Appendix J

Software Safety

J.0 Software Safety During Life Cycle Phases

The safety process should support a structured program life cycle model that incorporates both the system design and engineering process and the software acquisition process. Prominent software life cycle models include the waterfall and spiral methodologies. Although different models may carry different lifecycle emphasis, the adopted model should not affect the safety process itself. For discussion purposes only, this enclosure adopts a waterfall model (subject to IEEE/IEA Standard for Information Technology-software life cycle processes No. 12207.) For brevity, only the development phase of the Standard is addressed in terms of the relationship to software safety activities.

J.1 Safety Critical Software Development

A structured development environment and an organization using state-of-the-art methods are prerequisites to developing dependable safety critical software. The following requirements and guidelines are intended to carry out the cardinal safety rule and its corollary that no single event or action shall be allowed to initiate a potentially hazardous event. The system, upon detection of an unsafe condition or command, shall inhibit the potentially hazardous event sequence and originate procedures/functions to bring the system to a predetermined "safe" state.

The purpose of this section is to describe the software safety activities that should be incorporated into the software development phases of project development. The software safety information that should be included in the documents produced during these phases is also discussed. The term "software components" is used in a general sense to represent important software development products such as software requirements, software designs, software code or program sets, software tests, etc.

J.2 Software Concept and Initiation Phase

For most projects this lifecycle phase involves system level requirements and design development. Although most project work during this phase is concentrated on the subsystem level, software development has several tasks that must be initiated. These include the creation of important software documents and plans that will determine how, what, and when important software products will be produced or activities will be conducted. Each of the following documents should address software safety issues:

Document	**Software Safety Section**
System Safety Plan	Include software as a subsystem. Identify tasks.
Software Concepts Document	Identify safety critical processes.
Software Management Plan, and Software Configuration Management Plan	Coordination with systems safety tasks, flowdown incorporation of safety requirements. Applicability to safety critical software.
Software Security Plan	Security of safety critical software
Software Quality Assurance Plan	Support to software safety, verification of software safety requirements, safety participation in software reviews and inspections.

J.3 Software Requirements Phase

The cost of correcting software faults and errors escalates dramatically as the development life cycle progresses, making it important to correct errors and implement correct software requirements from the very beginning. Unfortunately, it is generally impossible to eliminate all errors. Software developers must therefore work toward two goals: developing complete and correct requirements, and correcting code to develop fault-tolerant designs that will detect and compensate for software faults. The second goal is required because the first is usually impossible to accomplish.

This section of the handbook describes the software safety team involvement in developing safety requirements for software. The software safety requirements can be top-down (flowed down from system requirements) and/or bottom-up (derived from hazard analyses). In some organizations, top-down flow is the only permitted route for requirements into software, and in those cases, newly derived bottom-up safety requirements must be flowed back into the system specification.

The requirements of software components are typically expressed as functions with corresponding inputs, processes, and outputs, plus additional requirements on interfaces, limits, ranges, precision, accuracy, and performance. There may also be requirements on the data of the program set - its attributes, relationships, and persistence, among others.

Software safety requirements are derived from the system and subsystem safety requirements developed to mitigate hazards identified in the Preliminary, System, and Subsystems Hazard Analyses.

Also, the assigned safety engineer flows requirements to systems engineering. The systems engineering group and the software development group have a responsibility to coordinate and negotiate requirement flowdown to be consistent with the software safety requirement flowdown.

The software safety organization should flow requirements into the Software Requirements Document (SRD) and the Software Interface Specification (SIS) or Interfaces Control Document (ICD). Safety-related requirements must be clearly identified in the SRD

SIS activities identify, define, and document interface requirements internal to the sub-system in which software resides, and between system (including hardware and operator interfaces), subsystem, and program set components and operation procedures. Note that the SIS is sometimes effectively contained in the SRD, or within an Interface Control Document (ICD) which defines all system interfaces, including hardware to hardware, hardware to software, and software to software.

J.3.1 Development of Software Safety Requirements

Software safety requirements are obtained from several sources, and are of two types: generic and specific. The generic category of software safety requirement is derived from sets of requirements that can be used in different programs and environments to solve common software safety problems. Examples of generic software safety requirements and their sources are given in **Section J.1.4.3** Generic Software Safety Requirements. Specific software safety requirements are system unique functional capabilities or constraints that are identified in three ways:

- Through top down analysis of system design requirements (from specifications): The system requirements may identify system hazards up-front, and specify which system functions are safety critical. The (software) safety organization participates or leads the mapping of these requirements to software.
- From the Preliminary Hazard Analysis (PHA): The PHA looks down into the system from the point of view of system hazards. Preliminary hazard causes are mapped to, or interact with, software. Software hazard control features are identified and specified as requirements.
- Through bottom up analysis of design data, (e.g. flow diagrams, FMEAs, fault trees etc.)

Design implementations allowed but not anticipated by the system requirements are analyzed and new hazard causes are identified. Software hazard controls are specified via requirements when the hazard causes map to, or interact with, software.

J.3.2 Safety Requirements Flowdown

Generic safety requirements are established "a priori" and placed into the system specification and/or overall project design specifications. From there they are flowed into lower level unit and module specifications. Other safety requirements, derived from bottom-up analysis, are flowed up from subsystems and components to the system level requirements. These new system level requirements are then flowed down across all affected subsystems. During the System Requirements Phase, subsystems and components may not be well defined. In this case, bottom-up analysis might not be possible until the Architectural Design Phase or even later.

An area of concern in the flowdown process is incomplete analysis, and/or inconsistent analysis of highly complex systems, or use of ad hoc techniques by biased or inexperienced analysts. The most rigorous (and most expensive) method of addressing this concern is adoption of formal methods for requirements analysis and flowdown. Less rigorous and less expensive ways include checklists and/or a standardized structured approach to software safety as discussed below and throughout this guidebook.

The following section contain a description of the type of analysis and gives the methodology by defining the task, the resources required to perform the analysis, and the expected output from the analyses.

Checklists and cross references

Tools and methods for requirement flow down analyses include checklists and cross-references. A checklist of required hazard controls and their corresponding safety requirements should be created and maintained. Then they can be used throughout the development life cycle to ensure proper flow down and mapping to design, code and test.

- Develop a systematic checklist of software safety requirements and any hazard controls, ensuring they correctly and completely include (and cross reference) the appropriate specifications, hazard analyses test and design documents. This should include both generic and specific safety requirements.
- Develop a hazard requirement flowdown matrix that maps safety requirements and hazard controls to system/software functions and to software modules and components. Where components are not yet defined, flow to the lowest level possible and tag for future flowdown.

Requirements Criticality Analysis

Criticality analysis identifies program requirements that have safety implications. A method of applying criticality analysis is to analyze the risks of the software/hardware system and identify those that could present catastrophic or critical risks. This approach evaluates each program requirement in terms of the safety objectives derived for the software component.

The evaluation will determine whether the requirement has safety implications and, if so, the requirement is designated "safety critical". It is then placed into a tracking system to ensure traceability of software requirements throughout the software development cycle from the highest-level specification all the way to the code and test documentation. All of the following techniques are focused on safety critical software components.

The system safety organization coordinates with the project system engineering organization to review and agree on the criticality designations. At this point the systems engineers may elect to make design changes to reduce the criticality levels or consolidate modules reducing the number of critical modules.

At this point, some bottom-up analyses can be performed. Bottom-up analyses identify requirements or design implementations that are inconsistent with, or not addressed by, system requirements. Bottom-up analyses can also reveal unexpected pathways (e.g., sneak circuits) for reaching hazardous or unsafe states. System requirements should be corrected when necessary.

It is possible that software components or subsystems might not be defined during the Requirements Phase, so those portions of the Criticality Analysis would be deferred to the Architectural Design Phase. In any case, the Criticality Analysis will be updated during the Architectural Design Phase to reflect the more detailed definition of software components.

The methodology for Requirements Criticality Analysis is:

- All software requirements are analyzed in order to identify additional potential system hazards that the system PHA did not reveal and to identify potential areas where system requirements were not correctly flowed to the software. Identified potential hazards are then addressed by adding or changing the system requirements and reflowing them to hardware, software and operations as appropriate.

- At the system level: identify hardware or software items that receive/pass/initiate critical signals or hazardous commands.
- At the software requirements level: identify software functions or objects that receive/pass/initiate critical signals or hazardous commands.
- This safety activity examines the system/software requirements and design to identify unsafe conditions for resolution such as out-of-sequence, wrong event, inappropriate magnitude, incorrect polarity, inadvertent command, adverse environment, deadlocking, and failure-to-command modes.
- The software safety requirements analysis considers such specific requirements as the characteristics discussed below as critical software characteristics.

The following resources are available for the Requirements Criticality Analysis. Note: documents in brackets correspond to terminology from DOD-STD-2167. Other document names correspond to NASA-STD-2100.91.

- Software Development Activities Plan [Software Development Plan] Software Assurance Plan [None], Software Configuration Management Plan [Same] and Risk Management Plan [Software Development Plan].
- System and Subsystem Requirements [System/Segment Specification (SSS), System/Segment Design Document].
- Requirements Document [Software Requirements Specifications].
- External Interface Requirements Document [Interface Requirements Specifications] and other interface documents.
- Functional Flow Diagrams and related data.

- Program structure documents.
- Storage and timing analyses and allocations.
- Background information relating to safety requirements associated with the contemplated testing, manufacturing, storage, repair, installation, use, and final disposition of the system.
- Information from the system PHA concerning system energy, toxic, and other hazardous event sources, especially ones that may be controlled directly or indirectly by software.
- Historical data such as lessons learned from other systems and problem reports.

Output products are the following:

- Updated Safety Requirements Checklist
- Definition of Safety Critical Requirements.

The results and findings of the Criticality Analyses should be fed to the System Requirements and System Safety Analyses. For all discrepancies identified, either the requirements should be changed because they are incomplete or incorrect, or else the design must be changed to meet the requirements. The analysis identifies additional hazards that the system analysis did not include, and identifies areas where system or interface requirements were not correctly assigned to the software.

The results of the criticality analysis may be used to develop Formal Inspection checklists for performing the formal inspection process described later in INSERT REFERENCE.

Critical Software Characteristics

Many characteristics are governed by requirements, but some may not be.

All characteristics of safety critical software must be evaluated to determine if they are safety critical. Safety critical characteristics should be controlled by requirements that receive rigorous quality control in conjunction with rigorous analysis and test. Often all characteristics of safety critical software are themselves safety critical.

Characteristics to be considered include at a minimum:

- Specific limit ranges
- Out of sequence event protection requirements (e.g., if-then statements)
- Timing
- Relationship logic for limits. Allowable limits for parameters might vary depending on operational mode or mission phase. Expected pressure in a tank varies with temperature, for example.
- Voting logic
- Hazardous command processing requirements (fault response
- Fault detection, isolation, and recovery

- Redundancy management/switchover logic; what to switch and under what circumstances, should be defined as methods to control hazard causes identified in the hazards analyses. For example, equipment that has lost control of a safety critical function should be switched to a good spare before the time to criticality has expired. Hot standby units (as opposed to cold standby) should be provided where a cold start time would exceed time to criticality.

This list is not exhaustive and often varies depending on the system architecture and environment.

J.3.3 Generic Software Safety Requirements

The generic category of software safety requirements are derived from sets of requirements and best practices used in different programs and environments to solve common software safety problems. Similar processors/platforms and/or software can suffer from similar or identical design problems. Generic software safety requirements capture these lessons learned and provide a valuable resource for developers.

Generic requirements prevent costly duplication of effort by taking advantage of existing proven techniques and lessons learned rather than reinventing techniques or repeating mistakes. Most development programs should be able to make use of some generic requirement; however, they should be used with care.

As technology evolves, or as new applications are implemented, new "generic" requirements will likely arise, and other sources of generic requirements might become available. A partial listing of generic requirement sources is shown below:

- EWRR (Eastern and Western Range Regulation) 127-1, Section 3.16.4 Safety Critical Computing System Software Design Requirements.
- AFISC SSH 1-1 System Safety Handbook - Software System Safety, Headquarters Air Force Inspection and Safety Center.
- EIA Bulletin SEB6-A System Safety Engineering in Software Development (Electrical Industries Association)
- Underwriters Laboratory - UL 1998 Standard for Safety - Safety-Related Software, January 4th, 1994

A listing of many of the generic software safety requirements is presented in the table below.

The failure of safety critical software functions shall be detected, isolated, and recovered from such that catastrophic and critical hazardous events are prevented from occurring.
Software shall perform automatic Failure Detection, Isolation, and Recovery (FDIR) for identified safety critical functions with a time to criticality under 24 hours
Automatic recovery actions taken shall be reported. There shall be no necessary response from ground operators to proceed with the recovery action.
The FDIR switchover software shall be resident on an available, non-failed control platform which is different from the one with the function being monitored.
Override commands shall require multiple operator actions.
Software shall process the necessary commands within the time to criticality of a hazardous event.
Hazardous commands shall only be issued by the controlling application, or by authorized ground personnel.
Software that executes hazardous commands shall notify ground personnel upon execution or provide the

reason for failure to execute a hazardous command.
Prerequisite conditions (e.g., correct mode, correct configuration, component availability, proper sequence, and parameters in range) for the safe execution of an identified hazardous command shall be met before execution.
In the event that prerequisite conditions have not been met, the software shall reject the command and alert the ground personnel.
Software shall make available status of all software controllable inhibits to the ground personnel.
Software shall accept and process ground personnel commands to activate/deactivate software controllable inhibits.
Software shall provide an independent and unique command to control each software controllable inhibit.
Software shall incorporate the capability to identify and status each software inhibits associated with hazardous commands.
Software shall make available current status on software inhibits associated with hazardous commands to the ground personnel.
All software inhibits associated with a hazardous command shall have a unique identifier.
Each software inhibit command associated with a hazardous command shall be consistently identified using the rules and legal values.
If an automated sequence is already running when a software inhibit associated with a hazardous command is activated, the sequence shall complete before the software inhibit is executed.
Software shall have the ability to resume control of an inhibited operation after deactivation of a software inhibit associated with a hazardous command.
The state of software inhibits shall remain unchanged after the execution of an override.
Software shall provide error handling to support safety critical functions.
Software shall provide caution and warning status to the ground personnel.
Software shall provide for ground personnel forced execution of any automatic safing, isolation, or switchover functions.
Software shall provide for ground personnel forced termination of any automatic safing, isolation, or switchover functions.
Software shall provide procession for ground personnel commands in return to the previous mode or configuration of any automatic safing, isolation, or switchover function.
Software shall provide for ground personnel forced override of any automatic safing, isolation, or switchover functions.
Software shall provide fault containment mechanisms to prevent error propagation across replaceable unit interfaces.
Software (including firmware) Power On Self Test (POST) utilized within any replaceable unit or component shall be confined to that single system process controlled by the replaceable unit or component.
Software (including firmware) POST utilized within any replaceable unit or component shall terminate in a safe state.
Software shall initialize, start, and restart replaceable units to a safe state.
For systems solely using software for hazard risk mitigation, software shall require two independent command messages for a commanded system action that could result in a critical or catastrophic hazard.
Software shall require two independent operator actions to initiate or terminate a system function that could result in a critical hazard.
Software shall require three independent operator actions to initiate or terminate a system function that could result in a catastrophic hazard.
Operational software functions shall allow only authorized access.
Software shall provide proper sequencing (including timing) of safety critical commands.
Software termination shall result in a safe system state.
In the event of hardware failure, software faults that lead to system failures, or when the software detects a configuration inconsistent with the current mode of operation, the software shall have the capability to place

the system into a safe state.
When the software is notified of or detects hardware failures, software faults that lead to system failures, or a configuration inconsistent with the current mode of operation, the software shall notify the crew, ground operators, or the controlling executive.
Hazardous processes and safing processes with a time to criticality such that timely human intervention may not be available, shall be automated (i.e., not require ground personnel intervention to begin or complete).
The software shall notify ground personnel during or immediately after execution of an automated hazardous or safing process.
Unused or undocumented codes shall be incapable of producing a critical or catastrophic hazard.
All safety critical elements (requirements, design elements, code modules, and interfaces) shall be identified as "safety critical."
An application software set shall ensure proper configuration of inhibits, interlocks, and safing logic, and exception limits at initialization.

Table J-1: Generic Software Safety Requirements listing

Coding Standards

Coding Standards, a class of generic software requirements, are, in practice, "safe" subsets of programming languages. These are needed because most compilers can be unpredictable in how they work. For example, dynamic memory allocation is predictable. In applications where some portions of memory are safety critical, it is important to control which memory elements are assigned in a particular compilation process; the defaults chosen by the compiler might be unsafe. Some attempts have been made at developing coding safety standards (safe subsets).

Timing, Sizing and Throughput Considerations

System design should properly consider real-world parameters and constraints, including human operator and control system response times, and flow these down to software. Adequate margins of capacity should be provided for all these critical resources. This section provides guidance for developers in specifying software requirements to meet the safety objectives. Subsequent analysis of software for Timing, Throughput and Sizing considerations is discussed elsewhere in this appendix.

Time to Criticality: Safety critical systems sometimes have a characteristic "time to criticality", which is the time interval between a fault occurring and the system reaching an unsafe state. This interval represents a time window in which automatic or manual recovery and/or safing actions can be performed, either by software, hardware, or by a human operator. The design of safing/recovery actions should fully consider the real-world conditions and the corresponding time to criticality. Automatic safing can only be a valid hazard control if there is ample margin between worst case (long) response time and worst case (short) time to criticality.

Automatic safing is often required if the time to criticality is shorter than the realistic human operator response time, or if there is no human in the loop. This can be performed by either hardware or software or a combination depending on the best system design to achieve safing.

Control system design can define timing requirements. Based on the established body of classical and modern dynamic control theory, such as dynamic control system design, and multivariable design in the s-domain (Laplace transforms) for analog continuous processes. Systems engineers are responsible for overall control system design. Computerized control systems use sampled data (versus continuous data). Sampled analog processes should make

use of Z-transforms to develop difference equations to implement the control laws. This will also make most efficient use of real-time computing resources.

Sampling rates should be selected with consideration for noise levels and expected variations of control system and physical parameters. For measuring signals that are not critical, the sample rate should be at least twice the maximum expected signal frequency to avoid aliasing. For critical signals, and parameters used for closed loop control, it is generally accepted that the sampling rate must be much higher; at least a factor of ten above the system characteristic frequency is customary.

Dynamic memory allocation: ensure adequate resources are available to accommodate usage of dynamic memory allocation, without conflicts. Identify and protect critical memory blocks. Poor memory management has been a leading factor in several critical failures.

Memory Checking: Self-test of memory usage can be as part of BIT/self-test to give advance warning of imminent saturation of memory.

Quantization: Digitized systems should select word-lengths long enough to reduce the effects of quantization noise to ensure stability of the system. Selection of word-lengths and floating-point coefficients should be appropriate with regard to the parameters being processed in the context of the overall control system. Too short word-lengths can result in system instability and misleading readouts. Too long word-lengths result in excessively complex software and heavy demand on CPU resources, scheduling and timing conflicts etc.

Computational Delay: Computers take a finite time to read data and to calculate and output results, so some control parameters will always be out of date. Controls systems must accommodate this. Also, check timing clock reference datum, synchronization and accuracy (jitter). Analyze task scheduling (e.g., with Rate Monotonic Analysis (RMA)).

J.4 Structured Design Phase Techniques

Structured design techniques greatly reduce the number of errors, especially requirements errors which are the most expensive to correct and may have the most impact on the overall safety of a system. These Structured Analysis and Design methods for software have been evolving over the years, each with its approach to modeling the needed world-view into software. The most recent analysis/design methods are Object Oriented Analysis & Design (OOA & OOD) and Formal Methods. To date, the most popular analysis methods have been Functional Decomposition, Data Flow (or Structured Analysis), and Information Modeling. OOA actually incorporates some of the techniques of all of these within its method, at lower levels, once the system is cast into objects with attributes and services. In the discussion of Structured Analysis, “analysis“ is considered as a process for evaluating a problem space (a concept or proposed system) and rendering it into requirements that reflect the needs of the customer. Functional Decomposition has been, and still is, a popular method for representing a system. Functional Decomposition focuses on what functions, and sub-functions, the system needs to perform and the interfaces between those functions. The general complaints with this method are 1) the functional capability is what most often changes during the design life cycle and is thus very volatile, and 2) it is often hard to see the connection between the proposed system as a whole and the functions determined to create that system. A detailed discussion of Structured Analysis and Formal Methods appears in Appendix D of this handbook.

J.4.1 Architectural Design Analysis

The software architectural design process develops the high level design that will implement the software requirements. All software safety requirements developed above are incorporated into the high-level software design as part of this process. The design process includes identification of safety design features and methods (e.g., inhibits, traps, interlocks and assertions) that will be used throughout the software to

implement the software safety requirements. After allocation of the software safety requirements to the software design, Safety Critical Computer Software Components (SCCSCs) are identified. Bottom-up safety analysis is performed on the architectural design to identify potential hazards, to define and analyze SCCSCs and the early test plans are reviewed to verify incorporation of safety related testing. Analyses included in the Architectural Design Phase are:

- Update Criticality Analysis
- Conduct Hazard Risk Assessment
- Analyze Architectural Design
- Interdependence Analysis
- Independence Analysis
- Update Timing/Sizing Analysis

J.4.2 Update Criticality Analysis

The software functions begin to be allocated to modules and components at this stage of development. Thus the criticality assigned during the requirements phase now needs to also be allocated to the appropriate modules and components.

Software for a system, while often subjected to a single development program, actually consists of a set of multi-purpose, multifunction entities. The software functions need to be subdivided into many modules and further broken down to components.

Some of these modules will be safety critical, and some will not. The criticality analysis provides the appropriate initial criticality designation for each software function. The safety activity relates identified hazards from the following analyses previously described to the Computer Software Components (CSCs) that may affect or control the hazards.

This analysis identifies all those software components that implement software safety requirements or components that interface with SCCSCs that can affect their output. The designation *Safety Critical Computer Software Component* (SCCSC) should be applied to any module, component, subroutine or other software entity identified by this analysis.

J.4.3 Conduct Risk Assessment

The safety activity performs a system risk assessment to identify and prioritize those SCCSCs that warrant further analysis beyond the architectural design level. System risk assessment of hazards as described in the NHB 1700 series of documents, consists of ranking risks by severity level versus probability of occurrence. This high-severity/high probability risks are prioritized higher for analysis and corrective action than low-severity/low probability risks.

While Requirements Criticality and Update Criticality analysis simply assign a Yes or No to whether each component is safety critical, the Risk Assessment process takes this further. Each SCCSCs is prioritized for analysis and corrective action according to the five levels of Hazard Prioritization ranking given previously.

J.4.4 Analyze Architectural Design

The safety activity analyzes the Architectural Design of those SCCSCs identified in the preceding paragraphs to ensure all safety requirements are specified correctly and completely in the Architectural Design. In addition, the safety activity determines where in the Architectural Design, and under what conditions

unacceptable hazards occur. This is done by postulating credible faults/failures and evaluating their effects on the system. Input/output timing, multiple event, out-of-sequence event, failure of event, wrong event, inappropriate magnitude, incorrect polarity, adverse environment, deadlocking, and hardware failure sensitivities are included in the analysis.

Methods used for FMEA (Failure Modes and Effects Analysis) can be used substituting software components for hardware components in each case. A widely used FMEA procedure is MIL-STD-1629, which is based on the following eight steps. Formal Inspections (described earlier), design reviews and animation/simulation augment this process.

1. Define the system to be analyzed
2. Construct functional block diagrams
3. Identify all potential item and interface failure modes
4. Evaluate each failure mode in terms of the worst potential consequences
5. Identify failure detection methods and compensating provisions
6. Identify corrective design or other actions to eliminate / control failure
7. Identify impacts of the corrective change
8. Document the analysis and summarize the problems which could not be corrected

Design Reviews

Design data is reviewed to ensure it properly reflects applicable software safety requirements. Design changes are generated where necessary. Applicability matrices, compliance matrices, and compliance checklists can be used to assist in completing this task. Output products are engineering change requests, hazard reports (to capture design decisions affecting hazard controls and verification) and action items.

Animation/Simulation

Simulators, prototypes (or other dynamic representations of the required functionality as specified by the design), and test cases to exercise crucial functions can be developed. Run the tests and observe the system response. Requirements can be modified as appropriate. Documented test results can confirm expected behavior or reveal unexpected behavior. The status of critical verifications is captured by hazard reports.

J.4.5 Interface Analysis

Interdependence Analysis

Examine the software to determine the interdependence among CSCs, modules, tables, variables, etc. Elements of software that directly or indirectly influences SCCSCs are also identified as SCCSCs, and as such should be analyzed for their undesired effects. For example, shared memory blocks used by two or more SCCSCs. The inputs and outputs of each SCCSC are inspected and traced to their origin and destination.

Independence Analysis

The safety activity evaluates available design documentation to determine the independence/dependence and interdependence of SCCSCs to both safety-critical and non-safety-critical CSCs. Those CSCs that are found to affect the output SCCSCs are designated as SCCSCs. Areas where FCR (Fault Containment Region) integrity is compromised are identified. The methodology is to map the safety critical functions to the software modules and map the software modules to the hardware hosts and FCRs. Each input and output of each SCCSC should be inspected. Resources are definition of safety critical functions needing to independent

design descriptions, and data diagrams. Design changes to achieve valid FCRs and corrections to SCCSC designations may be necessary

J.5 Detailed Design Analysis

During the Detailed Design phase, more detailed software artifacts are available, permitting rigorous analyses to be performed. Detailed Design Analyses can make use of artifacts such as detailed design specifications, emulators and Pseudo-Code Program Description Language products (PDL). Preliminary code produced by code generators within case tools should be evaluated. Many techniques to be used on the final code can be "dry run" on these design products. In fact, it is recommended that all analyses planned on the final code should undergo their first iteration on the code-like products of the detailed design. This will catch many errors before they reach the final code where they are more expensive to correct. The following techniques can be used during this design phase. Description of each technique follows the list.

- J.5.1 Design Logic Analysis
- J.5.2 Design Data Analysis
- J.5.3 Design Interface Analysis
- J.5.4 Design Constraint Analysis
- J.5.6 Software Fault Tree Analysis (SFTA)
- J.5.7 Petri-Nets
- J.5.8 Dynamic Flowgraph Analysis
- J.5.9 Measurement of Complexity
- J.5.10 Safe Subsets of Programming languages
- J.5.11 Formal Methods and Safety-Critical Considerations
- J.5.12 Requirements State Machines

J.5.1 Design Logic Analysis (DLA)

Design Logic Analysis (DLA) evaluates the equations, algorithms, and control logic of the software design. Logic analysis examines the safety-critical areas of a software component. A technique for identifying safety-critical areas is to examine each function performed by the software component. If it responds to, or has the potential to violate one of the safety requirements, it should be considered critical and undergo logic analysis. A technique for performing logic analysis is to analyze design descriptions and logic flows and note discrepancies.

The ultimate, fully rigorous DLA uses the application of Formal Methods (FM). Where FM is inappropriate, because of its high cost versus software of low cost or low criticality, simpler DLA can be used. Less formal DLA involves a human inspector reviewing a relatively small quantity of critical software artifacts (e.g. PDL, prototype code), and manually tracing the logic. Safety critical logic to be inspected can include failure detection/diagnosis; redundancy management, variable alarm limits, and command inhibit logical preconditions.

Commercial automatic software source analyzers can be used to augment this activity, but should not be relied upon absolutely since they may suffer from deficiencies and errors, a common concern of COTS tools and COTS in general.

J.5.2 Design Data Analysis

Design data analysis evaluates the description and intended use of each data item in the software design. Data analysis ensures that the structure and intended use of data will not violate a safety requirement. A technique used in performing design data analysis is to compare description-to-use of each data item in the design logic.

Interrupts and their effect on data must receive special attention in safety-critical areas. Analysis should verify that interrupts and interrupt handling routines do not alter critical data items used by other routines.

The integrity of each data item should be evaluated with respect to its environment and host. Shared memory, and dynamic memory allocation can affect data integrity. Data items should also be protected from being overwritten by unauthorized applications. Considerations of EMI affecting memory should be reviewed in conjunction with system safety.

J.5.3 Design Interface Analysis

Design interface analysis verifies the proper design of a software component's interfaces with other components of the system. This analysis will verify that the software component's interfaces have been properly designed. Design interface analysis verifies that control and data linkages between interfacing components have been properly designed. Interface requirements specifications are the sources against which the interfaces are evaluated.

Interface characteristics to be addressed should include data encoding, error checking and synchronization. The analysis should consider the validity and effectiveness of checksums and CRCs. The sophistication of error checking implemented should be appropriate for the predicted bit error rate of the interface. An overall system error rate should be defined, and budgeted to each interface. Examples of interface problems:

- Sender sends eight-bit word with bit 7 as parity, but recipient believes bit 0 is parity.
- Sender transmits updates at 10 Hz, but receiver only updates at 1 Hz.
- Sender encodes word with leading bit start, but receiver decodes with trailing bit start.
- Interface deadlock prevents data transfer (e.g., Receiver ignores or cannot recognize “ready to send”).
- User reads data from wrong address.
- Sender addresses data to wrong address.

In a language such as C, or C++ where data typing is not strict, sender may use different data types than reviewer expects. (Where there is strong data typing, the compilers will catch this).

J.5.4 Design Constraint Analysis

Design constraint analysis evaluates restrictions imposed by requirements, the real world and environmental limitations, as well as by the design solution. The design materials should describe all known or anticipated restrictions on a software component. These restrictions may include those listed below. Design constraint analysis evaluates the ability of the software to operate within these constraints.

- Update timing and sizing constraints
- Equations and algorithms limitations.
- Input and output data limitations (e.g., Range, resolution, accuracy).
- Design solution limitations.

- Sensor/actuator accuracy and calibration.
- Noise, EMI.
- Digital word-length (quantization/roundoff noise/errors).
- Actuator power / energy capability (motors, heaters, pumps, mechanisms, rockets, valves, etc.)
- Capability of energy storage devices (e.g., Batteries, propellant supplies).
- Human factors, human capabilities and limitations.
- Physical time constraints and response times.
- Off nominal environments (fail safe response).
- Friction, inertia, backlash in mechanical systems.
- Validity of models and control laws versus actual system behavior.
- Accommodations for changes of system behavior over time: wear-in, hardware wear-out, end of life performance versus beginning of life performance degraded system behavior and performance.

J.5.5 Rate Monotonic Analysis

Rate Monotonic Analysis is a useful analysis technique for software. It ensures that time critical activities will be properly verified.

J.5.6 Software Fault Tree Analysis (SFTA)

It is possible for a system to meet requirements for a correct state and to also be unsafe. It is unlikely that developers will be able to identify, prior to the fielding of the system, all correct but unsafe states which could occur within a complex system. In systems where the cost of failure is high, special techniques or tools such as Fault Tree Analysis (FTA) need to be used to ensure safe operation. FTA can provide insight into identifying unsafe states when developing safety critical systems. Fault trees have advantages over standard verification procedures. Fault trees provide the focus needed to give priority to catastrophic events, and they assist in determining environmental conditions under which a correct or incorrect state becomes unsafe.

J.5.7 Petri-Nets

Petri-nets are a graphical technique that can be used to model and analyze safety-critical systems for such properties as reachability, recoverability, deadlock, and fault tolerance. Petri-nets allow the identification of the relationships between system components such as hardware and software, and human interaction or effects on both hardware and software. Real-time Petri-net techniques can also allow analysts to build dynamic models that incorporate timing information. In so doing, the sequencing and scheduling of system actions can be monitored and checked for states that could lead to unsafe conditions.

The Petri-net modeling tool is different from most other analysis methods in that it clearly demonstrates the dynamic progression of state transitions. Petri-nets can also be translated into mathematical logic expressions that can be analyzed by automated tools. Information can be extracted and reformed into analysis assisting graphs and tables that are relatively easy to understand (e.g., reachability graphs, inverse Petri-net graphs, critical state graphs). Some of the potential advantages of Petri-nets over other safety analysis techniques include the following:

- Petri-nets can be used to derive timing requirements in real-time systems.
- Petri-nets allow the user to describe the system using graphical notation, and thus they free the analyst from the mathematical rigor required for complex systems.
- They can be applied through all phases of system development. Early use of Petri-nets can detect potential problems resulting in changes at the early stages of development where such changes are relatively easy and less costly than at later stages.
- They can be applied for the determination of worst case analysis and the potential risks of timing failures.
- A system approach is possible with Petri-nets since hardware, software and human behavior can be modeled using the same language.
- Petri-nets can be used at various levels of abstraction.
- Petri-nets provide a modeling language which can be used for both formal analysis and simulation.

Adding time and probabilities to each Petri-net allows incorporation of timing and probabilistic information into the analysis. The model may be used to analyze the system for other features besides safety.

Unfortunately, Petri-nets require a large amount of detailed analysis to build even relatively small systems, thus making them very expensive. In order to reduce expenses, a few alternative Petri-net modeling techniques have been proposed, each tailored to perform a specific type of safety analysis. For example, time Petri-net (TPN), take account for time dependency factor of real-time systems; inverse Petri-net, specifically needed to perform safety analysis, uses the previously discussed backward modeling approach to avoid modeling all of the possible reachable status; and critical state inverse Petri-nets, which further refine inverse Petri-net analysis by only modeling reachable states at predefined criticality levels.

Petri-net analysis can be performed at any phase of the software development cycle; though, it is highly recommended for reasons of expense and complexity that the process be started at the beginning of the development cycle and expanded for each of the succeeding phases. Petri-net, inverse Petri-net and critical state Petri-nets are all relatively new technologies, are costly to implement, and absolutely require technical expertise on the part of the analyst. Petri net analysis is a complex subject, and is treated in more detail in Appendix C of this handbook.

J.5.8 Dynamic Flowgraph Analysis

Dynamic Flowgraph Analysis is a new technique, not yet widely used and still in the experimental phase of evaluation. It does appear to offer some promise, and in many respects combines the benefits of conventional J.5.6 Software Fault Tree Analysis (SFTA) and J.5.7 Petri-Nets .

The Dynamic Flowgraph Methodology (DFM) is an integrated, methodical approach to modeling and analyzing the behavior of software-driven embedded systems for the purpose of dependability assessment and verification. The methodology has two fundamental goals: 1) to identify how events can occur in a system; and 2) identify an appropriate testing strategy based on an analysis of system functional behavior. To achieve these goals, the methodology employs a modeling framework in which models expressing the logic of the system being analyzed are developed in terms of contributing relationships between physical variables and temporal characteristics of the execution of software modules.

Models are analyzed to determine how a certain state (desirable or undesirable) can be reached. This is done by developing timed fault trees which take the form of logical combinations of static trees relating the system parameters at different points in time. The resulting information concerning the hardware and software states that can lead to certain events of interest can then be used to increase confidence in the system, eliminate unsafe execution paths, and identify testing criteria for safety critical software functions.

J.5.9 Measurement of Complexity

Software's complexity should be evaluated in order to determine if the level of complexity may contribute to areas of concern for workability, understandability, reliability and maintainability. Highly complex data and command structures are difficult, if not impossible, to test thoroughly and can lead to errors in logic either in the initial build or in subsequent updates. Not all paths can usually be thought out or tested for and this leaves the potential for the software to perform in an unexpected manner. Highly complex data and command structures may be necessary, however, there usually are techniques for avoiding too high a level of programming interweaving.

Linguistic, structural, and combined metrics exist for measuring the complexity of software and while discussed below briefly.

Use complexity estimation techniques, such as McCabe or Halstead. If an automated tool is available, the software design and/or code can be run through the tool. If there is no automated tool available, examine the critical areas of the detailed design and any preliminary code for areas of deep nesting, large numbers of parameters to be passed, intense and numerous communication paths, etc. (Refer to references cited above.) Resources are the detailed design, high level language description, source code, and automated complexity measurement tool(s).

Output products are complexity metrics, predicted error estimates, and areas of high complexity identified for further analysis or consideration for simplification.

Several automated tools are available on the market which provides these metrics. The level and type of complexity can indicate areas where further analysis, or testing, may be warranted. Beware, however, these metrics should be used with caution as they may indicate that a structure, such as a CASE statement, is highly complex while in reality that complexity leads to a simpler, more straight forward method of programming and maintenance, thus decreasing the risk of errors.

Linguistic measurements measure some property of the text without regard for the contents (e.g., lines of code, number of statements, number and type of operators, total number and type of tokens, etc). Halstead's Metrics is a well known measure of several of these arguments.

Structural metrics focuses on control-flow and data-flow within the software and can usually be mapped into a graphics representation. Structural relationships such as the number of links and/or calls, number of nodes, nesting depth, etc. are examined to get a measure of complexity. McCabe's Cyclomatic Complexity metric is the most well known and used metric for this type of complexity evaluation.

J.5.10 Safe Subsets of Programming languages

Safety specific coding standards are developed which identify requirements for annotation of safety-critical code and limitation on use of certain language features which can reduce software safety. The purpose of this section is to provide a technical overview of safety-critical coding practices for developers and safety engineers, primarily those involving restricting the use of certain programming language constructs.

The use of software to control safety-critical processes is placing software development environments (i.e. languages, compilers, utilities, etc.) under increased scrutiny. When computer languages are taught, students

are seldom warned of the limitations and insecurities that the environment possesses. An insecurity is a feature of a programming language whose implementation makes it impossible or extremely difficult to detect some violation of the language rules, by mechanical analysis of a program's text. The computer science profession has only recently focused on the issues of the inherent reliability of programming environments for safety-critical applications.

This section will provide an introduction on the criteria for determining which languages are well suited for safety-critical applications. In addition, an overview of a safe subset of the ADA language will be discussed with the rationale for rejecting language constructs. Reading knowledge of Pascal, ADA, C or another modern high level block structured language is required to understand the concepts that are being discussed.

There are two primary reasons for restricting a language definition to a subset: 1) some features are defined in an ambiguous manner and 2) some features are excessively complex. A language is considered suitable for use in a safety-critical application if it has a precise definition (complete functionality as well), is logically coherent, and has a manageable size and complexity. The issue of excessive complexity makes it virtually impossible to verify certain language features. Overall, the issues of logical soundness and complexity will be the key toward understanding why a language is restricted to a subset for safety-critical applications.

An overview of the insecurities in the ADA language standard is included in this entry. Only those issues that are due to the ambiguity of the standard will be surveyed. The problems that arise because a specific implementation (e.g., a compiler) is incorrect can be tracked by asking the compiler vendor for a historical list of known bugs and defect repair times. This information should give a user a basis with which to compare the quality of product and service of different vendors.

Insecurities Common to All Languages

All programming languages have insecurities either in their definition or their implementation. The evolutionary trend of computer languages shows a trend of newer languages trying to correct the shortfalls of older generation languages (even though some individuals complain about additional restrictions).

Probably the most common misuse in practically all-programming languages is that of uninitialized variables. This mistake is very hard to catch because unit testing will not flag it unless explicitly designed to do so. The typical manifestation of this error is when a program that has been working successfully is run under different environmental conditions and the results are not as expected.

Calls to de-allocate memory should be examined to make sure that not only is the pointer released but that the memory used by the structure is released.

The order of evaluation of operands when side effects from function calls modify the operands is generally dismissed as poor programming practice but in reality is an issue that is poorly defined (no standard of any type has been defined) and arbitrarily resolved by implementers of language compilers.

Method of Assessment

The technique used to compare programming languages will not deal with differences among manufacturers of the same language. Compiler vendor implementations, by and large, do not differ significantly from the intent of the standard, however standards are not unambiguous and they are interpreted conveniently for marketing purposes. One should be aware that implementations will not adhere 100% to the standard because of the extremely large number of states a compiler can produce. The focus of this study then is to review the definition of a few languages for certain characteristics that will provide for the user a shell against inadvertent misuse. When evaluating a language, the following questions should be asked of the language as a minimum:

- Can it be shown that the program cannot jump to an arbitrary location?
- Are there language features that prevent an arbitrary memory location from being overwritten?
- Are the semantics of the language defined sufficiently for static code analysis to be feasible?
- Is there a rigorous model of both integer and floating point arithmetic within the standard?
- Are there procedures for checking that the operational program obeys the model of the arithmetic when running on the target processor?
- Are the means of typing strong enough to prevent misuse of variables?
- Are there facilities in the language to guard against running out of memory at runtime?
- Does the language provide facilities for separate compilation of modules with type checking across module boundaries?
- Is the language well understood so designers and programmers can write safety-critical software?
- Is there a subset of the language which has the properties of a safe language as evidenced by the answers to the other questions?

J.5.11 Formal Methods and Safety-Critical Considerations

In the production of safety-critical systems or systems that require high assurance, Formal Methods* provide a methodology that gives the highest degree of assurance for a trustworthy software system. Assurance cannot be measured in a quantitative, objective manner for software systems that require reliability figures that are of the order of one failure in 10^9 hours of operation. An additional difficulty that software reliability cannot address to date, in a statistically significant manner, is the difference between catastrophic failures and other classes of failures.

Formal Methods have been used with success on both military and commercial systems that were considered safety-critical applications. The benefits from the application of the methodology accrue to both safety and non-safety areas. Formal Methods do not guarantee a precise quantifiable level of reliability; at present they are only acknowledged as producing systems that provide a high level of assurance.

On a qualitative level the following list identifies different levels of application of assurance methods in software development. They are ranked by the perceived level of assurance achieved with the lowest numbered approaches representing the highest level of assurance. Each of the approaches to software development is briefly explained by focusing on that part of the development that distinguishes it from the other methods.

Formal development down to object code requires that formal mathematical proofs be carried out on the executable code.

Formal development down to source code requires that the formal specification of the system undergo proofs of properties of the system.

Rigorous development down to source code is when requirements are written in a formal specification language and emulators of the requirements are written. The emulators serve the purpose of a prototype to test the code for correctness of functional behavior.

Structured development to requirements analysis then rigorous development down to source code performs all of the steps from the previous paragraph. The source code undergoes a verification process that resembles a proof but falls short of one.

Structured development down to source code is the application of the structured analysis/structured design method. It consists of a conceptual diagram that graphically illustrates functions, data structures, inputs, outputs, and mass storage and their interrelationships. Code is written based on the information in the diagram.

Ad hoc techniques encompass all of the non-structured and informal techniques (i.e. hacking, code a little then test a little).

J.5.12 Requirements State Machines

Requirements State Machines (RSM) are sometimes called Finite State Machines (FSM). An RSM is a model or depiction of a system or subsystem, showing states and the transitions between the states. Its goal is to identify and describe ALL possible states and their transitions. RSM analysis can be used on its own, or as a part of a structured design environment, e.g., object oriented design or formal methods.

Whether or not formal methods are used to develop a system, a high level RSM can be used to provide a view into the architecture of an implementation without being engulfed by all the accompanying detail. Semantic analysis criteria can be applied to this representation and to lower level models to verify the behavior of the RSM and determine that its behavior is acceptable. The analysis criteria will be listed in a section below and in subsequent sections because they are applicable at practically every stage of the development life cycle.

Characteristics of State Machines

A formal description of state machines can be obtained from texts on Automata Theory. This description will only touch on those properties that are necessary for a basic understanding of the notation and limitations. State machines use graph theory notation for their representation. A state machine consists of states and transitions. The state represents the condition of the machine and the transition represent changes between states. The transitions are directed (direction is indicated by an arrow), that is, they represent a directional flow from one state to another. A trigger or input that is labeled on the transition induces the transition from one state to another. Generally the state machine produces an output.

The state machine models should be built to abstract different levels of hierarchy. The models are partitioned in a manner that is based on considerations of size and logical cohesiveness. An uppermost level model should contain at most 15 to 20 states; this limit is based on the practical consideration of comprehensibility. In turn, each of the states from the original diagram can be exploded in a fashion similar to the bubbles in a data flow diagram/control flow diagram (DFD/CFD) (from a structured analysis/structured design methodology) to the level of detail required. An RSM model of one of the lower levels contains a significant amount of detail about the system.

The states in each diagram are numbered and classified as one of the following attributes: Passive, Startup, Safe, Unsafe, Shutdown, Stranded and Hazard. For the state machine to represent a viable system, the diagram must obey certain properties that will be explained later in this work.

The *passive* state represents an inert system, that is, nothing is being produced. However, in the passive state, input sensors are considered to be operational. Every diagram of a system contains at least one passive state. A passive state may transition to an unsafe state.

The *startup* state represents the initialization of the system. Before any output is produced, the system must have transitioned into the startup state where all internal variables are set to known values. A startup state must be proven to be safe before continuing work on the remaining states. If the initialization fails, a timeout may be specified and a state transition to an unsafe or passive state may be defined.

Properties of Safe State Machines

There are certain properties that the state machine representation should exhibit in order to provide some degree of assurance that the design obeys certain safety rules. The criteria for the safety assertions are based on logical considerations and take into account input/output variables, states, trigger predicates, output predicates, trigger to output relationship and transitions.

Input/Output Variables

All information from the sensors should be used somewhere in the RSM. If not, either an input from a sensor is not required or, more importantly, an omission has been made from the software requirements specification. For outputs it can be stated that, if there is a legal value for an output that is never produced, then a requirement for software behavior has been omitted.

State Attributes

The state attributes of the RSM are to be labeled according to the scheme in Chapter 10.

J.6 Code Analysis

Code analysis verifies that the coded program correctly implements the verified design and does not violate safety requirements. In addition, at this phase of the development effort, many unknown questions can be answered for the first time. For example, the number of lines of code, memory resources and CPU loads can be seen and measured, where previously they were only predicted, often with a low confidence level. Sometimes significant redesign is required based on the parameters of the actual code. Code permits real measurements of size, complexity and resource usage. Code Analyses include:

- Code Logic Analysis
- Software Fault Tree Analysis (SFTA)
- Petri-Nets
- Code Data Analysis
- Code Interface Analysis
- Measurement of Complexity
- Code Constraint Analysis
- Safe Subsets of Programming languages
- Formal Methods and Safety-Critical Considerations
- Requirements State Machines

Some of these code analysis techniques mirror those used in detailed design analysis. However, the results of the analysis techniques might be significantly different than during earlier development phases, because the final code may differ substantially from what was expected or predicted.

Each of these analyses, contained in this section, should be undergoing their second iteration, since they should have all been applied previously to the code-like products (PDL) of the detailed design.
There are some commercial tools available which perform one or more of these analyses in a single package. These tools can be evaluated for their validity in performing these tasks, such as logic analyzers, and path analyzers. However, unvalidated COTS tools, in themselves, cannot generally be considered valid methods for formal safety analysis. COTS tools are often useful to reveal previously unknown defects.

Note that the definitive formal code analysis is that performed on the final version of the code. A great deal of the code analysis is done on earlier versions of code, but a final check on the final version is essential. For safety purposes it is desirable that the final version have no "instrumentation" (i.e., extra code added), in order to see where erroneous jumps go. One may need to run the code on an instruction set emulator that can monitor the code from the outside, without adding the instrumentation.

J.6.1 Code Logic Analysis

Code logic analysis evaluates the sequence of operations represented by the coded program. Code logic analysis will detect logic errors in the coded software. Performing logic reconstruction, equation reconstruction and memory decoding conduct this analysis.

Logic reconstruction entails the preparation of flow charts from the code and comparing them to the design material descriptions and flow charts.

Equation reconstruction is accomplished by comparing the equations in the code to the ones provided with the design materials.

Memory decoding identifies critical instruction sequences even when they may be disguised as data. The analyst should determine whether each instruction is valid and if the conditions under which it can be executed are valid. Memory decoding should be done on the final un-instrumented code. Employment of Fault Trees and Petri Nets has been discussed in the previous section of this appendix.

J.6.2 Code Data Analysis

Code data analysis concentrates on data structure and usage in the coded software. Data analysis focuses on how data items are defined and organized. Ensuring that these data items are defined and used properly is the objective of code data analysis. This is accomplished by comparing the usage and value of all data items in the code with the descriptions provided in the design materials.

Of particular concern to safety is ensuring the integrity of safety critical data against being inadvertently altered or overwritten. For example, check to see if interrupt processing is interfering with safety critical data. Also, check the "typing" of safety critical declared variables.

J.6.3 Code Interface Analysis

Code interface analysis verifies the compatibility of internal and external interfaces of a software component. A software component is composed of a number of code segments working together to perform required tasks. These code segments must communicate with each other, with hardware, other software components, and human operators to accomplish their tasks. Check that parameters are properly passed across interfaces.

Each of these interfaces is a source of potential problems. Code interface analysis is intended to verify that the interfaces have been implemented properly. Hardware and human operator interfaces should be made part of the "Design Constraint Analysis" discussed below.

J.6.4 Measurement of Complexity

As a goal, software complexity should be minimized to reduce likelihood of errors. Complex software also is more likely to be unstable, or suffer from unpredictable behavior. Modularity is a useful technique to reduce complexity. Complexity can be measured via McCabe's metrics and similar techniques.

J.6.5 Update Design Constraint Analysis

The criteria for design constraint analysis applied to the detailed design can be updated using the final code. At the code phase, real testing can be performed to characterize the actual software behavior and performance in addition to analysis.

The physical limitations of the processing hardware platform should be addressed. Timing, sizing and throughput analyses should also be repeated as part of this process to ensure that computing resources and memory available are adequate for safety critical functions and processes.

Underflows/overflows in certain languages (e.g., ADA) give rise to "exceptions" or error messages generated by the software. These conditions should be eliminated by design if possible; if they cannot be precluded, then error handling routines in the application must provide appropriate responses, such as retry, restart, etc.

J.6.6 Code Inspection Checklists (including coding standards)

Coding standards are based on style guides and safe subsets of programming languages. Checklists should be developed during formal inspections to facilitate inspection of the code to demonstrate conformance to the coding standards.

Fagan Formal Inspections (FIs)

FIs are one of the best methodologies available to evaluate the quality of code modules and program sets. Many projects do not schedule any formal project-level software reviews during coding. When software is ready to be passed on to subsystems for integration, projects may elect to conduct an Integration Readiness Review when audit or inspection reports and problem reports may be evaluated. Other than these reports, the only formal documentation usually produced are the source code listings from configuration management.

J.6.7 Formal Methods

Generation of code is the ultimate output of Formal Methods. In a "pure" Formal Methods system, analysis of code is not required. In practice, however, attempts are often made to "apply" Formal Methods to existing code after the fact. In this case the analysis techniques of the previous sections (0 through 0) may be used to "extract" the logic of the code, and then compare the logic to the formal requirements expressions from the Formal Methods.

J.6.8 Unused Code Analysis

A common real world coding error is generation of code that is logically excluded from execution; that is, preconditions for the execution of this code will never be satisfied. Such code is undesirable for three reasons; a) it is potentially symptomatic of a major error in implementing the software design; b) it introduces unnecessary complexity and occupies memory or mass storage which is often a limited resource; and c) the unused code might contain routines which would be hazardous if they were inadvertently executed (e.g., by a hardware failure or by a Single Event Upset. SEU is a state transition caused by a high-speed subatomic particle passing through a semiconductor - common in nuclear or space environments).

There is no particular technique for identifying unused code; however, unused code is often identified during the course of performing other types of code analysis. Unused code can be found during unit testing with COTS coverage analyzer tools.

Care should be taken during logical code analyses to ensure that every part of the code is eventually exercised at some time during all possible operating modes of the system.

J.7 Test Phase

Two sets of analyses should be performed during the testing phase: analyses before the fact to ensure validity of tests, and analyses of the test results. Tests are devised to verify all safety requirements where testing has been selected as appropriate verification method. This is not considered here as analysis. Analysis before the fact should, as a minimum, consider test coverage for safety critical Must-Work-Functions.

J.7.1 Test Coverage

For small pieces of code it is sometimes possible to achieve 100% test coverage (i.e., to exercise every possible state and path of the code). However, it is often not possible to achieve 100 % test coverage due to the enormous number of permutations of states in a computer program execution, versus the time it would take to exercise all those possible states. Also there is often a large indeterminate number of environmental variables, too many to completely simulate.

Some analysis is advisable to assess the optimum test coverage as part of the test planning process. There is a body of theory that attempts to calculate the probability that a system with a certain failure probability will pass a given number of tests.

Techniques known as “white box” testing can be performed, usually at the modular level.
Statistical methods such as Monte Carlo simulations can be useful in planning "worst case" credible scenarios to be tested.

J.7.2 Test Results Analysis

Test results are analyzed to verify that all safety requirements have been satisfied. The analysis also verifies that all identified hazards have been eliminated or controlled to an acceptable level of risk. The results of the test safety analysis are provided to the ongoing system safety analysis activity. All test discrepancies of safety critical software should be evaluated and corrected in an appropriate manner.

J.7.3 Independent Verification and Validation

For high value systems with high-risk software, an IV&V organization is usually involved to oversee the software development. The IV&V organization should fully participate in the validation of test analysis.