methods that can be used to precisely rank the methods on effectiveness; the ranking is generally based on the formality and scope of the method. The ANSI/UL 1998-2nd edition approach, therefore, emphasizes the connection between identified risks that are linked to software, potential software defects that can lead to the identified risks, and methods aimed at avoiding or detecting those potential defects.

5.1.7.1 Fault-Tree Analysis (FTA) is a deductive reasoning technique that starts with an identified hazard (from the hazard list) and traces the pathway from the hazard into the product by determining events and the logical combination of events that lead to the hazard. An example of a fault tree is shown in figure 2. The event at the top is an identified hazard. This hazard can be caused by the occurrence of either of two events, event 1 or event 2. Event 1 in turn is caused by the occurrence of both of two events, wrong data and event 3. The wrong data event is caused by either the programmable component failure event or the sensor failure event. The programmable component failure event is caused by the software failure event or the hardware failure event. Thus, a software failure can lead to wrong data, which in turn can lead to a hazard if event 3 also occurs.

5.1.7.2 Failure Mode and Effects Analysis (FMEA) is an inductive reasoning technique that starts with a software component and traces a pathway to a hazardous event. This is used in combination with FTA to trace a software component to a hazardous state.

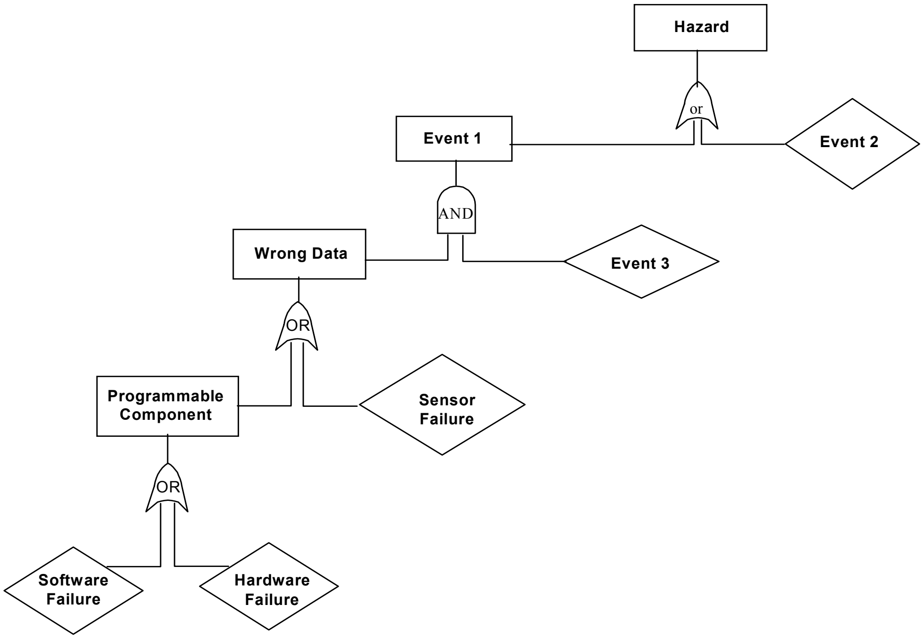


Figure 2.—Example of fault tree.

Failures of Component Compute Values				
Failure Type	**Failure Mode**	**Failure Effect**	**Where in Code**	**Mitigation**
Wrong Data	Out of range value	Branch condition fails and wrong code is executed	Data variables CurValue, NewValue	Assertions in code
	Inaccurate value	Incorrect output is delivered	Data variables CurValue, NewValue	Redundant data

Figure 3.—Example of FMEA.

Figure 3 shows a tabular method for documenting an FMEA.

5.1.7.3 Other Techniques. The System Safety Society's *System Safety Analysis Handbook* [Stephans and Talso 1997] contains a compendium of hazard identification and evaluation techniques, including techniques for system, hardware, software, and human factors issues. Each technique is described, including the process to apply the technique, areas of applicability, and thoroughness. The degree of preparation required to use the technique and the level of difficulty in applying it are also given.

5.1.8 Hazard Mitigation. Hazard mitigation is the process of eliminating, controlling, reducing, and/or warning against identified hazards. Specification of hazard mitigation techniques should take into account that an unanticipated combination of events may lead to a hazard. It is especially important to ensure that a mitigation technique does not introduce an additional hazard into the system. For example, an operator warning message that is difficult to understand may cause the operator to make an error.

5.2 Requirements and Design Specification Methods

Objective:

5.2.1 To describe requirements and design specification methods.

NOTE 31: Requirements and design specification methods vary from basic text documents to formal descriptions from which certain properties of a system can be proven. For safety applications, it is important to be able to verify and validate that safety requirements are being met at each stage of the development. This requires methods of specification that can be analyzed and that provide traceability of safety requirements down into the code. The methods should also be able to be used to specify testing requirements.

Recommendations:

5.2.3 Design Considerations

There are a number of practices when designing system software that can enhance the safety of the software.

5.2.3.1 Modularize Software

The smaller and less complicated the pieces of code, the less likely that an error can be made and the easier the code is to maintain. This is because the human mind can only comprehend a certain level of complexity and still be effective. It is paramount, however, that a good, robust software architecture be used to minimize the likelihood of interface problems.

5.2.3.2 Separation of Safety and Nonsafety Critical Functions

Keeping the safety and nonsafety critical software separated minimizes the possibility of an error introduced into the nonsafety critical part from corrupting the safety critical portion of the code.

5.2.3.3 Keep Safety Critical Code Small

The smaller the safety critical portion of the code, the easier it is to program it without error, debug it, and analyze it. Conversely, the more complex the safety critical code, the harder it is to program; thus, it is easier to unintentionally introduce errors. It is also exponentially more difficult to analyze.

5.2.3.4 Remove Extraneous Code

Many times, while coding, a programmer will leave a portion of code in place that is no longer needed and either comment it out or break before it. While in the short term this may be necessary, it should be removed before the software gets too far into testing. It is too easy to make an editing mistake and have a functioning piece of code now executing that should not be. Needless to say, this will most likely have an undesirable effect.

5.2.3.5 Avoid Novel Solutions to the Design

Programmers are a special breed, and many times they get bored with repetitive tasks. One of their favorite pastimes to break the boredom is to find new, novel, obtuse, or complex ways to do the same job that a simple piece of code could do. It is much more likely that an error can be introduced in such a way that it makes debugging, documentation, and maintenance much more difficult. That is not to say that a novel design cannot be a good one. Many times, the most novel design is elegant in its simplicity.

5.2.3.6 Use Static and Dynamic Code Testing

When testing software, the testing methods break down into two types: static and dynamic. With static testing methods, the software is reviewed (i.e., inspection, walkthrough) by trained professionals with specific knowledge and responsibility about the software development. With dynamic testing methods, the software is exercised by a computer (i.e., testing based on coverage, boundary testing, random testing). This type of testing generally requires detailed test planning and development of test cases. Generally, it takes a combination of static and dynamic testing to reach the desired quality goals. For more information about different testing methods, see appendix A.

5.2.4 Formal Specification

Formal specification of requirements is recommended and in some cases required for safety applications. There are also different methods for expressing and verifying the formal specification, including state diagrams, which are subject to analysis using graph theory or statistical methods, such as Markov analysis, and mathematical expressions and logical schema, which are subject to mathematical proof.

5.2.5 Formal Considerations

Some formal methods are:

- State-Machines [Jaffe et al. 1991; Modugno et al. 1997; Leveson et al. 1994; Heimdahl and Leveson 1996]
- System Fault Trees and Tri-State Models [Knight 1993; Knight and Kienzle 1993]
- Mathematical Logic [Jacky 1995]
- Cleanroom [Trammell et al. 1999]

NOTE 32: The first three methods were developed in university settings and applied to software development projects. The fourth is a commercially available method. All four methods are further described in the cited references.

5.2.6 Design specification methods that support analysis and verification and validation are commonly model-based. Many of the models are associated with particular methodologies, such as structured design and object-oriented design, or particular architectures, such as data flow, call and return, or independent components.

Table 5 lists some types of models that can be used for design specification and analysis.

Table 5.—Models for design specification and analysis

Model name	Description
Algorithm Model	A description of an algorithm containing information on input/output data, functional decompositions, computations, partitionings, and data dependencies. Can be implemented as a CASE model.
Workload Characterization Model	A description of the processing and communication workload, including data dependencies and memory addressing patterns, data flow timing, real-time processing constraints, data rates, transport delays, process update rates, estimated operation counts, and memory access counts.
Computational Model	The workload characterization of software processes embedded in graph models through a parameterized attribute definition language.
Data/Control Flow Model of Software Processes	Directed graph with Petri-net semantics. (See IEC 61508-7, section B.2.3.3, for an explanation of Petri-nets.)
Hardware Architecture Model	A structural model of the hardware architecture as a directed graph.
Performance Model	Constrained process graph that is a mapping of a data/control flow model of software processes to a structural model of hardware architecture and that has performance characteristics of architecture embedded in a graph model through an attribute definition language.
Functional Simulation Model	Code fragments used to simulate functionality of system components.
System Operational Model	A description of the operational environment that the software component will be a part of.
Behavioral Model	An object-oriented model that captures data and control flow, simulates the actual processing of data at a functional level, injects and reacts to faults, and integrates a large number of submodels that represent hardware architectural components, operating system and scheduling functions, hardware- and software-implemented dependability and safety mechanisms, and application algorithms.

Tables 6 through 8 list the types of information used in the models.

Table 6.—Software information for models

Type of information	Specific data items
Inputs/Outputs	Characteristics Structures Ranges Quantity Rates Sensors/actuators
Functional Decomposition	Description Inputs/outputs Processing workload and data flow between subfunctions Time constraints
Subfunctions	Inputs/outputs (sensors/actuators) Time constraints Processing or computations Transport delay Process update rate Jitter
Partitioning to Parallel Processors	Minimum granularity Dynamic load balancing requirements
Distributed Functions	Control Scheduling Synchronization Communications Data flow timing, throughput, bandwidth, latency, and memory Locality and interaction of references
Safety Functions, Critical Functions, and Fault-Tolerant Mechanisms.	Maximum computing delays Critical state information Mechanisms for improving robustness in the presence of faults Maximum allowable performance degradation Fault tolerance levels Fault avoidance methods Software quality assurance methods

Table 7.—System resource information for models

Processing Modules	Information
Power modules Processor modules Memory modules Timing/clocking modules Processor cache Data exchange Special-purpose modules (e.g., test interface, fault handling)	Functional specification Interfaces Timing requirements Estimate of design complexity and logical structure Fault classes and failure rates Processing speed Memory read/write times Capacities Volatile control registers and memory Software-resettable hardware Clock-deterministic hardware Phase-locked oscillators Redundancy level Number of spares Size, weight, power
Data/Communication Modules	**Information**
I/O communication links and modules (including sensors and actuators) Data acquisition and distribution modules Interprocessor communications links Internal processor bus Data buses	Topology and interconnections Protocols Performance Message structure Path allocation Timing, latency Redundant paths Transfer rates Capacity Bandwidth Communications overhead Sensors/actuators: specification of each, type, proposed redundancy, number of each type, number of bits Fault classes Failure rate for each mission phase Online test and calibration requirements
Special Purpose Features	**Information**
Fault Models	Type Frequency Mission phase or technology dependence Error characterization: count, origin, activity, duration extent, temporal behavior and cause of multiples

Fault Detection and Handling Mechanisms	Local and global error detection, error masking, and fault containment mechanisms Consistency Redundancy Fault-tolerant power Fault-tolerant clocking Local and global redundancy management, recovery and reconfiguration strategies Compensation strategies for delays and/or lost information Data consistency protocols Sparing strategy Function migration/reassignment Protection of critical functions Synchronization and startup Self-test State alignment Partitioning into hardware and software Detection of faults in fault-tolerant mechanisms Exception handling Times and effectiveness measures Fault models/sets addressed Assumptions: effectiveness of error detection, required times to detect, recover, and reconfigure

Table 8.—System software information for models

Type of information	Specific data items
Task Scheduling	Description of control characteristics: distributed, central, hierarchical
I/O Services	
Interrupt Services	Function overheads
Memory Management	Expected execution times per function
Utilities	Function response times
Interprocessor Communications Services	Uncertainty in response times
RAM Alignment Software	Software semaphores
Initialization Software	Reserved addresses in memory
Synchronization Software	Criticality of function
Functions Related to Safety	

5.3 Verification and Validation Methods

Objective:

5.3.1 To describe methods for software verification and validation.

NOTE 33: There are a wide range of methods for software verification and validation. Most are targeted to particular testing and analysis objectives, and no single method is suitable for all applications or for all objectives. Therefore, one must carefully choose among the many available methods those that will provide the level of confidence required in the correctness and completeness of software.

Recommendations:

5.3.2 One way of guiding the selection is through the use of standards such as IEC 61508 Part 3, which contains guidance on methods to use that are appropriate for each SIL level. This guidance is provided in the form of tables that list methods and indicate for each method whether it is not recommended, recommended, or highly recommended at that level.

NOTE 34: IEC 61508 Part 7 contains a description of each method.

5.3.3 Another way is to target methods to particular kinds of software defects. A developer should use in-house application knowledge to identify the types of defects encountered, then select verification and validation methods that are effective at finding those defects.

5.3.4 As a minimum, checklists can be used as a technique to list hazards, stimulate thinking, and to pass on lessons learned from mistakes and experience. They can also be used to systematically list design procedures and practices, thus reducing the likelihood of errors of omission. The following checklist has been adapted from published material, and examples are given based on analysis of actual MSHA accident data.

Software Checklist

Within the software specification, is there a clear and concise statement of— (1) Each safety function to be implemented? (2) The information to be given to the operator at any time? (3) The required action on each operator command, including illegal or unexpected commands? (4) The communications requirements between the PES and other equipment? (5) The initial states for all internal variables and external interfaces? (6) The required action on power-down and recovery (e.g., saving of important data in nonvolatile memory)? (7) The different requirements for each phase of plant/machine operation (e.g., startup, normal operation, shutdown)? (8) The anticipated ranges of input variables and the required action on out-of-range variables? (9) The required performance in terms of speed, accuracy, and precision? (10) The constraints put on the software by the hardware (e.g., speed, memory size, word length)? (11) Internal self-checks to be carried out and the action on detection of a failure?	
Does the software contain adequate error detection facilities allied to error containment, recovery, or safe shutdown procedures? *Example:* Erratic signals caused by moisture collecting on a control keypad were interpreted by the programmable controller software as valid command signals and resulted in an unplanned longwall shield movement.	
Are safety-critical areas of the software identified? *Example:* A software revision to the Shearer Initiated Roof Support Advance Mode software had inadvertently deleted the safety routine that prevented the processing of erroneous shearer location data. The shield advance system was originally programed to reject shearer location information outside of certain predefined parameters. The shearer gave erroneous location information to the shield advance system due to a hardware fault, this resulted in an unplanned longwall shield movement.	
Is access to the safety-critical sections of the software limited to authorized and competent people? *Example:* A programmable logic controller used to control power to equipment in a preparation plant was incorrectly reprogrammed such that a safety feature was bypassed that was originally present to prevent unintentional startup when power was restored after an outage. A mechanic was fatally injured while working in the blade mill area where power was not locked out and a circuit breaker controlling power to the mill was inadvertently reset.	
Are there adequate procedures for the control of software versions?	
Does the software contain only those features and capabilities required by the system? The software should not contain "undocumented" features.	

Does the system design prevent unauthorized or inadvertent access or modification to the software?	
Are feedback loops designed such that software runaway conditions do not occur due to feedback sensor or limit switch failures?	
Does the software detect improper operator inputs or sequences? If so, are alerts used to indicate the error and corrective actions? *Example:* An operator of a longwall shield had initiated an automatic override function, which should have locked out manually entered commands; however, the operator then entered a manual command. As a result, both commands were executed resulting in an unplanned movement.	
Is the software's response to out-of-range values specified for every input?	
Is the software's response to not receiving an expected input specified (i.e., are timeouts provided)?	
Does the software specify the length of the timeout, when to start counting the timeout, and the latency of the timeout (the point past which the receipt of new inputs cannot change the output result, even if they arrive before the actual output)?	
On a given input, will the software always follow the same path through the code (i.e., is the software's behavior deterministic)?	
Is each input bounded in time, i.e., does the specification include the earliest time at which the input will be accepted and the latest time at which the data will be considered valid (avoiding control decisions based on obsolete data)?	
Can a one-bit change in a register or variable, or a change to all 0's or all 1's, result in an unsafe decision outcome?	
Do critical data communicated between CPUs successfully pass data transmission verification checks in both CPUs?	
Do memory locations have unique references (e.g., one and only one name) to prevent memory usage conflicts between global/local variables?	
Do software-controlled sequences affecting safety require a minimum of two independent procedures for initiation?	
Does the software terminate to a known and predictably safe state?	
Are unused memory locations initialized to a pattern that, if executed as an instruction, will cause the system to revert to a known safe state?	
Are critical operational software instructions resident in nonvolatile read-only memory (ROM)?	
If a safety kernel is used, does it reside in ROM?	
If multiple, identical subsystems are used, is a mechanism in place to verify the proper software version for each subsystem?	

REFERENCES

Heimdahl MPE, Leveson NG [1996]. Completeness and consistency in hierarchical state-based requirements. IEEE Trans Software Eng *22*(6):363-377.

IEC [1998a]. Functional safety of electrical/electronic/programmable electronic safety-related systems. Geneva, Switzerland: International Electrotechnical Commission, Draft IEC65108-1 1: General requirements, version 4, May 12, 1998.

IEC [1998b]. Functional safety of electrical/electronic/programmable electronic safety-related systems. Geneva, Switzerland: International Electrotechnical Commission, Draft IEC65108-2, Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems, version 4, May 12, 1998.

IEC [1998c]. Functional safety of electrical/electronic/programmable electronic safety-related systems. Geneva, Switzerland: International Electrotechnical Commission, Draft IEC65108-3, Part 3: Software requirements, version 4, May 12, 1998.

IEC [1998d]. Functional safety of electrical/electronic/programmable electronic safety-related systems. Geneva, Switzerland: International Electrotechnical Commission, Draft IEC65108-4, Part 4: Definitions and abbreviations, version 4, May 12, 1998.

IEC [1998e]. Functional safety of electrical/electronic/programmable electronic safety-related systems. Geneva, Switzerland: International Electrotechnical Commission, Draft IEC65108-5, Part 5: Examples of methods for determination of safety integrity levels, version 4, May 12, 1998.

IEC [1998f]. Functional safety of electrical/electronic/programmable electronic safety-related systems. Geneva, Switzerland: International Electrotechnical Commission, Draft IEC65108-6, Part 6: Guidelines on the application of parts 2 and 3, version 4, May 12, 1998.

IEC [1998g]. Functional safety of electrical/electronic/programmable electronic safety-related systems. Geneva, Switzerland: International Electrotechnical Commission, Draft IEC65108-7 Part 7: Overview of techniques and measures, version 4, May 12, 1998.

Jacky J [1995]. Specifying a safety-critical control system in Z. IEEE Trans Software Eng *21*(2): 99-106.

Jaffe MS, Leveson NG, Heimdahl MPE, Melhart BE [1991]. Software requirements analysis for real-time process-control systems. IEEE Trans Software Eng *17*(3):241-258.

Knight JC [1993]. Development of a software safety process and a case study of its use. Annual report to the National Aeronautics and Space Administration, Langley Research Center, for SEAS proposal No. UVA/528344/CS93/103. Charlottesville, VA: University of Virginia, June.

Knight JC, Kienzle DM [1993]. Preliminary experience using Z to specify a safety-critical system. In: Bowen JP, Nicholls JE, eds. Proceedings of the Z User Workshop. New York, NY: Springer Verlag.

Leveson NG [1992]. High-pressure steam engines and computer software. In: Proceedings of the 14th International Conference on Software Engineering. Los Alamitos, CA: IEEE Computer Society Press, pp. 2-14

Leveson NG, Heimdahl MPE, Hildreth H, Reese JD [1994]. Requirements specification for process-control systems. IEEE Trans Software Eng *20*(9):684-707.

Miller EF Jr., Paige MR, Benson JP, Wisehart WR [1981]. Structural techniques of program validation. Tutorial: software testing and validation techniques. 2nd ed. Los Alamitos, CA: IEEE Computer Society Press, pp. 304-307.

Modugno F, Leveson NG, Reese JD, Partridge K, Sandy SD [1997]. Integrated safety analysis of requirements specifications. In: Proceedings of the Third IEEE International Symposium on Requirements Engineering. Los Alamitos, CA: IEEE Computer Society Press, pp. 148-159.

Myers GJ [1979]. The art of software testing. New York, NY: John Wiley & Sons.

Sammarco JJ, Fisher TJ [2001]. Programmable electronic mining systems: best practice recommendations (in nine parts). Part 2: 2.1 System safety. Pittsburgh, PA: U.S. Department of Health and Human Services, Public Health Service, Centers for Disease Control and Prevention, National Institute for Occupational Safety and Health, DHHS (NIOSH) Publication No. 2001-137, IC 9458.

Sammarco JJ, Kohler JL, Novak T, Morley LA [1997]. Safety issues and the use of software-controlled equipment in the mining industry. In: Proceedings of the IEEE-Industrial Applications Society 32nd Annual Meeting (October 5-9, 1997).

Sammarco JJ, Fisher TJ, Welsh JH, Pazuchanics MJ [1999]. Programmable electronics in mining: an introduction to safety. Pittsburgh, PA: NIOSH Special Workshop Report. Unpublished.

Scheper CO [1993]. Certification of reusable software components. Final report for contract F30602-92-C-0158. Rome, NY, U.S. Air Force, Rome Laboratory, March.

Scheper CO [1996]. A software code defect model for certification. Vol. 7, Final report for contract F30602-94-C-0024, Certification of Reusable Software Components. Rome, NY: U.S. Air Force, Rome Laboratory, June.

Stephans RA, Talso WW [1997]. System safety analysis handbook. 2nd ed. Unionville, VA: System Safety Society, Inc.

Trammell CJ, Poore JH, Linger RC, Prowell SJ [1999]. Cleanroom software engineering: technology and process: the SEI series. Reading, MA: Addison-Wesley.

APPENDIX A.—DESCRIPTION OF SELECTED SOFTWARE TESTING METHODS

Walkthroughs are a process of reading or otherwise visually reviewing a document or code, especially comparing it with other system descriptions. The document or code is reviewed with respect to technique, style, possible errors, violation of development standards, and other problems. Walkthroughs are characterized by the presence of the designer or programmer leading other members of the development team through the document or code while they ask questions and make comments. The process is guided by a checklist and governed by a set of detailed procedures.

Inspections are a review process similar to walkthroughs, except that the program author is not present. Work products are formally evaluated under the leadership of an independent moderator.

Formal Reviews are a review process similar to walkthroughs, with the exception that the customer is present**.**

Specifications/Requirements Reviews are walkthroughs or inspections conducted on a software requirements or specification document.

Design Reviews are walkthroughs or inspections conducted on a software design document to verify that requirements are correctly translated into design, that no additional functionality has been added, that interfaces are fully and correctly specified, and that design standards have been followed. A high-level design inspection validates the specification document against the requirements document. A low-level design inspection validates the design structure document against the requirements and design specification documents.

Code Reviews are walkthroughs or inspections conducted on software source code to verify that the design hierarchy (including interfaces) is correctly implemented and that coding standards have been followed. Checklists are used to identify common types of errors.

Code Reading is a process for code review conducted on software source code in which the code is read according to a systematic procedure with the goal of understanding how the code operates and then of determining if it is correct with respect to its required functionality. Step-wise abstraction is a technique used for code reading in which higher levels of abstraction are derived from prime (or nondecomposable) subprograms for the purpose of determining the actual functioning of the software and identifying errors or inconsistencies.

Structure Analysis is an evaluation of control and code structures and subprogram usage to detect program faults. It is especially directed toward identifying and verifying sequences of invocations of functions.

Error and Anomaly Detection is an analysis of the text of a program with respect to syntax, data, and physical units to detect program faults.

Dynamic methods are methods for evaluating software code by executing it. There are two types of dynamic methods: specification-based and implementation-based. Specification-based testing is also called black-box or functional testing; implementation-based testing is also called white-box or structural testing.

Specification-Based Testing tests that all requirements are met, tests that intended functions are performed fully and correctly, and exercises all paths through the requirements that may be specified for performing a required process. Software is treated as a black box, i.e., test cases are designed and input selected based on requirements and specifications without regard to the actual code, the software is executed using those inputs, and the resulting outputs are compared to the outputs that were expected based on the specifications and requirements. Complete testing would require that all possible values of all variables defined by the requirements and specification be tested. Since this is impossible, it is necessary to have a technique for selecting the best subset of all possible inputs. Frequently used selection techniques are equivalence partitions, boundary values, special values, and random testing.

Equivalence Partitions is a technique in which the input and output values are partitioned into at least two equivalence classes and input values are selected from each class. Any input value selected from a particular equivalence class should result in an output equivalent to that resulting from any other input value selected from that class. Both invalid and valid classes must be considered.

Boundary Values is a technique in which the set of inputs is selected from the boundary values of the equivalence classes that partition the complete input space defined by the specifications and requirements. The boundary values of an equivalence class are values outside the class that bound the values in the class. The minimum and maximum values in the class are also included in the boundary values.

Special Values is a technique in which the set of inputs is selected based on values that are likely to be associated with faults. Some specific types of special values are *distinct values* and *zero values*. Distinct values are assigned to input variables for some particular purpose, such as to ensure that the values of elements of an array differ and that particular indices are different in particular arrays. Zero values (zeroes) are assigned to variables in arithmetic expressions, and input variables are selected to cause internal variables to take on the value 0.

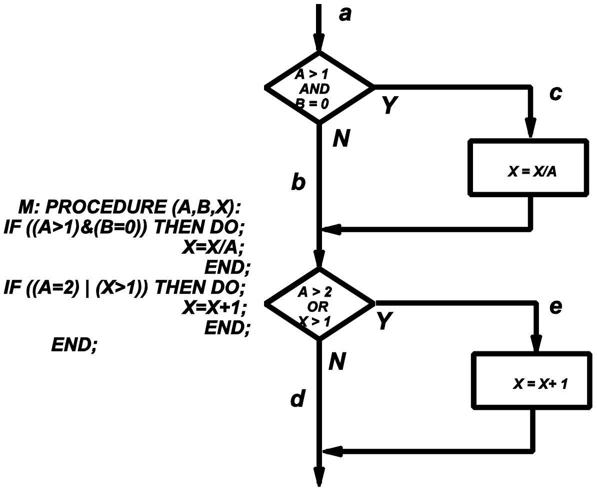
Random Testing is a technique in which the set of inputs are a randomly selected sample of all possible input values. The sample is selected according to the actual probability distribution of the input sequences.

Implementation-Based Testing verifies that the software design elements have been implemented correctly. It examines the internal structure of a program (i.e., the code) to determine its elementary components (e.g., data structures, statements, decisions, paths) and constructs test cases that will execute those elementary structures or combinations of those elementary structures. The degree to which the test cases exercise the elementary structures of the code is the measure of completeness (or adequacy) of structural testing and is referred to as "test coverage." The various testing techniques are categorized according to coverage of a particular type of structure. This form of testing is frequently called white-box testing to emphasize that it is distinguished from specification or black-box testing by its examination of the code. The specific structural techniques are differentiated by the type of structural element they address. The most commonly used techniques are statement, control flow, and data flow coverage. An additional structural technique, mutation testing, is used to measure the coverage of a set of test cases.

To illustrate differences in the various forms of structural testing, an example from Myers [1979] is shown in figure A-1. In this example we see that there are four statements; two decisions, each with two conditions and two outcomes; four possible paths through the code (a-b-d, a-b-e, a-c-d, and a-c-e); two computations; three data elements, one entry point, and one exit point.

Statement Coverage requires that every source code statement be executed at least once. This is the easiest, but weakest form of structural testing. Referring to figure A-1, we see that if a test case causes the Y branch to be taken at each decision point, then every statement will be executed. As indicated by Myers [1979], one test case (A ' 2, B ' 0, and X ' 3) would suffice to test all statements; however, only one path (a-c-e) would have been exercised and the correctness of the decision statements would not have been addressed.

Control Flow Coverage is a form of testing in which the elementary components to be executed are chosen according to the flow of control from statement to statement starting at the entry point and continuing to the exit point of the program. Thus, statements are selected for execution by each test case based on their relationship to the program's control flow. The number of control flow elements tested (versus the number of all control flow elements in the program) is the coverage metric associated with the test suite. Two control flow elements are commonly considered for control flow testing: decisions and paths. Decisions cause the control flow of a program to branch from one statement in a program to another, not necessarily its next linear successor.



```
 M: PROCEDURE (A,B,X):
IF ((A>1)&(B=0)) THEN DO;
                  X=X/A;
                    END;
IF ((A=2) | (X>1)) THEN DO;
                  X=X+1;
                    END;
      END;
```

Figure A-1.—Sample program code and control flow.

The goal of **Decision Coverage** testing is to determine that the decisions are correctly implemented and direct the control flow properly to result in the expected program output. For every decision, one or more conditions are evaluated, each of which has a set of possible outcomes and a set of possible outcomes of the decision, which result from the combination of the outcomes of its conditions. Branch coverage and multiple condition coverage are two of the several techniques that can be used to test decisions.

Branch Coverage attempts to invoke all outcomes of each decision and all entry points into the program. Branch coverage is also sometimes called decision coverage. Referring to figure A-1, branch coverage would require both the Y and N branches to be taken at each decision point. As indicated by Myers [1979], two test cases (A ' 3, B ' 0, X ' 3 and A ' 2, B ' 1, X ' 1) would provide 100% branch coverage; however, only two paths (a-c-d and a-b-e) would have been exercised and, just as in the statement coverage case, the correctness of the decision statements would not have been addressed.

Multiple Condition Coverage attempts to address the inadequacy of statement and branch coverage for testing the correctness of decision statements by ensuring that the conditions in the decision statements are completely exercised. It requires that all possible combinations of condition outcomes for each decision and all entry points into the program be invoked. This is sometimes extended to require that all combinations of input at each decision statement, and at any computation or assignment used as a flag for a decision, are executed. As detailed by Myers [1979] in the example in figure A-1, there are eight combinations of conditions to be covered by test cases: (1) A>1, B ' 0; (2) A>1, B…0; (3) A#1, B ' 0; (4) A#1, B…0; (5) A ' 2, X>1; (6) A ' 2, X#1; (7) A…2, X>1; and (8) A…2, X#1. These combinations can be covered by four test cases: (1) A ' 2, B ' 0, X ' 4; (2) A ' 2, B ' 1, X ' 1; (3) A ' 1, B ' 0, X ' 2; and (4) A ' 1, B ' 1, X ' 1; however, the a-c-d path would not have been exercised.

Thus, while decision coverage is more thorough than statement coverage, it still does not provide a high level of coverage of all the elementary components, in part because of its failure to exercise all combinations of branches.

Path Coverage testing attempts to address this area by identifying and exercising complete execution paths through the program. A path is a subset of program statements that are executed in sequence according to the program's control flow, where the first statement in the path is the first statement to be executed after the entry point into the program and the last statement in the path is the last statement to be executed before the exit point from the program. A major difficulty encountered in path testing is that even small programs can have very large numbers of unique paths, large enough, in fact, to make exhaustive path testing infeasible. Even if all paths could be tested, all defects would not necessarily be detected because path testing cannot detect missing paths or data-sensitivity errors [Myers 1979]. Two techniques for selecting the appropriate paths are Decision-to-Decision Path Testing and Linear Code Sequence and Jump (LCSAJ) Measures.

Decision-to-Decision Path testing exercises paths consisting of all program statements occurring between one decision statement and the next sequential decision statement and that are made executable by the selection of the outcome of the initial decision statement. These paths, also called

DD paths, are identified from a directed graph of the program consisting of nodes identifying program code statement segments and directed arcs identifying control transfers from one node to another. If each DD path is executed, then each statement is executed at least once and all possible outcomes of each decision are exercised [Miller et al. 1981].

Linear Code Sequence and Jump (LCSAJ) Measures is a technique that executes a sequence of consecutive code statements bounded by either the entry point into the code or a jump (branch) from one code location to another and by either the end of the program or the beginning of another jump. The code segments that form an LCSAJ are identified from the text of the program, not from a directed graph. LCSAJs are combined to form subpaths. A series of LCSAJ measures can be defined based on the number of LCSAJs exercised versus the total number of LCSAJs and the number *n* of LCSAJs constituting a subpath versus the total number of subpaths containing *n* or fewer LCSAJs.

Data Flow Coverage is a form of testing in which the elementary components to be executed are chosen according to the flow of data from statement to statement starting at the entry point and continuing to the exit point of the program. Thus, statements are selected for execution by each test case based on their relationship to the program's data elements, i.e, testing is based on the existence of a set of paths between definitions and assignments to data elements and references to and uses of those data elements. The goal of data flow coverage testing is to uncover data-sensitive errors and thus addresses the deficiencies in control flow testing in that area. There are several coverage criteria in this class, including the following: (1) all uses of a variable in any computation must be exercised (called all-c-uses), (2) all uses of a variable in a branch must be exercised and in such a way as to cause all outcomes of the branch to be exercised (called all-p-uses), (3) all-c-uses and all-p-uses (called all-uses), (4) every assignment to a variable must be exercised and every loopless path between the assignment and the use of the variable must be traversed (called all-du-paths), (5) each variable input to a subroutine is examined after the execution of the subroutine to demonstrate that there was no unexpected change in its value.

Mutation Testing is a structural technique used to measure the effectiveness of test cases. In mutation testing, the program is changed, or mutated, by inserting faults into the code. For example, an arithmetic or Boolean operator used in a computation or decision condition could be changed to another operator. The mutated program is executed using the test cases being evaluated, and it is observed whether the inserted faults are detected. If the fault is not detected, then the test cases are not providing adequate coverage of the program's elementary components.

Except for some of the inspection and review methods, all of the methods described above are used during the implementation and testing phases of software development. However, in safety applications, verification and validation must occur at the beginning of development with requirements specification and design. The methods discussed in the previous section for requirements specification and design also support verification and validation at those phases and provide traceability to the verification and validation activities of the implementation and testing phases.