# NASA Software Safety Guidebook

**National Aeronautics and Space Administration**
**I**

**Forward**

This document is a product of the NASA Software Program, an Agencywide program to promote the continual improvement of software engineering within NASA. The goals and strategies for this program are documented in the NASA Software Strategic Plan, July 13, 1995.

Additional information is available from the Software IV&V Facility on the world-wide-web site http://www.ivv.nasa.gov

## Contents

## Figures

# Tables

# 1. INTRODUCTION

This NASA Software Safety Guidebook was prepared by the NASA Glenn Research Center, Office of Safety Assurance Technologies, under a Center Software Initiative Proposal (CSIP) task for the National Aeronautics and Space Administration.

The NASA Software Safety Standard NASA-STD-8719.13A [1] prepared by NASA HQ addresses the “who, what, when and why” of Software Safety Analyses. This Software Safety Guidebook addresses the “how to”.

## *1.1 Scope*

The focus of this document is on analysis and development of safety critical software, including firmware (e.g. software residing in non-volatile memory, such as ROM, EPROM, or EEPROM) and programmable logic.

This Guidebook provides information on development activities and analyses used in the creation and assurance of safety critical software.  Resource data required for each task, methodologies and tools for performing the task, and the output products are detailed.  It also describes how to address software safety in the overall software development, management, and risk management activities.

This Guidebook goes on to describe techniques and procedures. Some techniques are well established and are illustrated in detail (or good reference sources are provided). Other techniques or analyses are new, and not much information is available. The Guidebook attempts to give a flavor of the technique or procedure as well as pointing to sources of more information.

To make the guidebook more practical, it contains analysis examples and possible pitfalls and problems that may be encountered during the analysis process. It is a synergistic collection of techniques either already in use throughout NASA and industry, or which have potential for use. Opinions differ widely concerning the validity of the various techniques, and this Guidebook attempts to present these opinions, without prejudging their validity. In most cases there are few or no "metrics" to quantitatively evaluate or compare the techniques. Moreover, this Guidebook is meant not only to provide possible techniques and analyses, but to open the reader to how to think about software from a safety perspective. It is important to observe software development with a "safety eye". This Guidebook points out things to look for (and look out for) in the development of safety critical software. Development approaches, safety analyses, and testing methodologies that lead to improved safety in the software product are included.

Numerous existing documents provide details on various analysis techniques. If a technique is well described elsewhere, references are provided. If a NASA standard or guideline exists which defines the format and/or content of a specific document, it is referenced and the user should follow the instructions of that document.

In addition to the existing techniques in the literature, some practical methods are presented which have been developed and used successfully at the system level for top-down software hazards analyses. Their approach is similar to NSTS 13830 Implementation Procedure for NASA Payload System Safety Requirements [2].

There are many different analysis techniques described in the open literature that are brought together, evaluated, and compared. This guidebook addresses the value added versus cost of each technique with respect to the overall software development and assurance goals.

The reader is expected to have some familiarity with the NASA methodologies for system safety analysis and/or software development. However, no experience with either is assumed or required. Readers completely unfamiliar with NASA methodologies for software development and system safety may have difficulty with some portions of this guidebook. Acronyms and definitions of terminology used in this guidebook are contained in Appendix-A.

## 1.2 Purpose

The purpose of this guidebook is to aid organizations involved in the development and assurance of safety critical software (i.e. software developers, software managers, software assurance, system safety and software safety organizations). It is meant to help both system safety people who are unfamiliar with software, and the software development organization which is unfamiliar with safety.

This guidebook focuses on software development and the tasks and analyses associated with it. Guidance on the acquisition of software, either commercial off-the-shelf or created under contract, is given in Section 7.

While the focus of this guidebook is on the development of software for **safety-critical** systems, much of the information and guidance is also appropriate to the creation of **mission critical** software.

## 1.3 Acknowledgments

Much of the material presented in this Guidebook has been based directly or indirectly on a variety of sources (NASA, government agencies, technical literature sources), and contains some original material previously undocumented. These sources are too numerous to list here, but are appropriately referenced throughout.

A special acknowledgment is owed to the NASA/Caltech Jet Propulsion Laboratory of Pasadena, California, whose draft "*Software Systems Safety Handbook*" [4] has been used verbatim or slightly modified in several sections of this Guidebook.

Our gratitude goes to the many NASA engineers and contractors who reviewed drafts of the guidebook.

We also thank the American Society of Safety Engineers for permission to reproduce portions of the paper Gowen, Lon D. and Collofello, James S. "Design Phase Considerations for Safety-Critical Software Systems". PROFESSIONAL SAFETY, April, 1995.

## 1.4 Associated Documents

Documents detailing software safety standards, software development standards, and guidebooks are listed in Section 8 References. Included are NASA Standards for software, as well as IEEE and military standards.

## 1.5 Roadmap of this Guidebook

This Guidebook provides information for two diverse groups: system safety and software development/management. It attempts to provide necessary software knowledge to the system safety engineer, specifically an increased understanding of the role that software plays in safety critical systems. In addition, software safety analysis techniques are described that may be performed by the safety engineer.

For the software developer (or software management), this guidebook provides information on development techniques and analyses that increase the safety of the software. More importantly, the software developer is presented with a new way of looking at the software - with a "safety eye".

Section 2 provides a preliminary look at the concepts of system safety, and how software relates to it. The section is written primarily for software developers, though system safety engineers will learn about various types of software that should be considered in a system safety context.

Section 3 gives a more in-depth look at software safety. In particular, guidance is provided on how to scope the safety effort and tailor the processes and analyses to the required level of effort. Details on the specific development processes and safety analyses are provided in the following sections.

Section 4 discusses development processes and techniques used by the software engineers (developers) while creating safety critical software. These techniques are described in detail, and are organized by the development lifecycle.

Section 5 details the safety analyses to be performed, organized by the software development lifecycle. This section is written primarily for the safety engineer, though the software developer should be aware of what the analyses entail.

Section 6 is a collection of specific problem areas that we felt should be addressed. Much of this section will be of interest to software developers. Safety engineers may wish to skim this section to obtain a better understanding of software.

Section 7 discusses the acquisition of software. Both COTS/GOTS (commercial and government off-the-shelf) software and software created under contract are considered.

Section 8 contains reference and resource information.

Appendix A provides definitions of commonly used terms and a list of acronyms.

Appendices B, C, and D provide details on three analysis techniques (Software Fault Tree Analysis, Software Failure Modes and Effects Analysis, and Requirements State Machine).

Appendix E contains a collection of checklists.

For those looking for particular tasks to perform at any specified time, the following chart is a roadmap that shows which tasks to perform for a given life cycle phase. For a more detailed breakdown of activities, see *Table 3-1 NASA Software Life-cycle - Reviews and Documents*.

| Life Cycle Phase | Tasks and Priorities | How To: Development Tasks | How To: Analysis Tasks |
|---|---|---|---|
| Project Management | Section 2 System Safety Program | Section 2.2 Which Software is Hazardous? | Section 2.3 Preliminary Hazard Analysis |
| Concept Initiation | Table 3-7 Software Requirements Phase | Section 4.1 Software Concept and Initiation Phase | Section 5.1 Software Safety Requirements Analysis |
| Software Requirements | Table 3-7 Software Requirements Phase | Section 4.2 Software Requirements Phase | Section 5.1 Software Safety Requirements Analysis |
| Software Architectural Design | Table 3-8 Software Architectural Design Phase | Section 4.3 Architectural Design Phase | Section 5.2 Architectural Design Analysis |
| Software Detailed Design | Table 3-9 Software Detailed Design Phase | Section 4.4 Detailed Design Phase | Section 5.3 Detailed Design Analysis |
| Software Implementation | Table 3-10 Software Implementation Phase | Section 4.5 Software Implementation | Section 5.4 Code Analysis |
| Software Test | Table 3-11 Software Testing Phase<br>Table 3-12 Dynamic Testing<br>Table 3-13 Software System Testing | Section 4.6 Software Integration and Test | Section 5.5 Test Analysis |

# 2. SOFTWARE SAFETY IN A SYSTEM SAFETY CONTEXT

Safety is a team effort and is everyone's responsibility. Software is a vital part of the system. Project managers, systems engineers, software lead and engineers, software assurance or QA, and system safety personnel all play a part in creating a safe system. The goal is for the software to contribute to the safety and functionality of the whole system.

But how do you know if any of your software is "unsafe"? What are the hazards that software may cause, or that software may control? Why should you even care about software safety?

When a device or system can lead to injury, death, or the loss of vital (and expensive) equipment, system safety is involved. Often hardware devices are used to mitigate the hazard potential, or to provide a "fail safe" mechanism should the worst happen. As software becomes a larger part of the electromechanical systems, hardware hazard controls are being replaced by software controls. Software can respond more quickly to potential problems. Software may also be able to respond more intelligently, avoiding a hazard rather than shutting down the system into a safe mode. The increased reliance on software means that the safety and reliability of the software becomes a vital component in a safe system.

## *2.1 What is a Hazard?*

A hazard is the presence of a potential risk situation that can result in or contribute to a mishap. Every hazard has at least one cause, which in turn can lead to a number of effects (i.e., damage, illness, etc.).

A hazard cause is typically a fault, error, or defect in either the hardware or software or a human operator error, that results in a hazard. A hazard control is a method for preventing the hazard or reducing the risk of the hazard occurring . Hazard controls use hardware (e.g. pressure relief valve), software (e.g. detection of stuck valve and automatic response to open secondary valve), operator procedures, or a combination of methods to avert the hazard. For every hazard cause there must be at least one control method, where control method is usually a design feature (hardware and/or software), or a procedural step. Examples of hazard causes and controls are given in *Table 2-1 Hazard Causes and Controls - Examples*.

Software faults can cause hazards and software can be used to control hazards. Some software hazard causes can be addressed with hardware hazard controls, although this is becoming less and less practical as software becomes more complex. For example, a hardwired gate array could be preset to look for certain predetermined hazardous words (forbidden or unplanned) transmitted by a computer, and shut down the computer upon detecting such a word. In practice, this is nearly impossible today because thousands of words and commands are usually sent on standard buses.

## 2.2 What Makes Software Hazardous?

Software is hazardous if it

- ✓ Controls hazardous or safety critical hardware
- ✓ Monitors safety critical hardware as part of a hazard control
- ✓ Provides information upon which a safety-related decision is made
- ✓ Performs analysis that impacts automatic or manual hazardous operations
- ✓ Verifies hardware hazard controls

Either remote or embedded real-time **software that controls or monitors hazardous or safety critical hardware is considered hazardous**. For example, software that controls an airlock or operates a high-powered laser is hazardous.

Software which does not control or monitor a real world (real time or near real time) process is normally not hazardous. Software safety therefore concerns itself mostly with software that controls real world processes, whose malfunction can result in a hazard. However, other types of software can also be the subject of software safety.

If the **software resides with safety critical software on the same physical platform**, it must also be considered safety critical unless adequately partitioned from the safety critical portion. Techniques such as firewalls and partitioning can be used to ensure that the non-critical software does not interrupt or disrupt the safety critical functions and operations. In addition, **software that monitors physical entities**, such as a pressure or temperature, is safety critical if the results they provide are used in making a safety-related decision.

**Software that performs off-line processes** may be considered safety critical as well. For example, software that verifies a software or hardware hazard control must operate correctly. Failure of the test software may allow a potential hazard to be missed. In addition, **software used in analyses that verify hazard controls or safety critical software** must also function correctly, to prevent inadvertently overlooking a hazard.

Computer simulations and models, such as finite element analyses programs (e.g. Nastran or Ansys), have become integrated into the design process and interface directly with CAD programs. These software packages "automatically" perform everything from linear and nonlinear stress analyses, to a variety of steady state and dynamic characteristic analyses, to the modeling of crash tests. These modeling programs are used to examine a variety of things from children's toys to rocket engines. The use of computer modeling for design evaluation will continue to expand since the cost savings potential from limiting or eliminating prototype testing will continue to drive industry.

The growing dependence on such computer modeling may lead to the potential misuse of these simulations. There is a tendency to assume the software does the analysis correctly and completely. The analyst must do "sanity checks" of the results, as a bare minimum, to verify that the modeling software is functioning correctly. For safety critical analyses, the modeling tool should be formally verified or tested. In addition, the analyst needs to have knowledge of the proper application of these "virtual design environments". Therefore, users of software simulation programs that analyze safety critical hardware should be certified. All program input parameters such as loads, restraints, and materials properties, should be independently verified to assure proper modeling and analysis.

### 2.2.1 What is Safety Critical Software?

Software that controls (activates, deactivates, etc.) functions which if performed or, if prevented from occurring, could result in injury to persons or damage to equipment is safety critical software.

Software subsystem hazard analyses should be performed on any safety critical functions involving:

- a hazard cause
- a hazard control
- software providing information upon which safety critical decisions are made
- software used as a means of failure/fault detection

Also, "contamination" of safety critical software by uncontrolled software sharing the same physical platform must be considered. Software that might not be safety critical in and of itself, could lock up the computer or write over critical memory areas when sharing a CPU or any routines with the safety critical software.

### 2.2.2 How Does Software Control Hazards?

Many hardware hazard causes can be addressed with software hazard controls. From a historic standpoint hardware controls were preferred. But often it is not feasible to have only hardware controls, or sometimes to have any hardware controls at all. Increasingly, hazard responses are delegated to software because of the quick reaction time needed and as personnel are replaced by computers.

For example, software can detect hazardous hardware conditions (via instrumentation) and execute a corrective action and/or warn operators. Software could detect what operational state a system is in and prevent certain activities which would be hazardous in that state, e.g. unlocking and opening a furnace door when the touch temperature is too high in the “furnace on” state.

### 2.2.3 What About Hardware Controls?

NASA relies primarily on hardware controls, in conjunction with software controls, to prevent hazards. Often software is the “first line of defense”, monitoring for unsafe conditions and responding appropriately. The software may perform an automatic safing operation, or provide a message to a human operator, for example. The hardware control is the backup to the software control. If the software fails to detect the problem or does not respond properly to alleviate the condition, then the hardware control is triggered.

Using a pressurized system as an example, the software monitors a pressure sensor. If the pressure goes over some threshold the software would respond by stopping the flow of gas into the system by closing a valve. If the software failed, either by not detecting the over-pressurization or by not closing the valve, then the hardware pressure relief valve would be used once the pressure reached a critical level.

While software controls can be, and are, used to prevent hazards, they must be implemented with care. Special attention needs to be placed on this software during the development process. Formal inspections of software controls is highly recommended. In addition, testing of the software control needs to be thorough and complete.

For catastrophic hazards, NASA allows software to be only one of three hazard controls. In each case where software is a potential hazard cause or is utilized as a hazard control, the software is "Safety Critical" and should be analyzed, and its development controlled.

**Table 2-1 Hazard Causes and Controls - Examples**

| Cause | Control | Example of Control Action |
|---|---|---|
| Hardware | Hardware | Pressure vessel with pressure relief valve. |
| Hardware | Software | Fault detection and safing function; or arm/fire check which activate or prevent hazardous conditions. |
| Hardware | Operator | Operator opens switch to remove power from failed unit. |
| Software | Hardware | Hardwired timer or discrete hardware logic to screen invalid commands or data. Or sensor directly triggering a safety switch to override a software control system. |
| Software | Software | Two independent processors, one checking the other and intervening if a fault is detected. Emulating expected performance and detecting deviations. |
| Software | Operator | Operator sees control parameter violation on display and terminates process. |
| Operator | Hardware | Three electrical switches in series in a firing circuit to tolerate two operator errors. |
| Operator | Software | Software validation of operator initiated hazardous command; or software prevents operation in unsafe mode. |
| Operator | Operator | Two crew members, one commanding the other monitoring. |

### 2.2.4 Caveats with Software Controls

When software is used to control a hazard, some care must be made to isolate it from the hazard cause it is controlling. For a hazard cause outside of the computer processing arena (e.g. stuck valve), the hazard control software can be co-located with the regular operations software. Partitioning of the hazard control software is recommended. Otherwise, all of the software must be treated as safety critical because of potential "contamination" from the non-critical code.

If the hazard cause is erroneous software, then the hazard control software can reside on a separate computer processor from the one where the hazard/anomaly might occur. Another option would be to implement a firewall or similar system to isolate the hazard control software, even though it shares the same processor as the hazard cause.

If the hazard cause is a processor failure, then the hazard control *must* be located on another processor, since the failure would most likely affect its own software's ability to react to that CPU hardware failure. This is a challenging aspect of software safety, because multiprocessor architectures are costly and system designers often prefer to use single processor designs, which are not failure tolerant. Also, system designers are sometimes reluctant to accept the NASA axiom that a single computer is inherently zero failure tolerant. Many also believe that computers fail safe, whereas NASA experience has shown that computers may exhibit hazardous failure modes. Another fallacy is to believe that upon any fault or failure detection, the safest action is always to shut down a program or computer automatically. This action can cause a more serious hazardous condition.

### 2.2.5 What is Fault Tolerance?

Fault tolerance is the ability of the system to withstand an unwanted event and maintain a safe condition. It is determined by the number of failures which can occur in a system or subsystem without the occurrence of a hazard. A one fault tolerant system has two hazard controls, so that one will still be operational if the other fails. A two fault tolerant system has three controls for the hazard.

NASA, based on extensive experience with spacecraft flight operations, has established levels of failure tolerance based on the hazard severity level necessary to achieve acceptable levels of risk.

- Catastrophic Hazards must be able to tolerate two hazard control failures.
- Critical Hazards must be able to tolerate a single hazard control failure.

## 2.3 The System Safety Program

A System Safety Program Plan is a prerequisite to performing analysis or development of safety critical software. The System Safety Program Plan outlines the organizational structure, interfaces, and the required criteria for analysis, reporting, evaluation, and data retention to provide a safe product. This safety plan describes forms of analysis and provides a schedule for performing a series of these system and subsystem level analyses throughout the development cycle. It also addresses how safety analyses results and the sign-off and approval process should be handled.

System safety program analyses follow the life cycle of the system development efforts. The system is comprised of the hardware, the software, and the interfaces between them (including human operators). What generally happens in the beginning of program development is that the hardware is conceived to perform the given tasks and the software concept is created that will operate and/or interact with the hardware. As the system develops and gains maturity, the types of safety analyses go from a single, overall assessment to ones that are more specific.

While software is often considered a SUBSET of the complete system (a sub-system), it is actually a "coexisting" system, acting with and upon the hardware system. Software should always be considered in a systems context. Software often commands and monitors the system functions, as well as performing communications, data storage, sensor interpretation, etc., and is critical to even having a system!

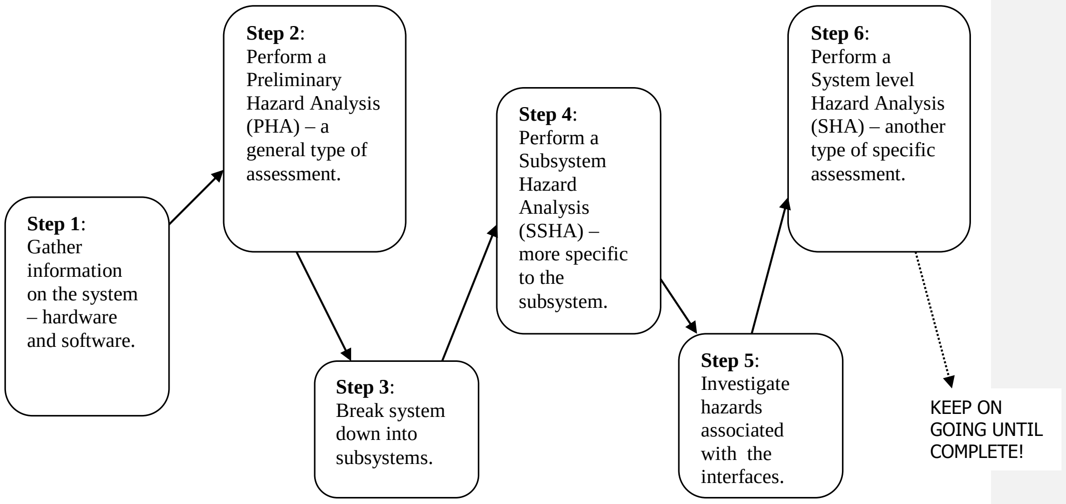

Step 1: Gather information on the system – hardware and software.
Step 2: Perform a Preliminary Hazard Analysis (PHA) – a general type of assessment.
Step 3: Break system down into subsystems.
Step 4: Perform a Subsystem Hazard Analysis (SSHA) – more specific to the subsystem.
Step 5: Investigate hazards associated with the interfaces.
Step 6: Perform a System level Hazard Analysis (SHA) – another type of specific assessment.
KEEP ON GOING UNTIL COMPLETE!

**Figure 2-1 Hazard Analysis**

The System Safety Program Plan should describe interfaces within the Assurance disciplines as well as the other Project disciplines. All analyses and tasks should be complementary and supportive, regardless of which group (development or assurance) has the responsibility. The analyses and tasks may be shared between the groups, and within each discipline, according to the resources and expertise of the project personnel.

Concurrent engineering can help to provide better oversight, allowing information and ideas to be exchanged between the various disciplines, reduce overlapping efforts, and improve communications throughout the Project. Safety and Assurance personnel bring a safety “point of view” to a project, and should be included at the earliest possible stage. The information obtained and rapport established by being an early member of the “team” will go a long way in solving problems later in the Project. Helping the project to design in safety from the beginning is far easier and cheaper than late-stage alterations or additions.

The Software System Safety Handbook [4] produced by the Department of Defense has an excellent reference to system safety from a risk management perspective. Chapter 3 of that document goes into detail about how risk and system safety are intertwined. Chapter 4 goes into detail on planning a software safety program, including hazard analyses. Appendix E of that document details generic requirements and guidelines for software development and test.

### 2.3.1 Safety Requirements Determination

Depending on the program or project, there are many applicable safety requirements. These requirements may be levied from standards, from the organization, or may be written into a contract. They provide the minimum boundaries that must be met to ensure that the system is safe and will not function in a manner that is harmful to people, itself, other systems, or the environment. Safety requirements can include those specific to NASA, the FAA, the

Department of Transportation (DOT), and even the Occupational Health and Safety Administration (OSHA).

Once the safety requirements have been identified and the available system information is gathered, it is time to perform the first analysis. A common assessment tool used during this beginning activity is the Preliminary Hazard Analysis (PHA). This analysis tool will be discussed in more detail in *Section 2.4*. The results of the PHA are a list of hazard causes and a set of candidate hazard controls. Any software hazard cause and software hazard controls or mitigation are taken forward as inputs to the software safety requirements flow-down process.

System hazard controls should be traceable to system requirements. If controls identified by the PHA are not in the system specification, it should be amended by adding safety requirements to control the hazards. Then the process of flow-down from system specification to software specification will include the necessary safety requirements.

At least one software requirement is generated for each software hazard control. Each requirement is incorporated into the Software Requirements Specification (SRS) as a safety critical software requirement.

**Any software item identified as a potential hazard cause, control, or mitigation, whether controlled by hardware, software or human operator, is designated as safety critical, and subjected to rigorous software quality control, analysis, and testing. It is also traced through the software safety analysis process until the final verification.**

## *2.4 Preliminary Hazard Analysis (PHA)*

An aerospace system generally contains three elements: hardware, software and one or more human operators (e.g., ground controllers, mission specialists or vehicle pilots). The hardware and software elements can be broken down further into subsystems and components. While individual elements, subsystems or components are often non-hazardous when considered in isolation, the combined system may exhibit various safety risks or hazards. This is especially true for software.

Although it is often claimed that “software cannot cause hazards”, this is only true where the software resides on a non-hazardous platform and does not interface or interact with any hazardous hardware or with a human operator. Similarly, a hardware component such as an electrical switch or a fluid valve might be non-hazardous as a stand-alone component, but may become hazardous or safety critical when used as an inhibit in a system to control a hazard.

Before any system with software can be analyzed or developed for use in hazardous operations or environment, a System PHA must be performed. Once initial system PHA results are available, safety requirements flow down and subsystem and component hazard analyses can begin. The PHA is the first source of “specific” software safety requirements, (i.e., unique to the particular system architecture). It is a prerequisite to performing any software safety analysis.

It is important when doing a PHA to consider how the software interacts with the rest of the system. Software is the heart and brains of most modern, complex systems, controlling and monitoring almost all operations. When the system is decomposed into sub-elements, how the software relates to each component should be considered. The PHA should also look at how the component may feed back to the software (e.g. failed sensor leading the software to respond inappropriately).

The PHA is the first of a series of system level hazard analyses, whose scope and methodology is described in the NASA NPG 8715.3 NASA Safety Manual [1], NSTS 13830 Implementation Procedure for NASA Payload System Safety Requirements [2], and NSTS-22254 Methodology for Conduct of Space Shuttle Program Hazard Analyses [3]. Section 2.4.3 contains a list of resources that describe the process of performing a Preliminary Hazard Analysis. While this guidebook will not duplicate that information, the following sections summarize the methodology used in a PHA for software developers, managers, and others unfamiliar with NASA system safety.

Note that the PHA is not a NASA-specific analysis, but is used throughout industry. IEEE 1228, Software Safety Plans, also requires that a PHA be performed.

### 2.4.1 PHA Approach

The following is an excerpt from NPG 8715.3 Appendix-D:

"In many ways the PHA is the most important of the safety analyses because it is the foundation on which the rest of the safety analyses and the system safety tasks are built. It documents which generic hazards are associated with the design and operational concept. This provides the initial framework for a master listing (or hazard catalog) of hazards and associated risks that require tracking and resolution during the course of the program design and development. The PHA also may be used to identify safety-critical systems that will require the application of failure modes and effects analysis and further hazard analysis during the design phases.

The program shall require and document a PHA to obtain an initial listing of risk factors for a system concept. The PHA effort shall be started during the concept exploration phase or earliest life cycle phases of the program. A PHA considers hardware, software, and the operational concepts. Hazards identified in the PHA will be assessed for risk based on the best available data, including mishap data from similar systems, other lessons learned, and hazards associated with the proposed design or function. Mishap and lessons learned information are available in the Incident Reporting Information System (IRIS) and the Lessons Learned Information System (LLIS). The risk assessment developed from the PHA will be used to ensure safety considerations are included in tradeoff studies of design alternatives; development of safety requirements for program and design specifications, including software for safety-critical monitor and control; and definition of operational conditions and constraints.

Extensions and refinements of the PHA should coincide with the development of the design after the conceptual phase. A system generally consists of several discrete subsystems that should be individually analyzed in subsystem hazard analysis (SSHA). The results of the SSHA`s in turn feed into the SHA, which will integrate its subsystems and identify hazards that cross the subsystem interfaces. The number of systems and subsystems in a program is a function of the complexity of individual projects and will be determined by the program."

See *Table 2-2 Generic Hazards Checklist* for a sample checklist of generic hazards. The last row gives some examples of how software can function as a control for a hazard. It is important to understand that this and other checklists are merely tools to encourage the thought process. Keep thinking and brainstorming for all the permutations of potential hazards, causes, and controls for a given system.

**Table 2-2 Generic Hazards Checklist**

| ***Generic Hazard*** | **Contamination / Corrosion** | **Electrical Discharge / Shock** | **Environmental / Weather** | **Fire / Explosion** | **Impact / Collision** | **Loss of Habitable Environment*** | **Pathological / Physiological/ Psychological** | **Radiation** | **Temperature Extremes** |
|---|---|---|---|---|---|---|---|---|---|
| | Chemical Disassociation<br><br>Chemical Replacement / Combination<br><br>Moisture<br><br>Oxidation<br><br>Organic (Fungus, Bacterial, Etc.)<br><br>Particulate<br><br>Inorganic (Includes Asbestos) | External Shock<br><br>Internal Shock<br><br>Static Discharge<br><br>Corona<br><br>Short | Fog<br><br>Lightning<br><br>Precipitation (Fog, Rain, Snow, Sleet, Hail)<br><br>Sand / Dust<br><br>Vacuum<br><br>Wind<br><br>Temperature Extremes | Chemical Change (Exothermic, Endothermic)<br><br>Fuel & Oxidizer in Presence of Pressure and Ignition Source<br><br>Pressure Release / Implosion<br><br>High Heat Source | Acceleration (Including Gravity)<br><br>Detached Equipment<br><br>Mechanical Shock / Vibration / Acoustical<br><br>Meteoroids / Meteorites<br><br>Moving / Rotating Equipment | Contamination<br><br>High Pressure<br><br>Low Oxygen Content<br><br>Low Pressure<br><br>Toxicity<br><br>Low Temperature<br><br>High Temperature | Acceleration / Shock / Impact / Vibration<br><br>Atmospheric Pressure (High, Low, Rapid Change)<br><br>Humidity<br><br>Illness<br><br>Noise<br><br>Sharp Edges<br><br>Sleep, Lack of<br><br>Visibility (Glare, Surface Fogging)<br><br>Temperature<br><br>Workload, Excessive | EMI<br><br>Ionizing Radiation (Includes Radon)<br><br>Non-ionizing Radiation (Lasers, Etc.) | High<br><br>Low<br><br>Variations |
| ***Software Controls Example*** | Receive data input from hardware sensors (gas chromatograph, particle detector, etc.). Activate caution and warning indicators if levels surpass programmed limits, and/or automatically shutdown sources or activate fans. | Shut down power prior to access of electrical components. | Receive data input from sensor readings of hardware devices (particle detector, wind velocity probe, etc.). Send commands to shut down hardware if programmed limits are surpassed. | Monitor temperature, activate fire suppression system if temperature goes over set threshold. | Monitor position of rotating equipment. Keep position within defined limits, or shutdown motion if exceeding limits. | Receive data input from sensor readings of hardware devices. Send commands to operate proper sequencing of valve operation. | Monitor pressure and rate of change. Control pressure system to keep rate of change under set limit. | Receive data input from sensor readings of hardware devices. Shut down high gain antenna when operational time limit is reached. | Monitor temperature. Sound warning if temperature outside of limits |

*Health issues require coordination with Occupational Health Personnel

### 2.4.1.1 Identifying Hazards

Preliminary hazard analysis of the entire system is performed from the top down to identify hazards and hazardous conditions. Its goal is to identify all credible hazards up front. Initially the analysis is hardware driven, considering the hardware actuators, end effects and energy sources, and the hazards that can arise. For each identified hazard, the PHA identifies the hazard causes and candidate control methods. These hazards and hazard causes are mapped to system functions and their failure modes. Most of the critical functions are associated with one or more system controls. These control functions cover the operation, monitoring and/or safing of that portion of the system safety assessment and must consider the system through all the various applicable subsystems including hardware, software, and operators.

To assure full coverage of all aspects of functional safety, it can be helpful to categorize system functions as two types:

1. "Must work" functions (MWF's)
2. "Must not work" functions (MNWF's)

The system specification often initially defines the criticality (e.g., safety critical) of some system functions, but may be incomplete. This criticality is usually expressed only in terms of the Must-Work nature of the system function, and often omits the Must-Not-Work functional criticality. The PHA defines all the hazardous MNWF's as well as the MWF's.

Examples:

1. A science experiment might have a system Criticality designation of 3 (Non-critical) in terms of its system function, because loss of the primary experiment science data does not represent a hazard. However, the experiment might still be capable of generating hazards such as electric shock due to inadvertent activation of a power supply during maintenance. Activation of power during maintenance is a MNWF.
2. An experiment might release toxic gas if negative pressure (vacuum) is not maintained. Maintaining a negative pressure is a MWF.
3. An air traffic control system and aircraft flight control systems are designed to prevent collision of two aircraft flying in the same vicinity. Collision avoidance is a MWF.
4. A spacecraft rocket motor might inadvertently ignite while it is in the STS Cargo Bay. Motor ignition is a MNWF, at that time. It is apparent that the MNWF becomes a MWF when it is time for the motor to fire.

If functions identified in the PHA were not included in the system specification, it should be amended to address control of those functions.

### 2.4.1.2 Risk Levels

The following definitions of hazard severity levels are from NASA NPG 8715.3.

| Hazard Severity Definitions | **Catastrophic**<br><br>Loss of entire system; loss of ground facility; loss of human life or permanent disability | **Critical**<br><br>Major system damage; severe injury or temporary disability |
|---|---|---|
| | **Moderate**<br><br>Minor system damage; minor injury | **Negligible**<br><br>Some system stress, but no system damage; no or minor injury |

Likelihood of occurrence are assigned probabilities that are determined by the project or program. The possibility that a given hazard may occur is based on engineering judgment. The following definitions of likelihood of occurrence are provided as an example only, and are based on NPG 8715.3 NASA Safety Manual.

| Likelihood of Occurrence Definitions | **Likely** | **Probable** |
|---|---|---|

| | | |
|---|---|---|
| | A one in ten chance (or greater) that the event will happen | A one-in-hundred to one-in-ten chance that the event will occur |
| **Possible**<br><br>Greater than a one in a thousand chance that the event will occur | **Unlikely**<br><br>Rare, less than one in a thousand chance the event will happen | **Improbable**<br><br>Very rare, possibility is like winning the lottery |

Hazards are prioritized by the system safety organization in terms of their severity and likelihood of occurrence, as shown in *Table 2-3 Hazard Prioritization - System Risk Index*.

**Table 2-3 Hazard Prioritization - System Risk Index**

| **Severity Levels** | **Likelihood of Occurrence** | | | | |
|---|---|---|---|---|---|
| | Likely | Probable | Possible | Unlikely | Improbable |
| Catastrophic | 1 | 1 | 2 | 3 | 4 |
| Critical | 1 | 2 | 3 | 4 | 5 |
| Moderate | 2 | 3 | 4 | 5 | 6 |
| Negligible | 3 | 4 | 5 | 6 | 7 |

1 = Highest Priority (Highest System Risk), 7 = Lowest Priority (Lowest System Risk)

The hazard prioritization is important for determining allocation of resources and acceptance of risk. Hazards with the highest risk, Level 1, are not permitted in a system design. A system design exhibiting “1” for hazard risk level must be redesigned to eliminate the hazard. The lowest risk levels, "5" and above, require minimal, if any, safety analysis or controls. For the three levels of risk in-between, the amount of safety analysis required increases with the level of risk. The extent of a safety effort is discussed within *Section 3*, where three levels of safety analysis are described, i.e. Minimum, Moderate and Full. These correspond to risk as follows:

**Table 2-4 System Risk Level**

| **System Risk Level** | **Class of Safety Analysis Recommended** |
|---|---|
| 1 | Not Applicable (Prohibited) |
| 2 | Full |
| 3 | Moderate |
| 4,5* | Minimum |
| 6,7 | None (Optional) |

*Level 5 systems fall between Minimum and Optional, and should be evaluated to determine the class of safety analysis required.

#### 2.4.1.3 NASA Policy for Hazard Elimination/Control

The NASA policy towards hazards of Risk Level 2, 3 or 4/5 is defined in NPG 8715.3 Section 3.4, as follows: “Hazards will be mitigated according to the following stated order of precedence:

- **Eliminate Hazards**

  Hazards are eliminated where possible. This is accomplished through design, such as by eliminating an energy source. For example, software could have access to a pressure control. If software access to the control is not needed, and malfunctioning software could lead to a hazard, then preventing software's access to the control eliminates the hazard.

- **Design for Minimum Hazards**

  Hazards that cannot be eliminated must be controlled. For those hazards, the PHA evaluates what can cause the hazards, and suggests how to control the potential hazard causes. Control by design is preferred. The hazard may be minimized by providing failure tolerance (e.g. by redundancy - series and/or parallel as appropriate), by providing substantial margins of safety, or by providing automatic safing. For example, software verifies that all conditions are met prior to ignition of rocket engines.

- **Incorporate Safety Devices**

  For example, a Fire Detection and Prevention system to detect and interact with a fire event. Software may be a part of these devices, and may also provide the trigger for the safety device.

- **Provide Caution And Warning Devices**

  Software may monitor a sensor and trigger the caution/warning. Any software used in these caution and warning devices is safety critical.

- **Develop Administrative Procedures and Training**

  Control by procedure is sometimes allowed, where sufficient time exists for a flight crew member or ground controller to perform a safing action. This concept of "time to criticality" was used in the NASA Space Station Freedom program.

### 2.4.2 Preliminary Hazard Analysis Process

First, System and Safety experts examine the proposed system concepts and requirements and identify System Level Hazards considering areas such as power sources, chemicals usage, mechanical structures, time constraints, etc. (as per *Section 2.4.1.1 Identifying Hazards* above).

Next, the hazard cause(s) are identified. Common hazard cause categories include: collision, contamination, corrosion, electrical shock/damage, explosion, fire, temperature extremes, radiation, illness/injury and loss of capability. Each hazard has at least one cause, such as a hardware component failure, operator error, or software fault. The PHA should identify some or all of the hazard causes, based on the system definition at that point in the development effort. Consideration of these categories as risks early in the development effort reduces the chance that any of these surface as a problem on a project.

Next, at least one hazard control must be identified for each hazard cause. NASA safety standards often stipulate the methods of control required for particular hazard causes. This is not necessary for PHA, but is necessary at later phases in the system development process. Each control method must be a "real feature", usually a design feature (hardware and/or software), or a procedural sequence and must be verifiable.

For each hazard control at least one verification method must be identified. Verification can be by analysis, test or inspection. In some cases, the verification method is defined by established NASA safety requirements (e.g. Payload Safety Requirements NSTS 1700.1 (V1-B)).

Each system hazard is documented in a "Hazard Report". Required data items for a Hazard Report are described below.

*Figure 2-2 Payload Hazard Report Form*, shows the NASA Shuttle / Station Payload Hazard Report, which identifies hazards, their causes, controls, verification methods and verification status. Detailed instructions for completing this form are given as an appendix when the form is downloaded from the NASA Payload Safety Homepage [http://wwwsrqa.jsc.nasa.gov/pce] and also in NPG 8715.3 NASA Safety Manual, Chapter-3, System Safety, and Appendix-D (Analysis Techniques) [1]. This form is offered as a good example. NASA does not have an agency wide standard except SER Payloads at this time so each project or group may develop their own. These reports once created are revisited and updated on subsequent Safety Analyses throughout the life cycle of the system. A summary of the contents of the form is described below:

- **Hazard Description**

  This describes a system hazard, such as an uncontrolled release of energy resulting in a mishap.

- **Safety Requirement**

  This can be a hardware or software requirement and is usually a system level requirement. It can result in further flow-down to software functionality by identifying software Hazard Controls as described below.

- **Hazard Cause**

  This is usually a fault or defect in hardware or software. Software causes include:

  - Failure to detect a problem
  - Failure to perform a function
  - Performing a function at the wrong time, out of sequence, or when the program is in the wrong state
  - Performing the wrong function
  - Performing a function incompletely
  - Failure to "pass along" information or messages

- **Hazard Control**

  This is usually a design feature to control a hazard cause. The hazard control should be related to the applicable safety requirements cited by the hazard report. For example, where independence and fault tolerance are required, the hazard control block describes how the design meets these requirements.

Some formats of hazard reports include a block to describe:

* **Hazard Detection Method**

  This is the means to detect imminent occurrence of a hazardous condition as indicated by recognizing unexpected values of measured parameters. In *Figure 2-2 Payload Hazard Report Form* it is implicitly included in the "Hazard Control" section.

- **Safety Verification Method**

  This identifies methods used to verify the validity of each Hazard Control. These methods include analysis, test, demonstration or inspection.

- **Status of Verification**

  This identifies scheduled or actual start and completion dates of each verification item, and if the item is open or closed at the time of writing.

Since all required information is typically not available at the start of the development life-cycle, details for the various items are filled in and expanded during the development life-cycle. Hazard causes and controls are identified early in the process and verifications are addressed in later life-cycle phases.

### 2.4.3 Tools and Methods for PHA

Tools and methods for performing a formal Preliminary Hazard Analysis are detailed in the following documents:

- NSTS 22254, Methodology for Conduct of Space Shuttle Program Hazard Analysis
- SOFTWARE SYSTEM SAFETY HANDBOOK, A Technical & Managerial Team Approach, December 1999 (Joint Software System Safety Committee)
- MIL-STD-882B, Task 201 (PHL) and Task 202 (PHA)
- http://books.usapa.belvoir.army.mil/cgi-bin/bookmgr/BOOKS/P385_16/FIGFIGUNIQ10

In addition, many system safety books describe the process of conducting a Preliminary Hazard Analysis.

**Figure 2-2 Payload Hazard Report Form**

| | | |
|---|---|---|
| **PAYLOAD HAZARD REPORT** | | a. NO: |
| b. PAYLOAD: | | c. PHASE: |
| d. SUBSYSTEM: | e. HAZARD GROUP: | f. DATE: |
| g. HAZARD TITLE: | | i. HAZARD CATEGORY<br>☐ CATASTROPHIC<br>☐ CRITICAL |
| h. APPLICABLE SAFETY REQUIREMENTS: | | |
| j. DESCRIPTION OF HAZARD: | | |
| k. HAZARD CAUSES: | | |
| l. HAZARD CONTROLS: | | |
| m. SAFETY VERIFICATION METHODS: | | |
| n. STATUS OF VERIFICATION: | | |

| o. APPROVAL | PAYLOAD ORGANIZATION | SSP/ISS |
|---|---|---|
| PHASE I | | |
| PHASE II | | |
| PHASE III | | |

JSC Form 542B (Rev November 22, 1999) (MS Word September 1997)

### 2.4.4 PHA is a Living Document

The PHA is not final because of the absence of design maturity. Later in the design process hazards may be added or deleted, and additional hazard analysis performed. But an initial set of hazards must be identified early in order to begin safety engineering tasks in a timely manner to avoid costly design impacts later in the process. The PHA is also required before software subsystem hazard analysis can begin. Those hazard causes residing in the software portion of a control system become the subject of the software subsystem hazard analysis. **It is important to reexamine software's role and safety impact throughout the system development phases.** Software is often relied on to work around hardware problems encountered which result in additions and/or changes to functionality.

## *2.5 Software Subsystem Hazard Analysis*

After safety critical software is identified in the first cycle of the PHA, software hazard analysis can begin. The first cycle of system hazard analysis and software hazard analysis are top-down only. Bottom-up analyses take place after a sufficient level of design detail is available. The first cycle of bottom up analysis is "Criticality analysis" of requirements, (as described in *Section 5.1.2 Requirements Criticality Analysis*). At this early phase, only the highest level of definition is available for functions that the software will perform and these may change as development proceeds.

Many analysis and development techniques exist for safety critical software, and there is no "one size fits all" approach. The extent of software safety effort required must be planned based on the degree of safety risk of the system. Hazards are prioritized by the system safety organization in terms of their severity and likelihood of occurrence, as shown in *Table 2-3 Hazard Prioritization - System Risk Index*. This system risk index identifies three levels of system safety effort. *Section 3* will expand upon this, and provide three levels of *software* safety effort, based on software's control of and responses to the hazardous hardware

*Section 3. SOFTWARE SAFETY PLANNING*, describes the levels of software safety effort, and presents guidelines on which analysis and development techniques may be selected for each. Strategies for tailoring the safety effort are also provided. *Section 4. SAFETY CRITICAL SOFTWARE DEVELOPMENT* describes software development tasks and techniques in detail. *Section 5. SOFTWARE SAFETY ANALYSIS* provides many analysis tasks and techniques that can be used throughout the software development lifecycle. *Section 6. SOFTWARE DEVELOPMENT ISSUES* discusses programming languages, tools, new technology, and programming practices from a safety point of view. Section 7. Software Acquisition discusses using off-the-shelf, previously developed, and contractor acquired software.

# 3. SOFTWARE SAFETY PLANNING

The following section describes how to plan a software safety effort, and how to tailor it according to the risk level of the system. In *Section 2.4.1.2 Risk Levels*, determination of the level of safety risk inherent in the system was presented. *Section 3.2 Scope of Software Subsystem Safety Effort* discusses tailoring the level of effort for both software development and software analysis tasks performed by software development personnel and software safety personnel.

On the development side, the software safety engineer works in conjunction with the system safety organization to develop software safety requirements, distribute those safety requirements to the software developers, and monitor their implementation. On the analysis side, the software safety engineer analyzes software products and artifacts to identify new hazards and new hazard causes to be controlled, and provides the results to the system safety organization to be integrated with the other (non-software) subsystem analyses.

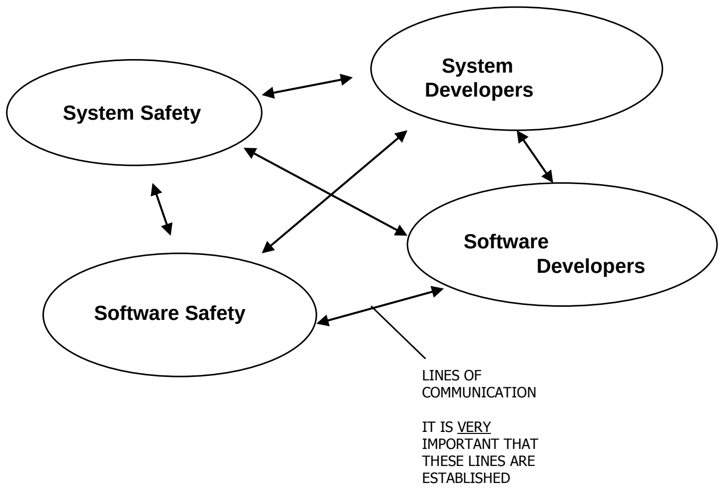


**Figure 3-1 Elements of a Safety Process**

The following sections of the guidebook will discuss the various analysis and development tasks used in the software development life cycle as depicted in a typical waterfall model.

## 3.1 Software Development Life-cycle Approach

*Table 3-1 NASA Software Life-cycle - Reviews and Documents* shows the typical NASA software waterfall design life-cycle phases and lists the reviews and deliverable project documents required at each life-cycle phase. Each of these reviews and project documents should contain appropriate references and reports on software safety. Software safety tasks and documents for each design life-cycle phase are also shown.

If a particular software safety task or document is defined elsewhere in this guidebook, the section number where it can be found is shown next to the name. Software development tasks and documents listed in the table will be described in the following subsections, organized by the life-cycle phase in which they are conducted or produced.

In rapid prototyping environments, or spiral life cycle environments, software artifacts, including prototype codes, are created early (i.e. in requirements or architectural design phases). The early artifacts are evaluated and performance results used to modify the requirements. Then the artifacts are regenerated using the new requirements. This process may go through several iterations. In these environments, Safety Analyses are also done in several smaller iterative steps. Safety and software development must work closely together to coordinate the best time to perform each round of analysis.

**At the end of a lifecycle activity** or phase, it is important to **verify** that

- ❖ All system safety requirements have been satisfied by this lifecycle phase.
- ❖ No additional hazards have been introduced by the work done during this lifecycle phase.

IEEE 1228-1994

**Table 3-1 NASA Software Life-cycle - Reviews and Documents**

| Life-cycle Phases | Milestone Reviews | Software Safety Tasks | Documents |
|---|---|---|---|
| Software Concept and Initiation<br><br>(Project System and Subsystem Requirements and Design Development) | SCR - Software Concept Review<br><br>Software Management Plan Review<br><br>Phase-0 Safety Review | Tailoring Safety Effort<br><br>Preliminary Hazard Analysis (PHA)<br><br>PHA Approach<br><br>Phase 0/1 Safety Reviews<br><br>Hazards Tracking and Problem Resolution | Software Management Plan<br><br>Software Systems Safety Plan<br><br>Software Configuration Management Plan<br><br>Software Quality Assurance Plan<br><br>Risk Management Plan |

| Life-cycle Phases | Milestone Reviews | Software Safety Tasks | Documents |
|---|---|---|---|
| Software Requirements | SRR - Software Requirements Review<br><br>Phase-1 Safety Review | ~~2.2~~ Safety Requirements Flow-down<br>Generic Software Safety Requirements<br>~~4.2.1~~ Development of Software Safety Requirements<br>~~5.1~~ Software Safety Requirements Analysis<br>~~5.1.2~~ Requirements Criticality Analysis<br>~~5.1.3~~ Specification Analysis<br>Software Fault Tree Analysis (SFTA)<br>Timing, Throughput, and Sizing Analysis | Software Requirements Document<br>Formal Inspection |
| Software Architectural or Preliminary Design | Software Preliminary Design Review (PDR)<br><br>Phase-1/2 Safety Review | ~~5.2.1~~ Update Criticality Analysis<br>~~5.2.2~~ Conduct Hazard Risk Assessment<br>~~5.2.3~~ Analyze Architectural Design<br>~~5.2.4.1~~ Interdependence Analysis<br>~~5.2.4.2~~ Independence Analysis<br>Formal Methods and Model Checking | Preliminary Acceptance Test Plan<br>Software Design Specification - (Preliminary)<br>Formal Inspection |
| Software Detailed Design | Software Critical Design Review (CDR)<br><br>Phase-2 Safety Review | Design Logic Analysis<br>~~5.3.2~~ Design Data Analysis<br>Design Interface Analysis<br>~~5.3.4~~ Design Constraint Analysis<br>Dynamic Flowgraph Analysis<br>~~5.3.13~~ Requirements State Machines<br>Software Failure Modes and Effects Analysis | Final Acceptance Test Plan<br>Software Design Specification - (Final)<br>Formal Inspection Reports |

| Life-cycle Phases | Milestone Reviews | Software Safety Tasks | Documents |
|---|---|---|---|
| Software Implementation (Code and Unit Test) | Formal Inspections, Audits<br><br>Phase-2/3 Safety Review | ~~5.4.1~~ Code Logic Analysis<br>~~5.4.3~~ Code Data Analysis<br>~~5.4.4~~ Code Interface Analysis<br>~~5.4.7~~ Code Inspections, Checklists, and coding standards<br>Unused Code Analysis<br>Interrupt Analysis | Source code listings<br>Unit Test reports<br>Formal Inspection Reports |
| Software Integration and Test | Test Readiness Review<br><br>Phase-3 Safety Review | Testing Techniques<br>Regression Testing<br>Software Safety Testing<br>Test Witnessing<br>Test Coverage<br>Test Results Analysis<br>Reliability Modeling | Test Procedures<br>Test Reports<br>Problem/Failure Resolution Reports |
| Software Acceptance and Delivery | Software Acceptance Review<br><br>Phase-3 Safety Review | | Software Delivery Documents<br>Acceptance Data Package |
| Software Sustaining Engineering and Operations | | (Same activities as for development) | (Update of all relevant documents) |

## 3.2 Scope of Software Subsystem Safety Effort

In Section 2.4.1.2 Risk Levels, the System Risk Index was developed. This index specified the hazard risk for the system *as a whole*. The software element of the system inherits from the system risk, modified by the extent with which the software controls, mitigates, or interacts with the hazardous elements. Merging these two aspects is not an exact science, and the information presented in this section is meant to guide the tailoring of the safety effort. Use intelligence, and not blind adherence to a chart, when determining the level of safety effort.

Begin the process of tailoring the software safety effort by determining how much software is involved with the hazardous aspects of the system. Search the PHA, Software Hazard Analysis, Software Risk Assessments, and other analyses for systems and sub-systems that can be initially categorized as safety critical or not. Then look at how the software relates to the safety critical system components.

Tailoring the software safety effort can be accomplished by following three steps:

1. Identify safety critical software
2. Categorize safety critical software subsystems (i.e., how critical is it?)
3. Determine the extent of development effort and oversight required

### 3.2.1 Identify Safety Critical Software

Before deciding how to apply software safety techniques to a project, it is important to first understand the many factors which determine if there even is a safety concern.

The first determination to be made is whether there is any software in the system which either monitors, controls, interfaces with directly, or is resident in a processor handling critical or hazardous system functions. Other safety critical software performs analysis or crunches numbers that will be used with, or for, safety critical equipment. Note that non-safety critical software becomes safety critical if it can impact safety critical software resident with it (on the same machine).

Next it is important to determine the reliability of the system. Most aerospace products are built up from small, simple components to make large, complex systems. Electronic components usually have a small number of states and can be tested exhaustively, due to their low cost, to determine their inherent reliability. (Mechanical components are often another matter.) Reliability of a large hardware system is determined by developing reliability models using failure rates of the components that make up the system. Reliability and safety goals of hardware systems can usually be reached through a combination of redundant design configurations and selection of components with suitable reliability ratings.

Reliability of software is much harder to determine. Software does not wear out or break down but may have a large number of states that cannot be fully tested. An important difference between hardware and software is that many of the mathematical functions implemented by software are not continuous functions, but functions with an arbitrary number of discontinuities [4]. Although mathematical logic can be used to deal with functions that are not continuous, the resulting software may have a large number of states and lack regularity. It is usually impossible to establish reliability values and prove correctness of design by testing all possible states of a medium to large (more than 40,000-50,000 lines of code) software system within a reasonable amount of time, if ever. Furthermore, testing can only commence after preliminary code has been generated, typically late in the development cycle. As a result, it is very difficult to establish accurate reliability and design correctness values for software.

If the inherent reliability of software cannot be accurately measured or predicted, and most software designs cannot be exhaustively tested, the level of effort required to meet safety goals must be determined using other characteristics of the system. The following characteristics have a strong influence on the ability of software developers to create reliable safe software:

- **Degree of Control**

  *The degree of control that the software exercises over safety-critical functions in the system.*

  Software which can cause a hazard if it malfunctions is considered safety critical software. Software which is required to either recognize hazardous conditions and implement automatic safety control actions, or which is required to provide a safety critical service, or to inhibit a hazardous event, will require more software safety resources and detailed assessments than software which is only required to recognize hazardous conditions and notify a human operator to take necessary safety actions. Human operators must then have redundant sources of data independent of software, and can detect and correctly react to misleading software data before a hazard can occur.

  Fatal accidents have occurred involving poorly designed human computer interfaces, such as the *Therac-25 X-ray machine* [2]. In cases where an operator relies only on software monitoring of critical functions, then a complete safety effort is required. (e.g., might require monitoring via two or more separate CPUs and resident software with voting logic.).

- **Complexity**

  *The complexity of the software system. Greater complexity increases the chances of errors.*

  The number of safety related software requirements for hazards control increases software complexity. Some rough measures of complexity include the number of subsystems controlled by software and the number of interfaces between software/hardware, software/user and software/software subsystems. Interacting, parallel executing processes also increase complexity. Note that quantifying system complexity can only be done when a high level of design maturity exists (i.e., detail design or coding phases). Software complexity can be estimated based on the number and types of logical operations it performs. Several automated programs exist to help determine software's complexity. These should be used if possible; however, the results should be used as guidelines only.

- **Timing criticality**

  *The timing criticality of hazardous control actions.*

  A system with software control of hazards that have fast reaction times or a system that has a slow response time, will require more of a software safety assurance effort. For example, spacecraft that travel beyond Earth orbit have a turnaround time spent notifying a ground human operator of a possible hazard and waiting for commands on how to proceed that may exceed the time it takes for the hazard to occur.

If the software does not meet any of the above criteria, then it is probably not safety critical. Having determined the software is safety critical, the next questions are "how critical?" and "what are the risks?".

### 3.2.2 Categorize Safety Critical Software Subsystems

Once an initial determination about safety criticality has been made, further analyses (such as the FMEA or FTA) will help to discover, confirm, or diminish the criticality rating. As the design progresses, it is essential to scope the software effort up front based on the initial analysis, then adjust as needed, as more information becomes available.

Analysis is an interactive process. The first pass, the PHA or initial Software Hazard Analysis, shows where to start focusing attention. Each successive analysis, requirements, design, code, indicates where to focus both development and further analysis.

#### 3.2.2.1 Software Control Categories

Categorizing software that controls safety critical functions in the system is based on the degree of control the software exercises over the functions. Software that can cause a hazard if it malfunctions is included in the category of high risk software. Software which is required to either recognize hazardous conditions and implement automatic safety control actions, or which is required to provide a safety critical service, or to inhibit a hazardous event, will require more software safety resources and detailed assessments.

Software which is required to recognize hazardous conditions and notify a human operator to take necessary safety actions will not require as rigorous a safety effort as the former examples. This example assumes that the human operator has redundant sources of data, independent of software, and can detect and correctly react to misleading software data before a hazard can occur. Adequate hardware safety features to prevent hazards will also move the software from "high risk" to lower risk, and require a less rigorous safety program.

In cases where an operator relies only on software monitoring of critical functions, complete safety effort is required.

A reference source of definitions for software control categories is from an older version of MIL-STD-882, specifically revision C. MIL-STD-882C [4] has been replaced by MIL-STD-882D, which does not reference the software categories. MIL-STD-882C categorized software according to their degree of control of the system, as described below:

IA. Software exercises autonomous control over potentially hazardous hardware systems, subsystems or components without the possibility of intervention to preclude the occurrence of a hazard. Failure of the software, or a failure to prevent an event, leads directly to a hazard's occurrence.

IIA. Software exercises control over potentially hazardous hardware systems, subsystems, or components allowing time for intervention by independent safety systems to mitigate the hazard. However, these systems by themselves are not considered adequate.

IIB. Software item displays information requiring immediate operator action to mitigate a hazard. Software failures will allow or fail to prevent the hazard's occurrence.

IIIA. Software item issues commands over potentially hazardous hardware systems, subsystems or components requiring human action to complete the control function. There are several, redundant, independent safety measures for each hazardous event.

IIIB. Software generates information of a safety critical nature used to make safety critical decisions. There are several redundant, independent safety measures for each hazardous event.

IV. Software does not control safety critical hardware systems, subsystems or components and does not provide safety critical information.

Complexity increases the possibilities of errors, an important point to consider. Increasing errors lead to the possibility of fault, which leads to failures. The following chart takes into consideration the complexity of the software when categorizing the software. The chart also relates back to the system risk index discussed in *Section 2.4.1.2 Risk Levels* and has already eliminated level 1 (prohibited) and level 5 (minimal risk). At this point the software category links the complexity of the software, the control which the software exerts on a system, and the system risk index. The links will be important in creating a Software Hazard Criticality Matrix.

**Table 3-2 Software Subsystem Categories**

| Software Category | Descriptions |
|---|---|
| **IA** (System Risk Index 2) | Partial or total autonomous control of safety critical functions by software. |
| | Complex system with multiple subsystems, interacting parallel processors, or multiple interfaces. |
| | Some or all safety-critical functions are time critical. |
| **IIA & IIB*** (System Risk Index 3) | Control of hazard but other safety systems can partially mitigate. Detects hazards, notifies human operator of need for safety actions. |
| | Moderately complex with few subsystems and/or a few interfaces, no parallel processing. |
| | Some hazard control actions may be time critical but do not exceed time needed for adequate human operator response. |
| **IIIA & III B*** (System Risk Index 4) | Several mitigating systems prevent hazard if software malfunctions. No safety critical data generated for a human operator. |
| | Simple system with only 2-3 subsystems, limited number of interfaces. |
| | Not time-critical. |

Note: System risk index number is taken from *Table 2-3 Hazard Prioritization - System Risk Index*
* A = software control of hazard. B = Software generates safety data for human operator

### *3.2.2.2 Software Hazard Criticality Matrix*

The Software Hazard Criticality Matrix is established using the hazard categories for the columns and the Software Control Categories for the rows. The matrix relates how involved the software is in controlling a hazard with how bad the hazard is. A Software Hazard Risk Index is assigned to each element of the matrix, just as System Risk Index numbers are assigned in the Hazard Prioritization - System Risk matrix.

**NOTE:** The *Software* Hazard Risk Index is **NOT** the same as the *System* Risk Index, though the two may appear similar. The difference is mainly that the System Risk Index of 1 (prohibited) has already been eliminated. The 1's that are shown in the top row of the Software Hazard Criticality Matrix (Table 3-3) are not the same as the 1's in the System Risk matrix

Unlike the System Risk Index, a low index number from the Software Hazard Criticality Matrix does not mean that a design is unacceptable. Rather, it indicates that greater resources need to be applied to the analysis and testing of the software and its interaction with the system.

**Table 3-3 Software Hazard Criticality Matrix**

| Software Control Category | Hazard Category | | | |
|---|---|---|---|---|
| | Catastrophic | Critical | Moderate | Negligible / Marginal |
| **IA** (System Risk Index 2) | 1 | 1 | 3 | 5 |
| **IIA & IIB** (System Risk Index 3) | 1 | 2 | 4 | 5 |
| **IIIA & IIIB** (System Risk Index 4) | 2 | 3 | 5 | 5 |
| **IV** (System Risk Index 5) | 3 | 4 | 5 | 5 |

Note: System risk index number is taken from *Table 2-3 Hazard Prioritization - System Risk Index*

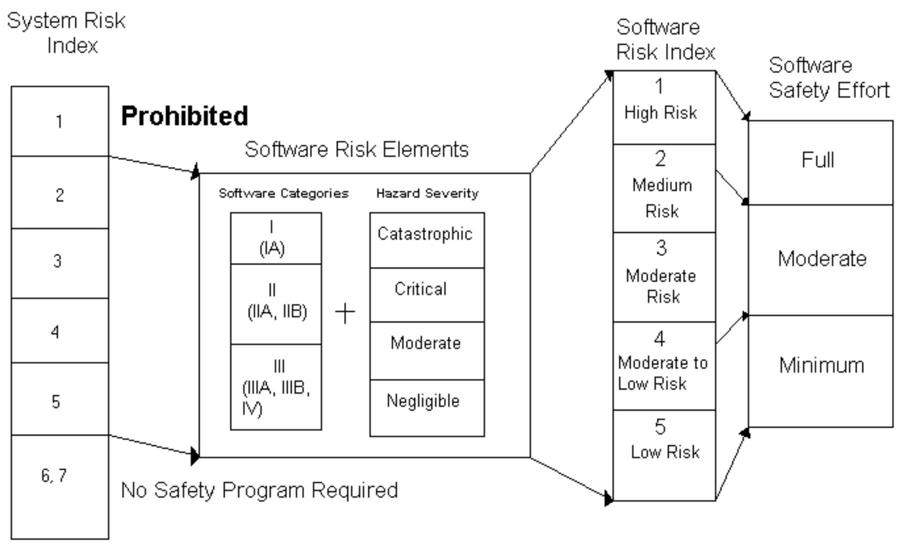
The interpretation of the Software Hazard Risk Index is given in Table 3-4. The level of risk relates directly to the amount of analysis and testing that should be applied to the software.

**Table 3-4 Software Hazard Risk Index**

| Software Hazard Risk Index | Suggested Criteria |
|---|---|
| **1** | High Risk: significant analysis and testing resources |
| **2** | Medium Risk: requirements and design analysis and in-depth testing required |
| **3-4** | Moderate Risk: high level analysis and testing acceptable with management approval |
| **5** | Low Risk: acceptable |

Figure 3-2 shows the relationships among the various risk indices and software criteria. The System Risk Index feeds into the Software Risk Index, modified by the software categories. The modification relates to how much control the software has over the hazard, either potentially causing it or in controlling/mitigating the hazard. Note that the Software Risk Index relates to only a subset of the System Risk Index, because the riskiest level (lowest System Index number) is prohibited, and the levels with the least system risk do not require a safety effort.

**Figure 3-2 Relationship of Risk Indices**



### 3.2.3 Determine the Development Effort and Oversight Required

#### *3.2.3.1 Determine Extent of Effort*

The level of required software safety effort for a system shown in *Table 3-5 Required Software Safety Effort*, is determined by its Software Hazard Risk Index as shown in Table 3-3. The mapping is essentially: Index 1 = full effort, 2 and 3 = moderate effort, and 4/5 = minimum effort. However, if your risk index is a two, consider whether you are a "high" two (closer to level one – more risk). If so, your safety effort should be the full safety effort, or somewhere between full and moderate. Also, if your Risk Index is a "high" 4, then the safety effort falls into the moderate category.

Another difference between the tables is with category IV software, which does not participate in any hazardous functions. Normally, no safety effort would be needed for such software. However, with a catastrophic and critical hazards, non-safety critical software should be evaluated for possible failures and unexpected behavior that could lead to the hazard.

Further definition of what is meant by **Full**, **Moderate**, and **Minimum** software safety effort will be discussed in *Section 3.2.3.3.*

**Table 3-5 Required Software Safety Effort**

| Software Category | Hazard Severity Level from Section 2.3.1.2 | | | |
|---|---|---|---|---|
| See Table 3-3 | Catastrophic | Critical | Moderate | Negligible / Marginal |
| IA (Software Risk Index 1) | Full | Full | Moderate | Minimum |
| IIA & IIB (Software Risk Index 2/3) | Full | Moderate | Minimum | Minimum |
| IIIA & IIIB (Software Risk Index 4/5) | Moderate | Moderate | Minimum | Minimum |
| IV Software does not directly control hazardous operations. | Minimum | Minimum | None | None |

**WARNING: Requirements are subject to change as the system design progresses!** Often items that were assigned to hardware are determined to be better (or more cheaply) developed in software. Some of those items may be safety critical. As system elements are redistributed, it is **vital** to revisit the software safety effort determination. If the new requirements lead to software controlling hazardous hardware, then more effort needs to be applied to the software safety program.

### 3.2.3.2 *Oversight Required*

The level of software quality assurance and independent oversight required for safety assurance depends on the system risk index as follows:

**Table 3-6 Degree of Oversight vs. System Risk**

| Software Risk Index | System Risk Index | Degree of Oversight |
|---|---|---|
| | 1 | Not applicable (Prohibited) |
| 1 | 2 | Fully independent IV & V[1] organization, as well as in-house SA |
| 2 | 3 | In house SA organization; Possible software IA[1] |
| 3 | 4 | In house SA organization |
| 4,5 | 5-7 | Minimal in house Software Assurance (SA) |

[1] NASA NPG 8730 "Software Independent Verification and Validation (IV&V) Management" details the criteria for determining if a project requires IV&V or Independent Assessment (IA). This NPG should be followed by all NASA projects when establishing the level of IV&V or IA required.

The level of oversight is for safety purposes, not for mission success. Oversight for mission success purposes can be greater or less than that required for safety.

A full-scale software development effort is typically performed on a safety critical flight system, (e.g. a manned space vehicle, or high value one-of-a-kind spacecraft or aircraft, critical ground control systems, critical facilities, critical ground support equipment, unmanned payloads on expendable vehicles, etc.). Other types of aerospace systems and equipment often utilize less rigorous development programs, such as non-critical ground control systems, non-critical facilities, non-critical ground support equipment, non-critical unmanned payloads on expendable vehicles, etc. In those cases, subsets of the milestone reviews and software safety development and analysis tasks can be used.

#### 3.2.3.3 Tailoring the Effort

Once the scope of the software safety effort has been determined, it is time to tailor it to a given project or program. The safety activities should be sufficient to match the software development effort and yet ensure that the overall system will be safe.

The scope of the software development and software safety effort is dependent on risk. Software safety tasks related to the life-cycle phases and milestone reviews are listed in *Table 3-1* of this guidebook. The tasks that will be performed are dependent on the risks associated with the software.

If your system will include off-the-shelf software (COTS, GOTS), reused software from another project, or software developed by a contractor, refer to Section 7 Software Acquisition. This section discusses the risks of the various types of acquired software. Additional analyses and tests may need to be performed, depending on the criticality of the acquired software and the level of knowledge you have about how it was developed and tested. The level of safety effort may be higher if the COTS/reused/contracted software is safety critical itself or interacts with the safety critical software.

This guidebook describes tasks, processes, methodologies, and reporting or documentation required for a full-fledged, formal software safety program for use on a large, "software-intensive" project. A project may choose to implement all the activities described in those sections or, based upon information presented in this section, may choose to tailor the software safety program.

However, even if a project should decide it does not intend to employ software controls of safety-critical system functions, some software safety tasks may still be necessary.

At the very minimum, a project must review all pertinent specifications, designs, implementations, tests, engineering change requests, and problem/failure reports to determine if any hazards have been inadvertently introduced. Software quality assurance activities should always verify that all safety requirements can be traced to specific design features, to specific program sets/code modules, and to specific tests conducted to exercise safety control functions of software (See *Section 5.1.1 Software Safety Requirements Flow-down Analysis*).

*Sections 3.2.3.3.1* through *3.2.3.3.3* below help determine the appropriate methods for software quality assurance, software development and software safety for full, moderate, and minimum safety efforts. Ultimately, the categorization of a project's software and the range of selected activities must be negotiated and approved by project management, software development, software quality assurance, and software systems safety personnel together. This may require educating various team members about software safety as well as knowledgeable negotiators.

##### 3.2.3.3.1 "Full" Software Safety Effort

Systems and subsystems that have severe hazards which can escalate to major failures in a very short period of time require the greatest level of safety effort. Some examples of these types of systems include life support, fire detection and control, propulsion/pressure systems, power generation and conditioning systems, and pyrotechnics or ordnance systems. These systems may require a formal, rigorous program of quality and safety assurance to ensure complete coverage and analysis of all requirements, design, code, and tests. Safety analyses, S/W analyses, safety design features, and Software Assurance (SA) oversight are highly recommended.

In addition, Independent Verification and Validation (IV&V) activities may be required. Each project, regardless of their level of safety criticality, must perform an IV&V evaluation at the beginning of the project, and whenever the project changes significantly. NASA NPG 8730 describes the process and responsibilities of all parties. IV&V provides for independent evaluation of the project software, including additional analyses and tests performed by the IV&V personnel. This is in addition to any analyses and tests performed by the project SA.

##### 3.2.3.3.2 "Moderate" Software Safety Effort

Systems and subsystems which fall into this category typically have either a limited hazard potential or, if they control serious hazards, the response time for initiating hazard controls to prevent failures is long enough to allow for notification of human operators and for them to respond to the hazardous situation. Examples of these types of systems include microwave antennas, low power lasers, and shuttle cabin heaters. These systems require a rigorous program for safety assurance of software identified as safety-critical. Non-safety-critical software must be regularly monitored to ensure that it cannot compromise the safety-critical portions of the software. Some analyses to assure there are no "undiscovered" safety critical areas and may need some S/W design safety features. Some level of Software Assurance oversight is still needed to assure late design changes don't affect the safety category.

All NASA projects, regardless of their level of safety criticality, must perform an IV&V evaluation at the beginning of the project, and whenever the project changes significantly. NASA NPG 8730 describes the process and responsibilities of all parties. While a project of this level may require IV&V, it is more likely to require an Independent Assessment.

From NPG 8730, "Software independent assessment (IA) is defined as a review of and analysis of the program/project's system software development lifecycle and products. The IA differs in scope from a full IV&V program in that IV&V is applied over the lifecycle of the system whereas an IA is usually a one time review of the existing products and plans." In many ways, IA is an outside audit of the projects development process and products (documentation, code, test results, etc.).

#### 3.2.3.3.3 "Minimum" Software Safety Effort

For systems in this category, either the inherent hazard potential of a system is very low or control of the hazard is accomplished by non-software means. Failures of these types of systems are primarily reliability concerns. This category may include such things as scan platforms and systems employing hardware interlocks and inhibits. Software development in these types of systems must be monitored on a regular basis to ensure that safety is not inadvertently compromised or that features and functions are added which now make the software safety critical. A formal program of software safety is not usually necessary. Good development practices and Software Assurance are still necessary, however.

Consider implementing some of the development activities in Section 4 and the analyses in Section 5. Many of these activities provide increased reliability as well as safety, and for a minimal effort. They help assure mission success.

As for the other safety levels, all NASA projects must perform an IV&V evaluation at the beginning of the project, and whenever the project changes significantly. NASA NPG 8730 describes the process and responsibilities of all parties. A project at this level is unlikely to require either IV&V or IA.

#### 3.2.3.3.4 Match the Safety Activities to Meet the Development Effort

During the software development process the development organization will be creating many products (documents, designs, code, etc.). The safety organization, as part of this process, will monitor and perform its own analysis, inspection, and review of the generated products with a safety slant.

Prior to this, the question that should be asked is, "How much work will these activities entail?" That is, for the type of development process required, "Which safety activities are required and how much of each activity is necessary?" All of this is dependent on the amount of risk as stated in the previous tables. Whether the effort is large or small, whether it is the entire project or just a sub-system, no matter the complexity, the effort will still require planning.

The software safety activities which will need tailoring are as follows:

| | |
|---|---|
| Analysis | There are many types of analyses which can be completed during software development. Every phase of the life-cycle can be affected by increased analysis as a result of safety considerations. The analyses can range from Requirements Criticality Analysis to Software Fault Tree Analysis of the design to Formal Methods. |
| Inspections | Inspections can take place in a number of settings and with varying products (requirements to test plans). Again, the number of inspections and products is dependent on the risk related to the system. |
| Reviews | The number of formal reviews and the setting up of delta reviews can be used to give the organization more places to look at the products as they are being developed. |

| | |
|---|---|
| Verification & Validation | Verification tests that the system was built right (according to the requirements). Validation checks that the right system was built (truly meets the needs and intent of the customer). V&V primarily involve testing of the software system, though it encompasses analyses, inspections, and reviews as well. The amount of testing can be tailored, with safety critical units and subsystems receiving the majority of the test effort. Traceability from the requirements, through design and coding, and into test is an important part of V&V. |

Development and designing in safety features such as firewalls, arm-fire commanding, etc. depends on where it is best applied and needed. The degree to which each of these activities are performed is related to risk and in turn to the software safety effort table just reviewed.

## 3.3 Incorporating Software Safety into Software Development

This section provides software safety guidance for planning both development and analysis tasks during different life cycle phases. Specific details pertaining to the performance and implementation of both tasks are discussed later in *Section 4. SAFETY CRITICAL SOFTWARE DEVELOPMENT* and *Section 5. SOFTWARE SAFETY ANALYSIS*.

There are many software engineering and assurance techniques, which can be used to control software development and result in a high-quality, reliable, and safe product. This section provides lists of recommended techniques, which have been used successfully by many organizations. Software developers may employ several techniques for each development phase, based on a project's required level of software safety effort. Other techniques, which are not listed in these tables, may be used if they can be shown to produce comparable results. Ultimately, the range of selected techniques must be negotiated and approved by project management, software development, software quality assurance, and software systems safety.

*Table 3-7 Software Requirements Phase* through *Table 3-13 Software System Testing* are modifications of tables that appear from an early International Electrotechnical Committee (IEC) draft standard IEC 1508, "Software For Computers In The Application Of Industrial Safety-Related Systems" [5]. This document is currently under review by national and international representatives on the IEC to determine its acceptability as an international standard on software safety for products which contain Programmable Electronic Systems (PES's). This set of tables is a good planning guide for software safety.

These tables provide guidance on the types of assurance activities which may be performed during the life-cycle phases of safety-critical software development. For this guidebook, the Software Safety Effort level, as determined from *Table 3-5 Required Software Safety Effort*, will determine which development activities are required for a particular project.

Each of the following tables lists techniques and recommendations for use based on safety effort level for a specific software development phase or phases. The recommendations are coded as:

| Recommendations Codes | |
|---|---|
| O | Mandatory |
| ✓✓ | Highly Recommended |
| ✓ | Recommended |
| 💣 | Not Recommended |

Most of the "Not Recommended" entries result from consideration of time and cost in relation to the required level of effort and the expected benefits. A mixture of entries marked as "Recommended" may be performed if extra assurance of safety or mission success is desired. "Highly Recommended" entries should receive serious consideration for inclusion in system development. If not included, it should be shown that safety is not compromised. In some cases the tables in this guidebook take a more conservative view of applicability of the techniques than the original IEC tables.

Each project and each organization is different. The final list of techniques to be used on any project should be developed jointly by negotiations between project management and safety assurance. Software developers should be included as well, since they will be the ones who must follow the agreed upon plan.

All the following tables, *Table 3-7 Software Requirements Phase* through *Table 3-13 Software System Testing*, list software development, safety and assurance activities which should be implemented in the stated phases of development. Appendix E contains forms that can be used to create lists of activities to be completed for each phase.

Table 3-7 Software Requirements Phase

| Technique | Safety Effort Level | | |
|---|---|---|---|
| | MIN | MOD | FULL |
| 2.4 Preliminary Hazard Analysis (PHA) | ✮ | ✮ | O |
| 5.1.1 Software Safety Requirements Flow-down Analysis | ✓ | ✓✓ | ✓✓ |
| 5.1.1.1 Checklists and cross references | ✓✓ | ✓✓ | ✓✓ |
| 5.1.2 Requirements Criticality Analysis | 💣 | ✓ | ✓✓ |
| 4.2.2 Generic Software Safety Requirements | ✓ | ✓✓ | ✮ |
| 5.1.3 Specification Analysis | 💣 | ✓ | ✓✓ |
| 4.2.3 Formal Methods - Specification Development | 💣 | ✓✓ | ✓✓ |
| 4.2.5 Formal Inspections of Specifications | ✓ | ✮ | ✮ |
| 5.1.5 Timing, Throughput And Sizing Analysis | ✓✓ | ✮ | O |
| 5.1.6 Software Fault Tree Analysis | ✓ | ✓✓ | ✮ |

Table 3-8 Software Architectural Design Phase

| Technique | Safety Effort Level | | |
|---|---|---|---|
| | MIN | MOD | FULL |
| 4.3.3 COTS and software reuse examination<br>7.1 Off-the-Shelf Software | ✓✓ | ✮ | O |
| 4.3.4 Selection of programming language, environment, tools, and operating system | ✓✓ | ✮ | O |
| 4.3.5 Coding Standards | ✮ | ✮ | ✮ |
| 5.2.1 Update Criticality Analysis | 💣 | ✓ | ✓✓ |
| 5.2.2 Conduct Hazard Risk Assessment | ✓ | ✓✓ | ✮ |
| 5.2.3 Analyze Architectural Design | ✓✓ | ✮ | ✮ |
| 5.2.4.1 Interdependence Analysis | ✓ | ✓✓ | ✮ |
| 5.2.4.2 Independence Analysis | ✓✓ | ✮ | ✮ |
| 5.2.5 Update Timing/Throughput/Sizing Analysis | ✓✓ | ✮ | O |
| 5.2.6 Update Software Fault Tree Analysis | ✓ | ✓✓ | ✮ |
| 5.2.7 Formal Inspections of Architectural Design Products | ✓ | ✓✓ | ✮ |
| 5.2.8 Formal Methods and Model Checking | 💣 | ✓ | ✓✓ |

**Table 3-9 Software Detailed Design Phase**

| Technique | Safety Effort Level | | |
|---|---|---|---|
| | MIN | MOD | FULL |
| 4.2.4 Model Checking | 💣 | ✓ | ✓✓ |
| 5.3.1 Data Logic Analysis | 💣 | ✓ | ✯ |
| 5.3.2 Design Data Analysis | ✓ | ✓✓ | ✯ |
| 5.3.3 Design Interface Analysis | ✓ | ✓✓ | ✯ |
| 5.3.4 Design Constraint Analysis | ✓ | ✓✓ | ✯ |
| 5.3.5 Design Functional Analysis | ✓ | ✓✓ | ✯ |
| 5.3.6 Software Element Analysis | ✓ | ✓✓ | ✯ |
| 5.3.7 Rate Monotonic Analysis | 💣 | ✓ | ✓✓ |
| 5.3.8 Dynamic Flowgraph Analysis | 💣 | ✓ | ✓✓ |
| 5.3.9 Markov Modeling | 💣 | ✓ | ✓✓ |
| 5.3.10 Measurement of Complexity | ✓✓ | ✓✓ | ✯ |
| 5.3.11 Selection of Programming languages | ✓ | ✓✓ | ✯ |
| 5.3.12 Formal Methods and Model Checking | 💣 | ✓ | ✓✓ |
| 5.3.13 Requirements State Machines | ✓ | ✓✓ | ✯ |
| 5.3.14 Formal Inspections of Detailed Design Products | ✓ | ✯ | ✯ |
| 5.3.15 Software Failure Modes and Effects Analysis | 💣 | ✓ | ✓✓ |
| 5.3.16 Updates to Previous Analyses (SFTA, Timing, Criticality, etc.) | ✓✓ | ✓✓ | ✯ |

**Table 3-10 Software Implementation Phase**

| Technique | Safety Effort Level | | |
|---|---|---|---|
| | MIN | MOD | FULL |
| 4.5.1 Coding Checklists | ✯ | ✯ | ✯ |
| 4.5.2 Defensive Programming | ✓✓ | ✯ | ✯ |
| 4.5.3 Refactoring | ✓ | ✓✓ | ✓✓ |
| 5.4.1 Code Logic Analysis | ✓ | ✓✓ | ✯ |
| 5.4.2 Code Data Analysis | ✓ | ✓✓ | ✯ |
| 5.4.3 Code Interface Analysis | ✓ | ✓✓ | ✯ |
| 5.4.4 Update Measurement of Complexity | ✓ | ✓✓ | ✯ |
| 5.4.5 Update Design Constraint Analysis | ✓ | ✓✓ | ✯ |
| 5.4.6 Formal Code Inspections, Checklists, and Coding Standards | ✓✓ | ✯ | ✯ |
| 5.4.7 Formal Methods | 💣 | ✓✓ | ✓✓ |
| 5.4.8 Unused Code Analysis | ✓ | ✓✓ | ✯ |
| 5.4.9 Interrupt Analysis | ✓ | ✓✓ | ✯ |
| 5.4.10 Final Timing, Throughput, and Sizing Analysis | ✓✓ | ✯ | O |
| 5.4.11 Program Slicing | ✓ | ✓✓ | ✓✓ |
| 5.4.12 Update Software Failure Modes and Effects Analysis | 💣 | ✓ | ✓✓ |

**Table 3-11 Software Testing Phase**

| Technique | Safety Effort Level | | |
|---|---|---|---|
| | MIN | MOD | FULL |
| 4.5.4 Unit Level Testing | ✓✓ | ✯ | ✯ |
| 4.6.3 Integration Testing | ✓ | ✓✓ | ✯ |
| 4.6.5 System & Functional Testing | ✯ | ✯ | ✯ |
| 4.6.6 Regression Testing | ✯ | ✯ | ✯ |
| 4.6.7 Software Safety Testing | ✯ | ✯ | ✯ |
| 7.1.4 OTS Analyses and Test | ✓ | ✓✓ | ✯ |
| 5.5.1 Test Coverage Analysis | ✓ | ✓✓ | ✯ |
| 5.5.2 Formal Inspections of Test Plan and Procedures | ✓ | ✓✓ | ✯ |
| 5.5.3 Reliability Modeling | 💣 | ✓ | ✓✓ |
| 5.5.4 Checklists of Tests | ✓ | ✓✓ | ✓✓ |
| 5.5.5 Test Results Analysis | ✯ | ✯ | ✯ |
| 5.5.6 Independent Verification and Validation | 💣 | 💣 | ✓✓ |

**Table 3-12 Dynamic Testing (Unit or Integration Level)**

| Technique | Safety Effort Level | | |
|---|---|---|---|
| | **MIN** | **MOD** | **FULL** |
| Typical sets of sensor inputs | ✓✓ | ✯ | ✯ |
| Test specific functions | ✓✓ | ✯ | ✯ |
| Volumetric and statistical tests | ✓ | ✓✓ | ✓✓ |
| Test extreme values of inputs | ✓ | ✯ | ✯ |
| Test all modes of each sensor | ✓ | ✯ | ✯ |
| Every statement executed once | ✓✓ | ✯ | ✯ |
| Every branch tested at least once | ✓✓ | ✯ | ✯ |
| Every predicate term tested | ✓ | ✓✓ | ✯ |
| Every loop executed 0, 1, many, max-1, max, max+1 times | ✓ | ✯ | ✯ |
| Every path executed | ✓ | ✓✓ | ✯ |
| Every assignment to memory tested | 💣 | ✓✓ | ✓✓ |
| Every reference to memory tested | 💣 | ✓✓ | ✓✓ |
| All mappings from inputs checked | 💣 | ✓✓ | ✓✓ |
| All timing constraints verified | ✓ | ✯ | ✯ |
| Test worst case interrupt sequences | ✓ | ✓ | ✯ |
| Test significant chains of interrupts | ✓ | ✓ | ✯ |
| Test Positioning of data in I/O space | ✓✓ | ✯ | ✯ |
| Check accuracy of arithmetic | 💣 | ✓✓ | ✯ |
| All modules executed at least once | ✯ | ✯ | ✯ |
| All invocations of modules tested | ✓✓ | ✯ | ✯ |

**Table 3-13 Software System Testing**

| Technique | Safety Effort Level | | |
|---|---|---|---|
| | MIN | MOD | FULL |
| Simulation (Test Environment) | ✓ | ✓✓ | ✯ |
| Load Testing | ✓✓ | ✯ | ✯ |
| Stress Testing | ✓✓ | ✯ | ✯ |
| Boundary Value Tests | ✓ | ✓✓ | ✯ |
| Test Coverage Analysis | ✓ | ✓✓ | ✯ |
| Functional Testing | ✯ | ✯ | ✯ |
| Performance Monitoring | ✓ | ✓✓ | ✯ |
| Disaster Testing | ✓ | ✓✓ | ✓✓ |
| Resistance to Failure Testing | ✓ | ✓✓ | ✯ |
| “Red Team” Testing | 💣 | ✓ | ✓✓ |
| Formal Progress Reviews | ✓ | ✯ | ✯ |
| Reliability Modeling | 💣 | ✓✓ | ✓✓ |
| Checklists of Tests | ✓ | ✓✓ | ✓✓ |

# 4. 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 support the cardinal safety rule and its corollary that no single event or action shall be allowed to initiate a potentially hazardous event, and that 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, which should be incorporated into the software development phases of project development. The software safety information, which should be included in the documents produced during these phases, is also discussed.

If NASA standards or guidelines exist which define the format and/or content of a specific document, they are referenced and should be followed. 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.

## *4.1 Software Concept and Initiation Phase*

For most NASA projects this life-cycle 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 which 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:

**Table 4-1 Software Safety Documentation**

| 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, flow-down 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. |

## 4.2 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:

1. To develop complete and correct requirements and correct code
2. To develop fault-tolerant designs, which will detect and compensate for software faults "on the fly".

   NOTE: (2) is required because (1) is usually impossible.

This section of the guidebook describes safety 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 hazards 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 (see *Section 2.4 PHA*). Also, system safety flows requirements to systems engineering. The systems engineering group and the software development group have a responsibility to coordinate and negotiate requirements flow-down to be consistent with the software safety requirements flow-down.

The software safety organization should flow requirements into the following documents:

- Software Requirements Document (SRD)
- Software Interface Specification (SIS) or Interface 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.

### 4.2.1 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 requirements are 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 4.2.2 Generic Software Safety Requirements*. Specific software safety requirements are system unique functional capabilities or constraints that are identified in three ways:

| | |
|---|---|
| Method 1 | Through top down analysis of system design requirements and specifications:<br><br>The system requirements may identify system hazards up-front, and specify which system functions are safety critical or a Fault Tree Analysis may be completed to identify safety critical functions. The software safety organization participates or leads the mapping of these requirements to software. |
| Method 2 | From the Preliminary Hazard Analysis (PHA):<br><br>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. |
| Method 3 | Through bottom up analysis of design data, (e.g. flow diagrams, Failure Mode Effects and Criticality Analysis (FMECA) etc.)<br><br>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 are linked to or interact with software. |

#### 4.2.1.1 *Safety Requirements Flow-down*

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.

*Section 5.1.1 Software Safety Requirements Flow-down Analysis*, verifies that the safety requirements have been properly flowed into the specifications.

Benefit-to-Cost Rating: **HIGH**

### 4.2.2 Generic Software Safety Requirements

Similar processors/platforms and/or software can suffer from similar or identical design problems. Generic software safety requirements are derived from sets of requirements and best practices used in different programs and environments to solve common software safety 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 and may have to be tailored from project to project.

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 sources for generic requirement is shown below:

1. NSTS 19943 Command Requirements and Guidelines for NSTS Customers
2. STANAG 4404 (Draft) NATO Standardization Agreement (STANAG) Safety Design Requirements and Guidelines for Munition Related Safety Critical Computing Systems
3. WSMCR 127-1 Range Safety Requirements - Western Space and Missile Center, Attachment-3 Software System Design Requirements. This document is being replaced by EWRR (Eastern and Western Range Regulation) 127-1, Section 3.16.4 Safety Critical Computing System Software Design Requirements.
4. AFISC SSH 1-1 System Safety Handbook - Software System Safety, Headquarters Air Force Inspection and Safety Center.
5. EIA Bulletin SEB6-A System Safety Engineering in Software Development (Electrical Industries Association)
6. Underwriters Laboratory - UL 1998 Standard for Safety - Safety-Related Software, January 4th, 1994
7. NUREG/CR-6263 MTR 94W0000114 High Integrity Software for Nuclear Power Plants, The MITRE Corporation, for the U.S. Nuclear Regulatory Commission.

Appendix E has a listing of Marshall Space Flight Center (MSFC) identified generic software safety requirements. They are provided in a checklist format.

Benefit-to-Cost Rating: **HIGH**

#### *4.2.2.1 Fault and Failure Tolerance/Independence*

Most NASA space systems employ failure tolerance (as opposed to fault tolerance) to achieve an acceptable degree of safety. This is primarily achieved via hardware, but software is also important, because improper software design can defeat the hardware failure tolerance and vice versa.

Not all faults lead to a failure, however, every failure results from one or more faults. A fault is an error that does not affect the functionality of the system, such as bad data from either input, calculations, or output, an unknown command, or a command or data coming at an unknown time. If properly designed, the software, or system, can respond to “glitches” by detecting these errors and correcting them, intelligently. This would include checking input and output data by possibly doing limit checking and setting the value to a known safe value, or requesting and/or waiting for the next data point. For I/O, have CRC checks and handshaking so that garbled or unrecognized messages could be detected and either dropped or request retransmission of the message.

Occasional bad I/O, data or commands should not be considered failures, unless there are too many of them and the system can not handle them. One or more intelligent fault collection routines should be part of the program to track, and possibly log, the number and type of errors. These collection routines would then either handle the caution and warning and/or recovery for the software system, or each collection routine could raise a flag to a higher level of control when the number of faults over time or the combination of fault types is programmed to determine that a system failure is imminent. With faults, the system should continue to operate normally.

A failure tolerant design detects a failure and puts the software and/or system into a changed operating state, either by switching to backup software or hardware (i.e. s/w routine, program, CPU, secondary sensor input or valve cut-off, etc.) or by reducing the functionality of the system but continuing to operate.

The question arises whether a system is to be built fault or failure tolerant or both. If the system, including software, is built to handle most probable, and some less probable but hazardous faults, it may be able to preclude many possible failure scenarios. Taking care of problems while they are still faults can help prevent the software, or the system, from going into failure. The complaint with building in fault tolerance is that it requires multiple checks and monitoring at very low levels. If major failures can be detected, isolated, stopped or recovered from, it is presumed that this would require less work and be simpler than fault tolerance.

For safety critical systems, it is best to design in both fault and failure tolerance. The fault tolerance keeps most of the minor errors from propagating into failures. Failures must still be detected and dealt with, whether as a result of fault collection/monitoring routines or by direct failure detection routines and/or hardware. In this guidebook, both fault and failure tolerance are discussed. The proper blending of both to meet the requirements of your particular system must be determined by the software designers and the safety engineers.

- **Must Work Functions (MWFs)**

  MWF's achieve failure tolerance through independent parallel redundancy. For parallel redundancy to be truly independent there must be dissimilar software in each parallel path. Software can sometimes be considered "dissimilar" if N-Version programming is properly applied, see *Section 4.3 Architectural Design Phase*.

- **Must Not Work Functions (MNWF's)**

  MNWF's achieve failure tolerance through independent multiple series inhibits. For series inhibits to be considered independent they must be generally controlled by different processors containing dissimilar software.

  In both cases, software must be specified to preserve the hardware failure tolerance via proper allocation amongst hardware units.

- **Fault/Failure Detection, Isolation and Recovery (FDIR)**

  FDIR is a problematic design area, where improper design can result in system false alarms, "bogus" system failures, or failure to detect important safety critical system failures.

  FDIR for the NASA Space Station has included two approaches:

  1. **Shadowing:** Fail-safe active hazard detection and safing can be used where a higher tier processor can monitor a lower tier processor and shut down the lower processor in the event that pre-defined allowable conditions are violated. Higher tier processors can emulate the application running in the lower tier, and compare predicted (expected) parameter values to actual measured values. This technique is sometimes called "shadowing" or convergence testing.

  2. **Built-in Test( BIT):** Sometimes FDIR can be based on self-test (BIT) of lower tier processors where lower level units test themselves, and report their good/bad status to a higher processor. The higher processor switches out units reporting a failed or bad status.

If too many faults or very serious failures occur, it may be necessary for the system to shut itself down in an orderly, safe manner. (For example, in the event of a power outage the system might continue for a short period of time on limited battery power during which period the software should commence an orderly shutdown to a safe system state).

Software responses to off-nominal scenarios should address safety considerations, and be appropriate to the situation. Complete system shutdown may not be appropriate in many cases.

How to achieve independence and other failure tolerance development methods are discussed more in *Section 4.3 Architectural Design Phase*.

Benefit-to-Cost Rating: **MEDIUM**

#### *4.2.2.2 Hazardous Commands*

Appendix-A Glossary of Terms defines a "hazardous command". Commands can be internal to a software set (e.g., from one module to another) or external, crossing an interface to/from hardware or a human operator. Longer command paths increase the probability of an undesired or incorrect command response due to noise on the communications channel, link outages, equipment malfunctions, or (especially) human error.

Reference [26] NSTS 1700.7B section 218 defines "hazardous command" as "...those that can remove an inhibit to a hazardous function, or activate an unpowered payload system". It continues to say "Failure modes associated with payload flight and ground operations including hardware, software, and procedures used in commanding from payload operations control centers (POCC's) and other ground equipment must be considered in the safety assessment to determine compliance with the (failure tolerance) requirements. NSTS 19943 treats the subject of hazardous commanding and presents the guidelines by which it will be assessed."

NSTS 1700.7B section 218 focuses on remote commanding of hazardous functions, but the principles can and should be generally applied. Both NSTS 19943 and EWRR 127-1 (Paragraph 3.16.7 b) recommend and require respectively, two-step commanding. EWRR 127-1 states "Two or more unique operator actions shall be required to initiate any potentially hazardous function or sequence of functions. The actions shall be designed to minimize the potential for inadvertent actuation". Note that two-step commanding is in addition to any hardware (or software) failure tolerance requirements, and is neither necessary nor sufficient to meet failure tolerance requirements. A two-step command does not constitute an inhibit. (See Glossary Appendix A for definition of inhibit.)

Software interlocks or preconditions can be used to disable certain commands during particular mission phases or operational modes. However, provision should be made to provide access to (i.e. enable) all commands in the event of unexpected emergency situations. Emergency command access is generally required by flight crews.

For example, when Apollo 13 experienced major problems, the nominal Lunar Module power up sequence timeline could not be completed before the Command Module battery power expired. A different (shorter) sequence was improvised.

Benefit-to-Cost Rating: **HIGH**

#### *4.2.2.3 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 in *Section 5.1.5 Timing, Sizing and Throughput Analysis*.

- **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**

  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**

  Control system design can define timing requirements. The design is 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.[1]

- **Sampling rates**

  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. A factor of at least ten above the system characteristic frequency is customary. [1]

- **Dynamic memory allocation**

  Dynamic memory allocation requires several variety of resources be available and adequate. The amount of actual memory (RAM) available, whether virtual memory (disk space) is used, how much memory the software (programs and operating system) uses statically, and how much is dynamically allocated are all factors in whether a dynamic allocation will fail or succeed. Several factors may not be known in detail, and worst-case values should be used.

  How the software deals with failed dynamic allocation should be considered. What a particular language or compiler does in such a situation should be thoroughly researched. If an exception or signal is generated, it should be handled by the application software. Allowing a default similar to the MS-DOS "abort, retry, fail" is a very bad idea for safety critical software.

  Critical memory blocks must be identified and protected from inadvertent corruption or deletion. Since the dynamic allocation mechanism is usually "unknown" (defined by the compiler vendor), methods must be used to detect or prevent the allocation of any of the critical areas. Processors with Memory Management Units (MMU) provide one mechanism. Checking the address range returned by the dynamic allocation routine against the critical memory addresses will work in systems that use physical (RAM) addresses or logical memory addresses. Care must be taken that logical and physical addresses are not compared to each other. CRC values or error-correcting codes are software ways to detect and/or correct critical data that may be accidentally overwritten.

- **Memory Checking**

  Testing of random access memory (RAM) can be a part of BIT/self-test and is usually done on power up of a system to verify that all memory addresses are available, and that the RAM is functioning properly. Memory tests may be done periodically to check for problems that may occur, due to a single event upset or other hardware (RAM) problems.

  Memory utilization checks may be used 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 [10]. 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)).

Benefit-to-Cost Rating: **HIGH**

### 4.2.3 Formal Methods - Specification Development

Reference [25] states: "Formal Methods (FM) consists of a set of techniques and tools based on mathematical modeling and formal logic that are used to specify and verify requirements and designs for computer systems and software."

Formal Methods have not gained a wide acceptance among all industries, mostly due to the difficulty of the formal proofs. A considerable learning curve must be surmounted for newcomers, which can be expensive. Once this hurdle is surmounted successfully, some users find that it can reduce overall development life-cycle cost by eliminating many costly defects prior to coding. In addition, many tools are now available to aid in using Formal Methods.

Software and system requirements are usually written in "human-readable" language. This can lead to ambiguity, when a statement that is clear to one person is interpreted differently by another. To avoid this ambiguity, requirements can be written in a formal, mathematical language. This is the first step in applying Formal Methods.

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. 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 [31]. 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.

1) **Formal development down to object code**

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

2) **Formal development down to source code**

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

3) **Rigorous development down to source code**

   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.

4) **Structured development to requirements**

   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.

5) **Structured development down to source code**

Structured development down to source code is the application of the structured analysis/structured design method proposed by DeMarco [32]. 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.

**6) Ad hoc**

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

The methodology described in this guidebook is that of level 3, rigorous development down to source code.

Detailed descriptions of Formal methods are given in the NASA Formal Methods Guidebook [25]. In addition, the following publications are recommended reading as primers in Formal Methods: Rushby [29], Miller, et al [30], and Butler, et al [31]. Anthony Hall [39] gives "Seven Myths of Formal Methods", and discusses using formal specification of requirements without formal proofs in a real-world development environment. Richard Kemmerer [40] shows how to integrate formal methods with the development process.

Benefit-to-Cost Rating: **MEDIUM**

The following descriptions of Formal Methods are taken from the NASA Langley FM Group internet World Wide Web home page:

#### *4.2.3.1 Why Is Formal Methods Necessary?*

A digital system may fail as a result of either physical component failure, or design errors. The validation of an ultra-reliable system must deal with both of these potential sources of error.

Well known techniques exist for handling physical component failure; these techniques use redundancy and voting. The reliability assessment problem in the presence of physical faults is based upon Markov modeling techniques and is well understood.

The design error problem is a much greater threat. Unfortunately, no scientifically justifiable defense against this threat is currently used in practice. There are 3 basic strategies that are advocated for dealing with the design error:

- Testing (lots of it)
- Design Diversity (i.e. software fault-tolerance: N-version programming, recovery blocks, etc.)
- Fault/Failure Avoidance (i.e. formal specification/verification, automatic program synthesis, reusable modules)

The problem with life testing is that in order to measure ultra reliability one must test for exorbitant amounts of time. For example, to measure a $10^{-9}$ probability of failure for a 1 hour mission one must test for more than 114,000 years.

Many advocate design diversity as a means to overcome the limitations of testing. The basic idea is to use separate design/implementation teams to produce multiple versions from the same specification. Then, non-exact threshold voters are used to mask the effect of a design error in one of the versions. The hope is that the design flaws will manifest errors independently or nearly so.

By assuming independence one can obtain ultra-reliable-level estimates of reliability even though the individual versions have failure rates on the order of $10^{-4}$. Unfortunately, the independence assumption has been rejected at the 99% confidence level in several experiments for low reliability software.

Furthermore, the independence assumption cannot ever be validated for high reliability software because of the exorbitant test times required. If one cannot assume independence then one must measure correlation's. This is infeasible as well, since it requires as much testing time as life-testing the system because the correlation's must be in the ultra-reliable region in order for the system to be ultra-reliable. Therefore, it is not possible, within feasible amounts of testing time, to establish that design diversity achieves ultra-reliability. Consequently, design diversity can create an illusion of ultra-reliability without actually providing it.

It is felt that formal methods currently offers the only intellectually defensible method for handling the design fault problem. Because the often quoted $1 - 10^{-9}$ reliability is well beyond the range of quantification, there is no choice but to develop life-critical systems in the most rigorous manner available to us, which is the use of formal methods.

#### *4.2.3.2 What Is Formal Methods?*

Traditional engineering disciplines rely heavily on mathematical models and calculation to make judgments about designs. For example, aeronautical engineers make extensive use of computational fluid dynamics (CFD) to calculate and predict how particular airframe designs will behave in flight. We use the term formal methods to refer to the variety of mathematical modeling techniques that are applicable to computer system (software and hardware) design. That is, formal methods is the applied mathematics engineering and, when properly applied, can serve a role in computer system design analogous to the role CFD serves in aeronautical design.

Formal methods may be used to specify and model the behavior of a system and to mathematically verify that the system design and implementation satisfy system functional and safety properties. These specifications, models, and verifications may be done using a variety of techniques and with various degrees of rigor. The following is an imperfect, but useful, taxonomy of the degrees of rigor in formal methods:

Level-1: Formal specification of all or part of the system.

Level-2: Formal specification at two or more levels of abstraction and paper and pencil proofs that the detailed specification implies the more abstract specification.

Level-3: Formal proofs checked by a mechanical theorem prover.

Level 1 represents the use of mathematical logic or a specification language that has a formal semantics to specify the system. This can be done at several levels of abstraction. For example, one level might enumerate the required abstract properties of the system, while another level describes an implementation that is algorithmic in style.

Level 2 formal methods goes beyond Level 1 by developing pencil-and-paper proofs that the more concrete levels logically imply the more abstract-property oriented levels. This is usually done in the manner illustrated below.

Level 3 is the most rigorous application of formal methods. Here one uses a semi-automatic theorem prover to make sure that all of the proofs are valid. The Level 3 process of convincing a mechanical prover is really a process of developing an argument for an ultimate skeptic who must be shown every detail.

Formal methods is not an all-or-nothing approach. The application of formal methods to only the most critical portions of a system is a pragmatic and useful strategy. Although a complete formal verification of a large complex system is impractical at this time, a great increase in confidence in the system can be obtained by the use of formal methods at key locations in the system.

### 4.2.4 Model Checking

Model checking is a form of Formal Methods that verifies finite-state systems. It is an “automatic” method, and tools exist to provide that automation (for instance: SPIN and SMV). Model checking can be applied to more than just software, and has been used to formally verify industrial systems.

The technique is especially aimed at the verification of reactive, embedded systems, i.e. systems that are in constant interaction with the environment. Model checking can be applied relatively easily at any stage of the existing software process without causing major disruptions. It has been extended to work with at least some infinite-state systems and also with real-time systems. Model checking can verify simple properties like reachability (does as system ever reach a certain state) or lack-of-deadlock (is deadlock avoided in the system), or more complex properties like safety (nothing bad ever happens) or liveness (something good eventually happens).

Benefit-to-Cost Rating: **MEDIUM**

#### *4.2.4.1 How Model Checking Works*

The first step in model checking is to describe the system in a state-based, formal way. Each model checker uses its own language for system description.

The second step is to express program flow using prepositional temporal logic. This logic deals with transitions from one state to another (stepping through the program), and what may or may not be true in each state. For instance, you can express a formula (property) that is true in some future state (eventually) or in all future states (always).

Once the system is modeled and the temporal logic is determined, algorithms are used to traverse the model defined by the system and check if the specification holds or not. Very large state-spaces can often be traversed in minutes. The technique has been applied to several complex industrial systems, ranging from hardware to communication protocols to safety critical plants and procedures.

For more details, the book “Model Checking” [37] describes the technique in detail. The website http://www.abo.fi/~johan.lilius/mc/mclinks.html contains references to current model checking research, people, tools, and projects. General information, as well as reviews of SMV and SPIN (the automated tools described below), can be found at: http://www.math.hmc.edu/~jpl/modelcheckers.html.

#### 4.2.4.2 Tools

Among the automated tools, the primary ones are SMV and SPIN. SMV is a symbolic model checker specialized on the verification of synchronous and asynchronous systems. SPIN is an on-the-fly model checker specialized on the verification of asynchronous systems.

Spin (http://netlib.bell-labs.com/netlib/spin/whatispin.html) is designed to test the specifications of concurrent (distributed) systems-- specifically communications protocols, though it applies to any concurrent system. It will find deadlocks, busy cycles, conditions that violate assertions, and race conditions. The software was developed at Bell Labs in the formal methods and verification group starting in 1980. Spin targets efficient software verification, not hardware verification. It uses a high level language to specify systems descriptions (PROMELA - PROcess MEta LAnguage). Spin has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification.

Spin also reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. It uses an "on-the-fly" approach where not all of the model must be in memory at once.

SMV (Symbolic Model Verifier) (http://www.cs.cmu.edu/~modelcheck/smv.html) comes from Carnegie Mellon University. The SMV system requires specifications to be written in the temporal logic CTL, and uses Kripke diagrams. The input language of SMV is designed to allow the description of finite state systems that range from completely synchronous to completely asynchronous, and from the detailed to the abstract. The language provides for modular hierarchical descriptions, and for the definition of reusable components. Since it is intended to describe finite state machines, the only data types in the language are finite ones - Booleans, scalars and fixed arrays. The logic CTL allows safety, liveness, fairness, and deadlock freedom to be specified syntactically.

In addition, other "academic" systems include:

- HyTech (http://www-cad.EECS.Berkeley.EDU/~tah/HyTech/)
- Kronos (http://www-verimag.imag.fr//TEMPORISE/kronos/index-english.html)
- MONA (http://www.brics.dk/mona/)
- Murphi (http://sprout.stanford.edu/dill/murphi.html)
- TREAT (http://www.cis.upenn.edu/~lee/inhye/treat.html)
- TVS (http://tvs.twi.tudelft.nl/)
- STEP (http://rodin.stanford.edu/)
- UPPAAL (http://www.docs.uu.se/docs/rtmv/uppaal/index.html)
- Verus (http://www.cs.cmu.edu/~modelcheck/verus.html)
- Vis (http://www-cad.eecs.berkeley.edu/~vis/)

Commercial programs include:

- FormalCheck (http://www.cadence.com/eda_solutions/flv_fveimc_l3_index.html)
- Time Rover (http://www.time-rover.com/TRindex.html)
- Rational Rose add-in (http://www.rationalrose.com/modelchecker/)

#### *4.2.4.3 Challenges*

The main challenge in model checking is the state explosion problem -- the fact that the number of states in the model is frequently so large that model checkers exceed the available memory and/or the available time. Several techniques are used to cope with this problem.

One type of technique is to build only a part of the state-space of the program, while still maintaining the ability to check the properties of interest. These are "partial-order techniques" (interleaving) and "abstraction techniques" (simpler system).

The "symbolic approach" is another way to overcome the problem. The idea is to *implicitly* represent the states and transitions of the system, rather than *explicitly*. Binary Decision Diagrams (BDDs) is an efficient encoding of Boolean formulas, and is the usual implicit representation. The BDD is used with the temporal formulas for the model checking. Therefore, the size of the BDD representation is the limiting factor and not the size of the explicit state representation.

"On-the-fly" techniques analyze portions of the model as it goes along, so that not all of it must be in memory at any one time.

### 4.2.5 Formal Inspections of Specifications

Formal Inspections* are structured technical reviews of a "product" of the software development life cycle, conducted for the purpose of finding and eliminating defects. The product can be any documentation, including requirements, design notes, test plans, or the actual source code. Formal Inspections differ from informal reviews or walkthroughs in that there are specified steps to be taken, and roles are assigned to individual reviewers.

Formal inspections are not formal methods! Formal inspections are a structured way to find defects in some software product, from a requirements document to the actual code. Formal methods are a mathematical way of specifying and verifying a software system. The two methods can be used together or separately.

NASA has published a standard and guidebook for implementing the Formal Inspection (FI) process, Software Formal Inspections Standard (NASA-STD-2202-93) [36] and Software Formal Inspections Guidebook (NASA-GB-A302) [25]. FI's should be performed within every major step of the software development process, including requirements specification, design, coding, and testing.

Formal Inspections have the most impact when applied *early* in the life of a project, especially the requirements specification and definition stages of a project. Impact means that the bugs are found earlier, when it's cheaper to fix them. Studies have shown that the majority of all faults/failures, including those that impinge on safety, come from missing or misunderstood requirements. Formal Inspection greatly improves the communication within a project and enhances understanding of the system while scrubbing out many of the major errors/defects.

For the FI of software requirements, the inspection team should include representatives from Systems Engineering, Operations, Software Design and Code, Software Product Assurance, Safety, and any other system function that software will control or monitor. It is very important that software safety be involved in the FI's. Each individual may review the requirements from a generic viewpoint, or they may be assigned a specific point of view (tester, programmer/designer, user, safety) from which to review the document.

It is also very helpful to have inspection checklists for each phase of development that reflect both generic and project specific criteria. The requirements discussed in this section and in Robyn R. Lutz's paper "Targeting Safety-Related Errors During Software Requirements Analysis" [42] will greatly aid in establishing this checklist. Also, the checklists provided in the NASA Software Formal Inspections Guidebook are helpful.

The method of reporting findings from FI's is described in references [25] and [36]. In addition to those formats, the software safety engineer might also find it useful to record safety related findings in the format shown in Table 4-2.

Benefit-to-Cost Rating: **HIGH**

* Formal inspections are also known as Fagan Inspections, named after John Fagan of IBM who devised the method.

**Table 4-2 Subsystem Criticality Analysis Report Form**

| | | | | |
|---|---|---|---|---|
| Document Number: CL-SPEC- 2001 | | | | |
| Document Title: Software Requirements Specification - Cosmolab Program | | | | |
| Paragraph Number / Title | Requirements(s) text excerpt | Problem / Hazard Description | Recommendations | Hazard Report Reference Number |
| 3.3 Limit Checking | Parameters listed in Table 3.3 shall be subjected to limit checking at a rate of 1 Hz. | Table only gives one set of limits for each parameter, but expected values for parameters will change from mode to mode. | During certain modes, false alarms would result because proper parameter values will exceed preset limit check values. Implement table driven limit values which can be changed during transitions from mode to mode. | CL-1;9 |

### 4.2.6 Test Planning

At the end of the specification phase, the test plan can be written. System tests can be defined that verify the functional aspects of the software under nominal conditions, as well as performance, load, stress, and other tests that verify acceptable behavior in non-standard situations.

Safety tests of the system should be designed at this time. The tests should demonstrate how the software and system meets the safety requirements in the SRD. Specify pass/fail criteria for each test. Specify any special procedures, constraints, and dependencies for implementing and running safety tests. Describe the review and reporting process for safety-critical components. List test results, problems, and liens.

## 4.3 Architectural Design Phase

The design of a program set represents the static and dynamic characteristics of the software that will meet the requirements specified in the governing SRD. Projects developing large amounts of software may elect to separate design development into separate phases, preliminary (architectural) and detailed/critical. Those with relatively small software packages may combine them into one phase.

### 4.3.1 Safety Objectives of Architectural Design

The main safety objective of the architectural design phase is to define the strategy for achieving the required level of failure tolerance in the different parts of the system. The degree of failure tolerance required can be inversely related to the degree of fault reduction used, (e.g. Formal Methods). However, even the most rigorous level of fault reduction will not prevent all faults, and some degree of failure tolerance is generally required.

Ideas, techniques, and approaches to be used during architectural design are as follows:

- **Modularity**

  During this phase the software is usually be partitioned into modules, and the number of safety critical modules should be minimized. Interfaces between critical modules should also be designed for minimum interaction (low coupling).

- **Traceability**

  Requirements previously developed must be flowed down into the architecture, and be traceable.

- **Independence/Failure Tolerance**

  Despite the difficulty to proving software independence, some NASA programs continue to use this approach. As discussed earlier in *Section 4.2.2.2 Fault and Failure Tolerance/Independence*, two types of independence are often required; first to prevent fault propagation and second to achieve failure tolerance.

  To achieve failure tolerance for safety critical MWF's and MNWF's, certain Safety Critical Computer Software Components (SCCSC's) must be independent of each other. This usually means operating on different hardware hosts.

- **MNWF**:

  For two in-series inhibits to be independent, no single failure, human mistake, event or environment may activate both inhibits. For three series inhibits to be independent, no two failures, human mistakes, events or environments (or any combination of two single items) may activate all three inhibits. Generally this means that each inhibit must be controlled by a different processor with different software (N-version programming, see below).

- **MWF**

  For two parallel strings to be independent, no single failure may disable both strings. For three parallel strings, no two failures may disable all three strings. In software, N-version programming is preferred in safety critical parallel strings, (i.e., each string is implemented using uniquely developed code).

  Depending on the design, architectural techniques may include:

  - Convergence testing
  - Majority voting
  - Fault containment regions
  - Redundant Architecture
  - N-Version programming
  - Recovery blocks
  - Resourcefulness
  - Abbott-Neuman Components
  - Self-Checks.

For non-discrete continuously varying parameters which are safety critical, a useful redundancy technique is convergence testing or “shadowing”. A higher level process emulates lower level process(es) to predict expected performances and decide if failures have occurred in the lower level processes. The higher level process implements appropriate redundancy switching when it detects a discrepancy. Alternatively the higher level process can switch to a subset or degraded functional set to perform minimal functions when insufficient redundancy remains to keep the system fully operational.

Some redundancy schemes are based on majority voting. This technique is especially useful when the criteria for diagnosing failures are complicated. (e.g. when an unsafe condition is defined by exceeding an analog value rather than simply a binary value). Majority voting requires more redundancy to achieve a given level of failure tolerance, as follows: 2 of 3 achieves single failure tolerance; 3 of 5 achieves two failure tolerance. An odd number of parallel units are required to achieve majority voting.

Fault Propagation is a cascading of a software (or hardware or human) error from one module to another. To prevent fault propagation within software, SSCSC’s must be fully independent of non-safety critical components and be able to either detect an error within itself and not allow it to be passed on or the receiving module must be able to catch and contain the error.

### 4.3.1.1 Fault Containment Regions

One approach is to establish Fault Containment Regions (FCR’s) to prevent propagation of software faults. This attempts to prevent fault propagation such as from non-critical software to SCCSC’s; from one redundant software unit to another, or from one SCCSC to another. Techniques~~known~~ such as firewalling or~~partitioning~~ “come from” checks should be used to provide sufficient isolation of FCR’s to prevent hazardous fault propagation.

FCR's are best partitioned or firewalled by hardware. Leveson (Section 3 Reference [2] ) states that "logical" firewalls can be used to isolate software modules, such as isolating an application from an operating system. To some extent this can be done using defensive programming techniques and internal software redundancy. (e.g., using authorization codes, or cryptographic keys). However, within NASA this is normally regarded as hazard mitigation, but not hazard control, because such software/logical safeguards can be defeated by hardware failures or EMI/Radiation effects.

A typical method of obtaining independence between FCR's is to host them on different/ independent hardware processors. Sometimes it is acceptable to have independent FCR's hosted on the same processor depending on the specific hardware configuration. For example, if the FCR's are stored in separate memory chips and they are not simultaneously or concurrently multitasked in the same Central Processing Unit (CPU) at the same time.

Methods of achieving independence are discussed in more detail in Reference [1], "The Computer Control of Hazardous Payloads", NASA/JSC/FDSD, 24 July 1991. FCR's are defined in reference [2], SSP 50038 Computer Based Control System Safety Requirements - International Space Station Alpha.

#### *4.3.1.2 N-Version Programming*

N-Version Programming is one method that can be used to implement fault tolerant behavior. Multiple, independent versions of the software execute simultaneously. If the answers all agree, then the process continues. If there is disagreement, then a voting method is used to determine which of the answers is correct.

In the past, some NASA policy documents have essentially stipulated the use of N-Version programming in any attempt to achieve failure tolerance. Reference [2] discusses in more detail the JSC position on N-Version programming. They recognize that the technique has limitations. Many professionals regard N-Version programming as ineffective, or even counter productive.

Efforts to implement N-Version programming should be carefully planned and managed to ensure that valid independence is achieved. In practice applications of N-Version programming on NSTS payloads are limited to small simple functions. However, the NSTS power up of the engines has N-Version programming as well.

Note that, by the NSTS 1700.7B stipulation, two processors running the same operating system are neither independent nor failure-tolerant of each other, regardless of the degree of N-Version programming used in writing the applications.

In recent years, increasing controversy has surrounded the use of N-Version programming. In particular, Knight and Leveson [13] have jointly reported results of experiments with N-Version programming, claiming the technique is largely ineffective. Within NASA, Butler and Finelli [28] have also questioned the validity of N-version programming, even calling it "counter productive". Though it has worked very effectively on some occasions, it should be evaluated carefully before being implemented.

One major problem with N-version programming is that it increases complexity, which has a direct relationship with the number of errors. In one NASA study of an experimental aircraft, all of the software problems found during testing were the result of the errors in the redundancy management system. The control software operated flawlessly! Another difficulty with n-version programming is that achieving true independence is very difficult. Even if separate teams develop the software, studies have shown that the software is still often not truly independent.

References [2] and [11] give some useful background for N-Version programming, and also for:

- Recovery blocks
- Resourcefulness
- Abbott-Neuman Components
- Self-Checks

#### *4.3.1.3 Redundant Architecture*

Redundant architecture refers to having two versions of the operational code. Unlike n-version programming, the two versions do not need to operate identically. The primary software is the high-performance version. This is the "regular" software you want to run – it meets all the required functionality and performance requirements.

However, if problems should develop in the high-performance software, particularly problems or failures that impact safety, then a "high-assurance" kernel (also called a safety kernel) is given control. The high-assurance kernel may have the same functionality as the high-performance software, or may have a more limited scope. The primary aspect is that it is *safe*. The high-assurance kernel will almost certainly be less optimized (slower, stressed more easily, lower limits on the load it can handle, etc.).

The Carnegie Mellon Software Engineering Institute (SEI) Simplex Architecture (Section 7 reference [9]) is an example of a redundant architecture.. This architecture includes the high-performance/high-assurance kernels, address-space protection mechanisms, real-time scheduling algorithms, and methods for dynamic communication among modules. This process requires using analytic redundancy to separate major functions into high-assurance kernels and high-performance subsystems.

### 4.3.2 Structured Design Techniques

It is generally agreed that 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 & Object Oriented Design (OOA & OOD) and Formal Methods (FM). 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 following discussion, "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 that 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.

Structured Analysis (DeMarco [34], Yourdon [33]) became popular in the 1980's and is still used by many. The analysis consists of interpreting the system concept (or real world) into data and control terminology, that is into data flow diagrams. The flow of data and control from bubble to data store to bubble can be very hard to track and the number of bubbles can get to be extremely large. One approach is to first define events from the outside world that require the system to react, then assign a bubble to that event, bubbles that need to interact are then connected until the system is defined. This can be rather overwhelming and so the bubbles are usually grouped into higher level bubbles. Data Dictionaries are needed to describe the data and command flows and a process specification is needed to capture the transaction/transformation information. The problems have been: 1) choosing bubbles appropriately, 2) partitioning those bubbles in a meaningful and mutually agreed upon manner, 3) the size of the documentation needed to understand the Data Flows, 4) still strongly functional in nature and thus subject to frequent change, 5) though "data" flow is emphasized, "data" modeling is not, so there is little understanding of just what the subject matter of the system is about, and 6) not only is it hard for the customer to follow how the concept is mapped into these data flows and bubbles, it has also been very hard for the designers who must shift the data flow diagram organization into a format that can be implemented.

Information Modeling, using entity-relationship diagrams, is really a forerunner for OOA. The analysis first finds objects in the problem space, describes them with attributes, adds relationships, refines them into super and sub-types and then defines associative objects. Some normalization then generally occurs. Information modeling is thought to fall short of true OOA in that, according to Peter Coad & Edward Yourdon [37], 1) services, or processing requirements, for each object are not addressed, 2) inheritance is not specifically identified, 3) poor interface structures (messaging) exists between objects, and 4) classification and assembly of the structures are not used as the predominate method for determining the system's objects.

This guidebook will present in more detail the two new most promising methods of structured analysis and design, Object-Oriented and Formal Methods (FM). OOA/OOD and FM can incorporate the best from each of the above methods and can be used effectively in conjunction with each other.

Lutz and Ampo [27] described their successful experience of using OOD combined with Formal Methods as follows:

> "For the target applications, object-oriented modeling offered several advantages as an initial step in developing formal specifications. This reduced the effort in producing an initial formal specification. We also found that the object-oriented models did not always represent the "why," of the requirements, i.e., the underlying intent or strategy of the software. In contrast, the formal specification often clearly revealed the intent of the requirements."

#### *4.3.2.1 Object Oriented Analysis and Design*

Object Oriented Design (OOD) is gaining increasing acceptance worldwide. OOD methods include those of Coad-Yourdon, Shlaer-Mellor, Rumbaugh, and Booch methods. These fall short of full Formal Methods because they generally do not include logic engines or theorem provers. But they are more widely used than Formal Methods, and a large infrastructure of tools and expertise is readily available to support practical OOD usage.

OOA/OOD is the new paradigm and is viewed by many as the best solution to most problems. Some of the advantages of modeling the real world into objects is that 1) it is thought to follow a more natural human thinking process and 2) objects, if properly chosen, are the most stable perspective of the real world problem space and can be more resilient to change as the functions/services and data & commands/messages are isolated and hidden from the overall system. For example, while over the course of the development life-cycle the number, as well as types, of functions (e.g. turn camera 1 on, download sensor data, ignite starter, fire engine 3, etc.) may change, the basic objects (e.g. cameras, sensors, starter, engines, operator, etc.) needed to create a system usually are constant. That is, while there may now be three cameras instead of two, the new Camera-3 is just an instance of the basic object 'camera'. Or while an infrared camera may now be the type needed, there is still a 'camera' and the differences in power, warm-up time, data storage may change, all that is kept isolated (hidden) from affecting the rest of the system.

OOA incorporates the principles of abstraction, information hiding, inheritance to the problem space, which are the three most "human" means of classification. These combined principles, if properly applied, establish a more modular, bounded, stable and understandable software system. These aspects of OOA should make a system created under this method more robust and less susceptible to changes, properties which help create a safer software system design.

Abstraction refers to concentrating on only certain aspects of a complex problem, system, idea or situation in order to better comprehend that portion. The perspective of the analyst focuses on similar characteristics of the system objects that are most important to them. Then, at a later time, the analyst can address other objects and their desired attributes or examine the details of an object and deal with each in more depth. Data abstraction is used by OOA to create the primary organization for thinking and specification in that the objects are first selected from a certain perspective and then each object is defined in detail. An object is defined by the attributes it has and the functions it performs on those attributes. An abstraction can be viewed, as per Shaw [38], as "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary".

Information hiding also helps manage complexity in that it allows encapsulation of requirements which might be subject to change. In addition, it helps to isolate the rest of the system from some object specific design decisions. Thus, the rest of the s/w system sees only what is absolutely necessary of the inner workings of any object.

Inheritance "defines a relationship among classes [objects], wherein one class shares the structure or behavior defined in one or more classes. Inheritance thus represents a hierarchy of abstractions, in which a subclass [object] inherits from one or more superclasses [ancestor objects]. Typically, a subclass augments or redefines the existing structure and behavior of its superclasses" [39].

Classification theory states that humans normally organize their thinking by:

- looking at an object and comparing its attributes to those experienced before (e.g. looking at a cat, humans tend to think of its size, color, temperament, etc. in relation to past experience with cats)
- distinguishing between an entire object and its component parts (e.g. a rose bush versus its roots, flowers, leaves, thorns, stems, etc.)
- classification of objects as distinct and separate groups (e.g. trees, grass, cows, cats, politicians)

In OOA, the first organization is to take the problem space and render it into objects and their attributes (abstraction). The second step of organization is into Assembly Structures, where an object and its parts are considered. The third form of organization of the problem space is into Classification Structures during which the problem space is examined for generalized and specialized instances of objects (inheritance). That is, if looking at a railway system the objects could be engines (provide power to pull cars), cars (provide storage for cargo), tracks (provide pathway for trains to follow/ride on), switches (provide direction changing), stations (places to exchange cargo), etc. Then you would look at the Assembly Structure of cars and determine what was important about their pieces parts, their wheels, floor construction, coupling mechanism, siding, etc. Finally, Classification Structure of cars could be into cattle, passenger, grain, refrigerated, and volatile liquid cars.

The purpose of all this classification is to provide modularity which partitions the system into well defined boundaries that can be individually/independently understood, designed, and revised. However, despite "classification theory", choosing what objects represent a system is not always that straight forward. In addition, each analyst or designer will have their own abstraction, or view of the system which must be resolved. Shlaer and Mellor [40], Jacobson [41], Booch [39], and Coad and Yourdon [37] each offer a different look at candidate object classes, as well as other aspects of OOA/OOD. These are all excellent sources for further introduction (or induction) into OOA and OOD. OO does provide a structured approach to software system design and can be very useful in helping to bring about a safer, more reliable system.

While there is a growing number of OO "gurus" with years of practical experience, many OO projects are implemented by those with book-knowledge and little direct experience. It is important not to take everything written in the OOA/OOD books as the only correct way to do things. Adaptation of standard methods may be important in your environment. As an example, the team of software designers who worked on the Mars Pathfinder mission [38] decided to use Object Oriented Design, though their developers had only book-knowledge of the methodology. Attempting to follow the design methodologies verbatim lead to a rapidly increasingly complex set of objects. The team eventually modified the design methodology by combining the "bottom up" approach they had been using with a more "top down" division into subsystems.

#### *4.3.2.2 Unified Modeling Language (UML)*

The Unified Modeling Language (UML) is a methodology and language for specifying, visualizing, and documenting the development artifacts (design) of an object-oriented system. The UML represents the unification of the Booch, Objectory, and OMT methods and is their direct and upwardly compatible successor. It also incorporates ideas from a many other methodologists, including Coad, Gamma, Mellor, Shlaer, and Yourdon.

UML is a graphical modeling language, using a variety of diagrams and charts to show the structure and relationships of an object-oriented design. Class diagrams show the individual classes and how they relate to each other, such as being contained within another class. Each class box can contain some or all of the attributes (data) and operations (methods) of the class.

Relationships among classes come from the following set:

- **Associations** between classes means that they communicate via messages (calling each other's methods).
- **Aggregations** are a specialized association, where one class "owns" the other.
- **Compositions** show that one class is included within another class.
- **Generalizations** represent an inheritance relationship between the classes.
- **Dependencies** are similar to associations, but while one class depends on another, it doesn't contain a pointer or reference to the other class.
- **Realizations** are relationships where one modeling element is the implementation (realization) of another.

The major features of UML include:

- Use cases and scenarios
- Object and class models
- State charts and other behavioral specifications
- Large-scale structuring
- Design patterns
- Extensibility mechanisms

The types of diagrams used by UML include:

- Use-case diagrams
- Class diagrams
- State-machine diagrams
- Message-trace diagrams
- Object-message diagrams

- Process diagrams
- Module diagrams
- Platform diagrams

UML is quickly becoming the standard OO modeling language. Tools already incorporate it, and some can even generate code directly from the UML diagrams. UML has been adapted for real-time systems. Many books now exist for learning UML, as well as on applying UML to specific environments or integrating it with other design methodologies.

### 4.3.3 Selection of COTS and Reuse

During the architectural design phase, and even earlier, decisions are made to select Off-The-Shelf (OTS) items (software, hardware, or both) that are available "as is" from a commercial source (Commercial Off-The-Shelf (COTS)) or to reuse applications developed from other similar projects (i.e., Government Off-The-Shelf (GOTS) items). Any modifications of these items place them in another category – Modified Off-the-Shelf (MOTS) items.

OTS items commonly used include operating systems, processor and device microcode, and libraries of functions. It is becoming prohibitively expensive to custom develop software for these applications. In addition, the desire to not "reinvent the wheel" is strong, especially when faced with budget and schedule constraints. There is also a strong trend in government to use commercial products, instead of custom developing similar but much more expensive products.

Section 7.1 Off-the-Shelf Software covers the pros and cons of OTS and reused software in more detail. Many issues need to be considered before making the decision to use OTS software, or to reuse software from a previous project. While OTS software may appear cost-effective, the additional analyses, tests, glueware code development, and other activities may make it more expensive than developing the software in-house. The section also provided recommendations for additional analyses and tests for OTS software in safety-critical systems.

### 4.3.4 Selection of development tools and operating systems

It is at the architectural design phase that the tools to develop the software, as well as the operating system it will run on, are often selected. The choice of tools/OS can have an impact on the safety of the software. Some operating systems have more "safety" features than others. Some tools make finding errors easier.

Suggestions for what to look for when selecting an operating system, programming language, and development tool are included in *Section 6 SOFTWARE DEVELOPMENT ISSUES* .

### 4.3.5 Coding Standards

Coding Standards can be considered a class of generic software requirements that indicate what software constructs, library functions, and other language-specific information *must* or *must not* be used. As such, they are, in practice, "safe" subsets of programming languages. Coding standards may be developed by the software designer, based on the software and hardware system to be used, or may be general standards for a "safer" version of a particular language.

How compilers "work" internally (convert the higher level language into machine operations) is often undefined and highly variable between compilers. For example, how dynamic memory allocation is implemented is not part of most language specifications, and therefore varies between compilers. Even for a specific compiler, information on how such a process is implemented is difficult to obtain. The location of the allocated memory is usually not predictable, and that may not be acceptable for a safety critical software module. Another example is the order that global items are initialized. Coding standards can be used to make sure that no global item depends on another global item having been already initialized.

It is important that all levels of the project agree to the coding standards, and that they are enforced. If the programmers disagree, they may find ways to circumvent it. Safety requires the cooperation of everyone. Include those who will actually do the programming, as well as software designers and software leads, in any meetings where coding standards will be discussed.

Coding standards may also contain requirements for how the source code is to look, such as indentation patterns and comment formats. However, it is best to separate these requirements into a separate **coding style** document. This avoids the problem of developers shelving the coding standards, because they disagree with the coding style. While trivial in many ways, coding style requirements help make the software more readable by other developers. In addition, they make it more difficult for "typical" errors to be inserted in the code, and easier for them to be found during an inspection.

Create a checklist from the agreed-upon coding standard, for use during software formal inspections and informal reviews. Enforce conformance to the standard during the inspections. Do not rate "style" issues as highly as safety issues. In fact, style issues can be ignored, unless they seriously impact on the readability of the code, or the project decides that they must be enforced.

Benefit-to-Cost Rating: **HIGH**

### 4.3.6 Test Plan Update

At this development phase, the main modules (units) of the software have been defined. This is the time to determine the integration order of the units, and the integration tests that will be run. Update the Test Plan with this information.

## *4.4 Detailed Design Phase*

The following tasks during the detailed design phases should support software safety activities.

- **Program Set Architecture**

  Show positions and functions of safety-critical modules in design hierarchy.

  Identify interfaces of safety-critical components.

  Identify hazardous operations scenarios.

- **Internal Program Set Interfaces**

  Include information on functional interfaces of safety-critical modules.

  Include low level requirements and designs of these interfaces.

  Identify Shared Data

Identify databases/data files which contain safety-critical data and all modules which use them, safety-critical or not.

Document how this data is protected from inadvertent use or changes by non-safety-critical modules.

- **Functional Allocation**

  Document how each safety-critical module can be traced back to original safety requirements and how the requirement is implemented.

  Specify safety-related design and implementation constraints.

  Document execution control, interrupt* characteristics, initialization, synchronization, and control of the modules. Include any finite state machines.

  ***** For high risk systems, interrupts should be avoided as they may interfere with software safety controls developed especially for a specific type of hazard. Any interrupts used should be priority based.

- **Error Detection and Recovery**

  Specify any error detection or recovery schemes for safety-critical modules.

  Include response to language generated exceptions; also responses to unexpected external inputs, e.g. inappropriate commands, or out-of-limit measurements.

- **Inherited or Reused Software and COTS**

  Describe the results of hazard analyses performed on COTS or inherited or reused software.

  Ensure that adequate documentation exists on all software where it is to be used in critical applications.

- **Design Feasibility, Performance, and Margins**

  Show how the design of safety-critical modules is responsive to safety requirements. Include information from any analyses of prototypes or simulations. Define design margins of these modules.

- **Integration**

  Specify any integration constraints or caveats resulting from safety factors.

  Show how safety controls are not compromised by integration efforts.

- **Interface Design**

  For each interface specify a design that meets the safety requirements in the ICD, SIS document of equivalent.

  Minimize interfaces between critical modules.

  Identify safety-critical data used in interfaces.

- **Modularity**

  New modules and sub-modules shall be categorized as safety critical or not.

- **Traceability**

  For each of the above, identify traceability to safety requirements.

- **Testing**

  Identify test and/or verification methods for each safety critical design feature in the software test plan and procedures.

  Results of preliminary tests of prototype code should be evaluated and documented in the Software Development Folders (SDF's).

  Any Safety Critical findings should be reported to the Safety Engineer to help work out any viable solutions.

## 4.5 Software Implementation

It is during software implementation (coding) that software controls of safety hazards are actually implemented. Safety requirements have been passed down through designs to the coding level. Managers and designers must communicate all safety issues relating to the program sets and code modules they assign to programmers. Safety-critical designs and coding assignments should be clearly identified. Programmers must recognize not only the explicit safety-related design elements but should also be cognizant of the types of errors which can be introduced into non-safety-critical code which can compromise safety controls. Coding checklists should be provided to alert for these common errors. This is discussed in the next section.

### 4.5.1 Coding Checklists

Software developers should use coding checklists in software coding and implementation. The coding checklists should be generated by an overall ongoing Specification Activity beginning in the requirements phase, (see *Section 5.1.3 Specification Analysis*). Checklists should contain questions that can serve as reminders to programmers to look for common defects.

- **Safety Checklists**

  During this phase, software safety checklists should be used to verify that safety requirements identified earlier in the design process (as described in previous *Section 4.2.1 Development of Software Safety Requirements,* have, in fact, been flowed into the software detailed design.

- **Code Review Checklists**

  These checklists are for use by the developer, when reviewing her own code or the code of a fellow developer. The checklist should include common errors as well as the coding standards. Personalize the checklist to include individual "common" errors, such as forgetting to end a comment block. This serves as a means to find those errors, and as reminder not to make the same mistakes.

  Additional, more formal checklists should be used during Formal Inspections. These will emphasize the coding standards and the most common errors. Common errors may be gleaned from the individual developers, from this guidebook, or from other sources such as textbooks and articles.

- **Updating requirements**

  Often during this development phase, missing requirements are identified, or new system requirements are added and flowed down to software, such as fault detection and recovery. It may become apparent that various capabilities were assumed, but not explicitly required, so were not implemented. Checklists can help identify these missing requirements. Once missing requirements are identified they must be incorporated by "back-filling" (updating) the requirements specifications prior to implementation in order to maintain proper configuration control. This is less likely to be necessary if Formal Methods or Formal Inspections are used, at least during the requirements phase.

Benefit-to-Cost Rating: **HIGH**

### 4.5.2 Defensive Programming

Hazards can be mitigated (but not controlled) using defensive programming techniques. This incorporates a degree of fault/failure tolerance into the code, sometimes by using software redundancy or stringent checking of input and output data and commands. However, software alone cannot achieve true system redundancy and failure tolerance. Appropriately configured hardware is necessary.

An example of defensive programming is sometimes called "come from" checks. Critical routines have multiple checks in them to test whether they should be executing at some particular instant. One method is for each preceding process to set a flag in a word. If all the proper criteria are met then the routine in question is authorized to execute.

Section 6.15 Good Programming Practices for Safety provides more examples of defensive programming techniques.

Benefit-to-Cost Rating: **HIGH**

### 4.5.3 Refactoring

Refactoring is a technique to restructure object-oriented code in a disciplined way. It is the process of taking an object design and rearranging it in various ways to make the design more flexible and/or reusable. There are several reasons you might want to do this, efficiency and maintainability being probably the most important.

The term "refactoring" comes from mathematics, where you factor an expression into an equivalence. The factors are cleaner ways of expressing the same statement. In software refactoring, there must also be equivalence; the beginning and end products must be functionally identical.

Practically, refactoring means making code clearer, cleaner, simpler and elegant. Refactoring is a good thing, because complex expressions are typically built from simpler, more easily understood components. Refactoring either exposes those simpler components or reduces them to the more efficient complex expression (depending on which way you are going).

A common method of refactoring is to refactor along inheritance lines. For instance, in a design review you find out that two classes in your system that do not share a common superclass both implement very similar or identical behavior. It would be advantageous to then refactor these two classes by moving the common behavior into a shared superclass, and then changing the classes so that they descend from that class.

You can also refactor along composition lines. If you find that a class is implementing two different sets of responsibilities that do not interact with each other much, or that use two subsets of the attributes of the original class, you may want to refactor that class into two different classes, one of which perhaps contains the other.

You also want to refactor when the code is broken, but it isn't broken on the particular axis called "what results the program computes". Some of the defects that can be fixed by refactoring are:

- Unify duplicate code
- Make the code read better (typically introducing more "why")
- Remove unused indirection
- Isolate existing logic from a needed change
- Replace one algorithm with another
- Make the program run faster

Be careful if using refactoring, however. Regression tests should show if the refactored unit is functionally identical to the original. If it is not, then the code has been broken by the refactoring, or was broken originally and is now corrected. Either way, you now have a problem that needs to be fixed.

Too much refactoring may also invalidate code inspections that were done on the original code. Refactoring safety critical units should be followed by a re-inspection of the code, preferably by the original Formal Inspection team.

Benefit-to-Cost Rating: **MEDIUM**

### 4.5.4 Unit Level Testing

Test *planning* starts in the specification phase. This is where the system functional tests are developed, at a high level of understanding. The functional test *execution* begins at the end of integration testing. Integration test *planning* begins during the design phase, when the individual "units" are defined. Integration tests are executed once the actual unit integration begins. Integration and functional testing are described below in *Section 4.6*. Unit level testing is *planned* during the detailed design phase, when the functions within a unit are defined, and *executed* once the code compiles.

Unit level testing is important because it can access levels of the software that might not be reachable once the units are integrated. This testing is usually performed by the developer, though another developer in the group may perform the unit testing. A basic entry criteria for unit testing is that the unit compile without errors. Unit level testing can identify implementation problems requiring changes to the software.

Unit level tests come in two varieties: white-box and black-box. White-box tests include all those where you must know something about the innards of the module. Black-box tests check the inputs/outputs of the module only, and aren't concerned about what happens inside. White-box tests include those that check the path/branch coverage, loop implementation and statement execution. Black-box tests look at the input domain (values), output ranges, and error handling.

Path tests verify that each path through the unit has been tested. Each decision point (if statement, case statement, etc.) leads to two or more paths. Not every combination of paths needs to be tested (that would take too long), but each path decision should be tested for all valid possibilities, and the paths they lead to. Statement execution means that each statement in the program is executed at least once. This is usually done as part of the path tests, but separate tests can be run as well.

Loops are tested by bypassing them, one pass through, a "typical" number of loops, one less than the maximum, the maximum number of loops, and one more than the maximum. Many of these tests are at the "boundaries" of the loop range, which are common sources of problems. Often loops are executed one too few or one two many times.

Boundary values are also used in the input black-box tests. In these, the values that can be input to the unit are tested at typical values, the boundaries, and invalid values. In addition, black-box tests can be created to test the capability, stability, and "graceful behavior" under error conditions of the unit.

Safety tests should be designed for any safety critical units. These tests should demonstrate coverage of the safety test requirements (see *Section 4.6.7 Software Safety Testing*). Each safety test should have pass/fail criteria. The test plan or software management plan should describe the review and reporting process for safety-critical components, including how to report problems discovered in unit test. A test report should be written for unit tests of safety critical items.

Unit tests for object-oriented software consists of testing the class methods in the same way that procedures and functions are tested in structured programs. Construction, destruction, and copying of the class should also be tested.

It is good practice for the developer to keep a list of bugs found while unit testing. This list is a learning tool for the developer to discover what his common errors are. Feed those errors back into the code review checklist, so they are found earlier (at review, rather than test) in the development cycle. In a team where the developers are not penalized for the defects they find, sharing the bug list with other developers and the QA department can help in education and process improvement. Developers can learn from the mistakes of others, and the metrics derived from them can help identify areas to focus on when trying to improve the software development process.

Benefit-to-Cost Rating: **HIGH**

## 4.6 Software Integration and Test

Software testing beyond the unit level is usually handled by someone other than the developer, except in the smallest of teams. These tests are almost exclusively black-box tests, where the inner workings of the units are unknown. All that is known is the interface (inputs, outputs, and their ranges).

The test phases are:

- Integration Testing
  - ™ Unit integration and testing
  - ™ Integration of the software with the hardware
- System Testing
  - ™ Functional testing
  - ™ Performance testing
  - ™ Load testing
  - ™ Stress testing
  - ™ Disaster testing
  - ™ Stability testing
  - ™ Acceptance testing
  - ™ "Red Team" testing

Software testing verifies analysis results, investigates program behavior, and confirms that the program complies with safety requirements. Testing is the operational execution of a software component in a real or simulated environment. Integration testing is often done in a simulated environment, and system testing is usually done on the actual hardware. However, hazardous commands or operations should be tested in a simulated environment first. You don't want to start the rocket engines or set off the ordnance by accident! Testing, conducted in accordance with the safety test plan and procedures, will verify that the software meets safety requirements.

Normally, the software testing ensures that the software performs all required functions correctly, and can exhibit graceful behavior under anomalous conditions. Safety testing focuses on locating program weaknesses and identifying extreme or unexpected situations that could cause the software to fail in ways that would violate safety requirements. Safety testing complements rather than duplicates developer testing.

Fault injection has been successfully used to test critical software (e.g. TUV in Germany). Faults are inserted into code before starting a test and the response is observed. In addition, all boundary and performance requirements should be tested at, below and above the stated limits. It is necessary to see how the software system performs, or fails, outside of supposed operational limits.

The safety testing effort should be limited to those software requirements classed as safety-critical items. In addition, if the safety-critical items are separated from the others via a partition or firewall, the integrity of the partitioning must be tested. Safety testing can be performed as an independent series of tests or as an integral part of the developer's test effort. However, remember that any software which impacts or helps fulfill a safety critical function is safety critical as well.

Any problems discovered during testing should be analyzed and documented in discrepancy reports as well as test reports. Discrepancy reports contain a description of the problems encountered, recommended solutions, and the final disposition of the problem. In addition, defect information may be kept in a defect tracking database. Such a database not only allows tracking of problems for a particular project, but can serve as a source for "lessons learned" and improvement for subsequent projects.

### 4.6.1 Testing Techniques

All of this section is taken, with permission, from the Frequently Asked Questions (FAQ) created by Rick Hower, © 1996-2000. The website is "Software QA and Testing Frequently-Asked-Questions", http://www.softwareqatest.com/, and is an excellent introduction to software testing.

**What kinds of testing should be considered?**

- Black box testing - not based on any knowledge of internal design or code. Tests are based on requirements and functionality.
- White box testing - based on knowledge of the internal logic of an application's code. Tests are based on coverage of code statements, branches, paths, conditions.
- unit testing - the most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses.
- incremental integration testing - continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.
- integration testing - testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.
- functional testing - black-box type testing geared to functional requirements of an application; this type of testing should be done by testers. This doesn't mean that the programmers shouldn't check that their code works before releasing it (which of course applies to any stage of testing.)
- system testing - black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.
- end-to-end testing - similar to system testing; the 'macro' end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

- sanity testing - typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every 5 minutes, bogging down systems to a crawl, or destroying databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.
- regression testing - re-testing after fixes or modifications of the software or its environment. It can be difficult to determine how much re-testing is needed, especially near the end of the development cycle. Automated testing tools can be especially useful for this type of testing.
- acceptance testing - final testing based on specifications of the end-user or customer, or based on use by end-users/customers over some limited period of time.
- load testing - testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system's response time degrades or fails.
- stress testing - term often used interchangeably with 'load' and 'performance' testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database system, etc.
- performance testing - term often used interchangeably with 'stress' and 'load' testing. Ideally 'performance' testing (and any other 'type' of testing) is defined in requirements documentation or QA or Test Plans.
- usability testing - testing for 'user-friendliness'. Clearly this is subjective, and will depend on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers.
- install/uninstall testing - testing of full, partial, or upgrade install/uninstall processes.
- recovery testing - testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.
- security testing - testing how well the system protects against unauthorized internal or external access, willful damage, etc; may require sophisticated testing techniques.
- compatibility testing - testing how well software performs in a particular hardware/software/operating system/network/etc. environment.
- user acceptance testing - determining if software is satisfactory to an end-user or customer.
- comparison testing - comparing software weaknesses and strengths to competing products.
- alpha testing - testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers.

- beta testing - testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers.

**What steps are needed to develop and run software tests?**

The following are some of the steps to consider:

- Obtain requirements, functional design, and internal design specifications and other necessary documents
- Obtain budget and schedule requirements
- Determine project-related personnel and their responsibilities, reporting requirements, required standards and processes (such as release processes, change processes, etc.)
- Identify application's higher-risk aspects, set priorities, and determine scope and limitations of tests
- Determine test approaches and methods - unit, integration, functional, system, load, usability tests, etc.
- Determine test environment requirements (hardware, software, communications, etc.)
- Determine testware requirements (record/playback tools, coverage analyzers, test tracking, problem/bug tracking, etc.)
- Determine test input data requirements
- Identify tasks, those responsible for tasks, and labor requirements
- Set schedule estimates, timelines, milestones
- Determine input equivalence classes, boundary value analyses, error classes
- Prepare test plan document and have needed reviews/approvals
- Write test cases
- Have needed reviews/inspections/approvals of test cases
- Prepare test environment and testware, obtain needed user manuals/reference documents/configuration guides/installation guides, set up test tracking processes, set up logging and archiving processes, set up or obtain test input data
- Obtain and install software releases
- Perform tests
- Evaluate and report results
- Track problems/bugs and fixes
- Retest as needed

- Maintain and update test plans, test cases, test environment, and testware through life cycle

**What's a 'test case'?**

A test case is a document that describes an input, action, or event and an expected response, to determine if a feature of an application is working correctly. A test case should contain particulars such as test case identifier, test case name, objective, test conditions/setup, input data requirements, steps, and expected results.

Note that the process of developing test cases can help find problems in the requirements or design of an application, since it requires completely thinking through the operation of the application. For this reason, it's useful to prepare test cases early in the development cycle if possible.

**What should be done after a bug is found?**

The bug needs to be communicated and assigned to developers that can fix it. After the problem is resolved, fixes should be re-tested, and determinations made regarding requirements for regression testing to check that fixes didn't create problems elsewhere. If a problem-tracking system is in place, it should encapsulate these processes. A variety of commercial problem-tracking/management software tools are available (see the 'Tools' section for web resources with listings of such tools). The following are items to consider in the tracking process:

- Complete information such that developers can understand the bug, get an idea of its severity, and reproduce it if necessary.
- Bug identifier (number, ID, etc.)
- Current bug status (e.g., 'Released for Retest', 'New', etc.)
- The application name or identifier and version
- The function, module, feature, object, screen, etc. where the bug occurred
- Environment specifics, system, platform, relevant hardware specifics
- Test case name/number/identifier
- One-line bug description
- Full bug description
- Description of steps needed to reproduce the bug if not covered by a test case or if the developer doesn't have easy access to the test case/test script/test tool
- Names and/or descriptions of file/data/messages/etc. used in test
- File excerpts/error messages/log file excerpts/screen shots/test tool logs that would be helpful in finding the cause of the problem
- Severity estimate (a 5-level range such as 1-5 or 'critical'-to-'low' is common)
- Was the bug reproducible?

- Tester name
- Test date
- Bug reporting date
- Name of developer/group/organization the problem is assigned to
- Description of problem cause
- Description of fix
- Code section/file/module/class/method that was fixed
- Date of fix
- Application version that contains the fix
- Tester responsible for retest
- Retest date
- Retest results
- Regression testing requirements
- Tester responsible for regression tests
- Regression testing results

A reporting or tracking process should enable notification of appropriate personnel at various stages. For instance, testers need to know when retesting is needed, developers need to know when bugs are found and how to get the needed information, and reporting/summary capabilities are needed for managers.

**What if there isn't enough time for thorough testing?**

Use risk analysis to determine where testing should be focused. Since it's rarely possible to test every possible aspect of an application, every possible combination of events, every dependency, or everything that could go wrong, risk analysis is appropriate to most software development projects. This requires judgment skills, common sense, and experience. (If warranted, formal methods are also available.)

Considerations can include:

- Which functionality is most important to the project's intended purpose?
- Which functionality is most visible to the user?
- Which functionality has the largest safety impact?
- Which functionality has the largest financial impact on users?
- Which aspects of the application are most important to the customer?
- Which aspects of the application can be tested early in the development cycle?

- Which parts of the code are most complex, and thus most subject to errors?
- Which parts of the application were developed in rush or panic mode?
- Which aspects of similar/related previous projects caused problems?
- Which aspects of similar/related previous projects had large maintenance expenses?
- Which parts of the requirements and design are unclear or poorly thought out?
- What do the developers think are the highest-risk aspects of the application?
- What kinds of problems would cause the worst publicity?
- What kinds of problems would cause the most customer service complaints?
- What kinds of tests could easily cover multiple functionalities?
- Which tests will have the best high-risk-coverage to time-required ratio?

### 4.6.2 Test Setups and Documentation

Testing should be performed either in a controlled environment in which execution follows a structured test procedure and the results are monitored, or in a demonstration environment where the software is exercised without interference.

**Controlled testing** executes the software on a real or a simulated computer using special techniques to influence behavior. This is the usual mode of testing, where a test procedure (script) is developed and followed, and the results are noted. Automatic testing is also included in this category. All of the integration and system tests that will be discussed in the following sections are controlled tests.

When using a simulator, rather than the real system, the fidelity of the simulators should be carefully assessed. How close is the simulator to the "real thing"? How accurate is the simulator? Has the simulator itself been verified to operate correctly?

**Demonstration testing** executes the software on a computer and in an environment identical to the operational computer and environment. Demonstrations may be used in the acceptance test, to show the user how the system works. Autonomous systems, where the internal operation is completely under the control of the software, would also be demonstrated, especially for the acceptance test.

**Safety testing** exercises program functions under both nominal and extreme conditions. Safety testing includes nominal, stress, performance, load and error-handling testing. These tests are discussed in *Section 4.6.5 System Tests*. Additional safety tests may be developed to test the software under specific conditions (e.g. unexpected loss of power) or for specific results (door doesn't open when the furnace is on, no matter what command is issued to the software)

**Configuration Management** should act as the sole distributor of media and documentation for all system tests and for delivery to [sub]system integration and testing. Pulling the latest program off the developers machine is not a good idea. One aspect of system testing is *repeatability*, which can only be assured if the software under test comes from a known, and fixed, source.

Software Test Reports should incorporate software safety information in the following sections:

- **Unit Testing Results**

  Report results from unit tests in the Software Development Folders. Ideally, all results should be documented, even if just in notes in a log book. Safety critical units should have more formal documentation (test reports).

- **System Test Reports**

  Report results of testing safety-critical interfaces versus the requirements outlined in the Software Test Plan. Any Safety Critical findings should be used to update the hazard reports.

### 4.6.3 Integration Testing

Integration is the process of piecing together the "puzzle", where each piece is a software unit. The end result is a complete system, and the final integration test is essentially the first system functional test.

The order of integration of the units should be decided at the end of the architectural design, when the units are identified. Various schemes can be used, such as creating a backbone (minimal functionality) and adding to it, or doing the most difficult modules first. Keep in mind the hardware schedule when deciding the order of integration. Late hardware may hold up software integration, if the software that needs it is integrated early in the cycle.

Stubs and drivers are used to "simulate" the rest of the system, outside of the integrated section. Stubs represent units below (called by) the integrated section. Drivers represent the part of the software that calls the integrated section.

Integration tests are black-box tests that verify the functionality of the integrated unit. They are "higher level" black-box unit tests, where the "unit" is the new, integrated whole.

Special safety tests may be run as safety critical units are integrated. These tests should exercise functionality that may be unavailable once the system is completely integrated. Also, some safety tests may be run early, so that any problems can be corrected before development is completed.

### 4.6.4 Object Oriented Testing

Object-oriented software requires some changes in the testing strategy, prior to the full system tests. Once the software is fully integrated, it doesn't matter what the underlying software design is. A system is a system.

However, when the system is integrated, whole classes are added at a time. Besides the normal "functional" tests that are performed during integration testing, consider the following tests as well:

- Object A creates Object B, invoking B's constructor
- A deletes B, invoking B's destructor. Check for memory leaks here!
- A sends a "message" to B (invokes a method of B). Check for situations where B is not present (not created or already destroyed). How does the error handling system deal with that situation?

As each class is integrated into the system, it is the *interfaces* with other classes that must be tested.

Object-oriented testing methods have not reached the maturity of more traditional testing. The best way to test OO software is highly debated. The following resources provide some useful insights or links regarding OO testing:

- "Testing Object-Oriented Software" (book), by David C. Kung, Pei Hsia, and Jerry Gao, October 1998. ISBN 0-8186-8520-4
- "Practical Techniques for Testing Objects", Powerpoint Presentation, http://www.cigital.com/presentations/testing_objects/sld001.htm
- Bibliography: Testing Object-Oriented Software http://www.rbsc.com/pages/ootbib.html
- Cetus links on Object-Oriented Testing: http://www.cetus-links.org/oo_testing.html
- Myths about OO testing: http://www.rbsc.com/pages/myths.html
- State of the art in 1995: http://www.stsc.hill.af.mil/crosstalk/1995/apr/testinoo.asp

### 4.6.5 System Testing

System testing begins when the software is completely integrated. Several types of tests are usually run. Not every test is useful for every system, and the software developer should choose those that test specific requirements or that may show problems with the system.

- **Functional Testing** consists of running the system in a nominal manner. The system is verified to perform all the functions specified in the requirements, and to not perform functions that are designated "must not work". A complete, end-to-end functional test is often designated as the System Test, though system testing actually encompasses many more types of tests. A scaled-down version of the functional test is often used as the acceptance test. Benefit-to-Cost Rating: **HIGH**

- **Stress tests** are designed to see how much the system can handle before it breaks down. While a capacity test (performance) may test that the system can store the required number of files, a stress test will see just how many files can be stored before the disk runs out of room. Aspects of the system that might be stressed are CPU usage, I/O response time, paging frequency, memory utilization, amount of available memory, and network utilization. The closer a system's peak usage is to the breakdown point, the more likely it is that the system will fail under usage. Give yourself adequate margin, if at all possible. Benefit-to-Cost Rating: **HIGH**

- **Stability tests** look for sensitivity to event sequences, intermittent bad data, memory leakage, and other problems that may surface when the system is operated for an extended period of time. Benefit-to-Cost Rating: **HIGH**

- **Resistance to Failure** tests how gracefully the software responds to errors. Errors should be detected and handled locally. Erroneous user input should be handled appropriately (e.g. ignored with an error message), as well as bad input from sensors or other devices. Fault injection tests fit in this category. Benefit-to-Cost Rating: **HIGH**

- **Compatibility tests** verify that the software can work with the hardware and other software systems it was designed to interface with. Benefit-to-Cost Rating: **HIGH**
- **Performance testing** verifies the "CARAT" parameters – Capacity, Accuracy, Response time, Availability, and Throughput. Capacity is the number of records, users, disk space, etc. Accuracy is the verification of algorithm results and precision. Response time is often important in real-time systems, and will be included in the specifications. Availability is how long the system can be used (based on how often it fails, and how long to repair it when it does fail). Throughput is the peak or average number of events per unit time that the system can handle. **Load tests** are a form of performance testing. Benefit-to-Cost Rating: **HIGH**
- **Disaster testing** checks the software's response to physical hardware failure. Pulling the power plug while the software is running is one example. Disaster tests point to areas where the software needs fixing, or where additional hardware is needed (such as a backup battery, to allow the software to shut down gracefully). Benefit-to-Cost Rating: **MEDIUM**
- **Installation** test shows that the software can be installed successfully. Benefit-to-Cost Rating: **MEDIUM**
- **"Red Team" testing** is a totally unscripted, "break the code" type of testing. It is only performed after all other testing is completed, and in an environment where a true safety problem cannot develop (such as on a simulator). The testers do whatever they want to the software except actually break the hardware it runs on (or related hardware). This is a random test. Successful completion suggests that the program is robust. Failure indicates that something needs to be changed, either in the software or in the operating procedures. Benefit-to-Cost Rating: **LOW** (but fun!)

### 4.6.6 Regression Testing

Whenever changes are made to the system after it is baselined (first release), a regression test must be run to verify that previous functionality is not affected and that no new errors have been added. This is vitally important! **"Fixed" code may well add its own set of errors to the code.** If the system is close to some capacity limit, the corrected code may push it over the edge. Performance issues, race conditions, or other problems that were not evident before may be there now.

If time permits, the entire suite of system and safety tests would be rerun as the complete regression test. However, for even moderately complex systems, such testing is likely to be too costly in time and money. Usually, a subset of the system tests make up the regression test suite. Picking the proper subset, however, is an art and not a science.

**Minimization** is one approach to regression test selection. The goal is to create a regression test suite with the minimal number of tests that will cover the code change and modified blocks. The criteria for this approach is coverage – what statements are executed by the test. In particular, every statement in the changed code must be executed, and every modified block must have at least one test.

**Coverage** approaches are based on coverage criteria, like the minimization approach, but they are not concerned about minimizing the number of tests. Instead, all system tests that exercise the changed or affected program component(s) are used.

**Safe** approaches place less emphasis on coverage criteria, and attempt instead to select every test that will cause the modified program to produce different output than the original program. Safe regression test selection techniques select subsets that, under certain well-defined conditions, exclude no tests (from the original test suite) that if executed would reveal faults in the modified software.

Program slicing can be a helpful technique for determining what tests to run. Slicing finds all the statements that can affect a variable, or all statements that a variable is involved with. Depending on the changes, slicing may be able to show what modules may be affected by the modification.

Whatever strategy is used to select the regression tests, it should be a well thought out process. Balance the risks of missing an error with the time and money spent on regression testing. Very minor code changes usually require less regression testing, unless they are in a very critical area of the software. Also consider including in the regression suite tests that previously found errors, tests that stress the system, and performance tests. You want the system to run at least as well *after* the change as it did *before* the change!

For safety critical code, or software that resides on the same platform as safety critical code, the **software safety tests must be repeated**, even for minor changes.

**Regression testing should include functional, performance, stress, and safety testing of the altered code and all modules it interacts with.**

### 4.6.7 Software Safety Testing

Developers must perform software safety testing to ensure that hazards have been eliminated or controlled to an acceptable level of risk. Also, document safety-related test descriptions, procedures, test cases, and the associated qualifications criteria. Implementation of safety requirements (inhibits, traps, interlocks, assertions, etc.) shall be verified. Verify that the software functions safely both within its specified environment (including extremes), and under specified abnormal and stress conditions. For example, two failure tolerant systems should be exercised in all predicted credible two failure scenarios.

IEEE 1228-1994, Software Safety Plans, specifies that the following software safety tests may be performed:

- Computer software unit level testing that demonstrates correct execution of critical software elements.
- Interface testing that demonstrates that critical computer software units execute together as specified.

- Computer software configuration item testing that demonstrates the execution of one or more system components.
- System-level testing that demonstrates the software's performance within the overall system.
- Stress testing that demonstrated the software will not cause hazards under abnormal circumstances, such as unexpected input values or overload conditions.
- Regression testing that demonstrates changes made to the software did not introduce conditions for new hazards.

Software Safety, usually represented by the Software Quality Assurance organization, should participate in the testing of safety-critical computer software components at all levels of testing, including informal testing, system integration testing, and Software Acceptance testing.

Benefit-to-Cost Rating: **HIGH**

### 4.6.8 Test Witnessing

Software safety personnel should ensure that tests of safety-critical components are conducted in strict accordance with the approved test plans, descriptions, procedures, scripts and scenarios, and that the results are accurately logged, recorded, documented, analyzed, and reported. Ensure that deficiencies and discrepancies are corrected and retested.

In addition to testing under normal conditions, the software should be tested to show that unsafe states can not be generated by the software as the result of feasible single or multiple erroneous inputs. This should include those outputs which might result from failures associated with the entry into, and execution of, safety-critical computer software components. Negative and No-Go testing should also be employed, and should ensure that the software only performs those functions for which it is intended, and no extraneous functions. Lists of specific tests for safety-critical software can be found in *Section 3.3 Incorporating Software Safety into Software Development* in this Guidebook.

Witnessing verifies that the software performs properly and safely during system integration stress testing and system acceptance testing. System acceptance testing should be conducted under actual operating conditions or realistic simulations.

### 4.7 Software Acceptance and Delivery Phase

Once the software has passed acceptance testing it can be released either as a stand-alone item, or as part of a larger system acceptance.

An Acceptance Data Package (ADP) should accompany the release of the software. This package should include, as a minimum, the following:

- ✓ Instructions for installing all safety-critical items. Define the hardware environment for which the software is certified.
- ✓ Liens identify all safety-related software liens, such as missing design features, or untested features.
- ✓ Constraints describe operational constraints for hazardous activities. List all open, corrected but not tested, or uncollected safety-related problem reports. Describe environmental limitations of use, allowable operational envelope.
- ✓ Operational procedures describe all operational procedures and operator interfaces. Include failure diagnosis and recovery procedures.

In addition, the ADP should contain the following:

- ✓ Certificate of Conformance to requirements, validated by Quality Assurance Organization, and IV & V Organization (if applicable).
- ✓ Waivers lists any waivers of safety requirements.
- ✓ Program Set Acceptance Test Results lists results of safety tests.
- ✓ New or Changed Capabilities lists any approved change requests for safety-critical items.
- ✓ Problem Disposition lists any safety-related problem reports.
- ✓ Version Description Document describes as built versions of software modules to be used with this release of the system.

### 4.8 Software Operations & Maintenance

Maintenance of software differs completely from hardware maintenance. Unlike hardware, software does not degrade or wear out over time, so the reasons for software maintenance are different.

The main purposes for software maintenance are as follows:

- to correct known defects
- to correct defects discovered during operation
- to add or remove features and capabilities (as requested by customer, user or operator)
- to compensate or adapt for hardware changes, wear out or failures.

The most common safety problem during this phase is lack of configuration control, resulting in undocumented and poorly understood code. "Patching" is a common improper method used to "fix" software "on the fly". Software with multiple undocumented patches has resulted in major problems where it has become completely impossible to understand how the software really functions, and how it responds to its inputs.

In some cases, additional software has been added to compensate for unexpected behavior which is not understood. ( for example, "garbage collectors"). It is beneficial to determine and correct the root cause of unexpected behavior, otherwise the software can "grow" in size to exceed available resources, or become unmanageable.

After software becomes operational, rigorous configuration control must be enforced. For any proposed software change, it is necessary to repeat all life cycle development and analysis tasks performed previously from requirements (re-) development through code (re-)test. Full original testing is recommended as well as any additional tests for new features.

It is advisable to perform the final verification testing on an identical off-line analog (or simulator) of the operational software system, prior to placing it into service.

# 5. SOFTWARE SAFETY ANALYSIS

Safety is an integral part of the software life-cycle, from the specification of safety-related requirements, through inspection of the software controls, and into verification testing for hazards. Within each life-cycle phase, the safety engineer performs various analysis tasks. If problems are found, they are fed back through the system until they are corrected or mitigated. While finding unsafe elements of the system is often the focus of the analyses, a "negative" analysis (no hazards or major problems) can give the project assurance that they are on the right path to a safe system.

This section describes various techniques that have been useful in NASA activities and within industry. How to do the analysis (the methodology), what are the inputs, and what products are generated, is described in each case. Wherever possible, checklists are provided to aid in the use of the technique.

Analysis techniques fall into two categories:

1. Top down system hazards and failure analyses, which look at possible hazards or faults and trace down into the design to find out what caused them.
2. Bottom up review of design products to identify failure modes not predicted by top down analysis. This will ensure validity of assumptions of top down analysis, and verify conformance to requirements.

Typically, both types of analyses will be used in a typical software safety analysis activity, though the specific techniques used will be tailored for the project. A benefit-to-cost rating is given for each technique, to assist in the planning of software safety activities. The rating is HIGH (benefits far outweigh costs), MEDIUM (less benefits or higher cost) and LOW (high cost for what benefits you get). This scale is subjective and meant to be only one consideration when choosing analysis techniques.

Results of software safety analysis are reported back to the system safety organization for integration in the system safety plan. For example, a new software hazard may require changes to the hardware configuration to mitigate it. Or an analysis may show that software can contain a hazard sufficiently to allow it to be a control.

As the software becomes more defined within the software life cycle, individual program sets, modules, units, etc. are identified that are safety-critical. The analyses used vary with the phase of development, building on previous analyses or using the new level of software definition to refine the safety analysis. Each set of analyses are described in the following sections:

5.1 Software Safety Requirements Analysis

5.2 Architectural Design Analysis

5.3 Detailed Design Analysis

5.4 Code Analysis

5.5 Test Analysis

5.6 Operations & Maintenance

## 5.1 Software Safety Requirements Analysis

The Requirements Analysis Activity verifies that safety requirements for the software were properly flowed down from the system safety requirements, and that they are correct, consistent and complete. It also looks for new hazards, software functions that can impact hazard controls, and ways the software can behave that are unexpected. These are primarily top down analyses.

Bottom up analysis of software requirements are performed such as Requirements Criticality Analysis to identify possible hazardous conditions. This results in another iteration of the PHA that may generate new software requirements. Specification analysis is also performed to ensure consistency of requirements.

Analyses included in the Software Requirements Phase are:

- Software Safety Requirements Flow-down Analysis
- Requirements Criticality Analysis
- Specification Analysis
- Formal Inspections
- Timing, Throughput And Sizing Analysis
- Preliminary Software Fault Tree Analysis

### 5.1.1 Software Safety Requirements Flow-down Analysis

Safety requirements are flowed down into the system requirements specifications, and from there into the subsystem specifications, as described in Section 4.2.1.1. This includes the software subsystem requirements.

Problems in the flow-down process can be caused by incomplete analysis, inconsistent analysis of highly complex systems, or use of ad hoc techniques by biased or inexperienced analysts. The following references are a good starting point for anyone who falls into the “inexperienced” category :

- MIL-STD-882C System Safety Program Requirements (the ‘C’ version, not the current ‘D’, has some description on how to verify flow down of requirements)
- NSTS-22254 Methodology for Conduct of Space Shuttle Program Hazard Analyses
- “Safeware : System Safety and Computers” (Book), Nancy Leveson, April 1995
- “Safety-Critical Computer Systems” (Book), Neil Storey, August 1996
- “Software Assessment: Reliability, Safety, Testability” (Book), Michael A. Friedman and Jeffrey M. Voas(Contributor), August 16, 1995
- “Discovering System Requirements”, A. Terry Bahill and Frank F. Dean, http://tide.it.bond.edu.au/inft390/002/Resources/sysreq.htm

The most rigorous (and most expensive) method of addressing this concern is adoption of formal methods for requirements analysis and flow-down. This was described previously in *Section 4.2.3 Formal Methods - Specification Development*. Less rigorous and less expensive ways include checklists and/or a standardized structured approach to software safety as discussed below and throughout this guidebook.

Benefit-to-Cost Rating: **HIGH**

#### *5.1.1.1 Checklists and cross references*

Checklists are a tool for making sure you haven't forgotten anything important, while doing an analysis or reviewing a document. They are a way to put the collective experience of those who created and reviewed the checklist to work on your project. They are a starting point, and should be reviewed for relevance for each project. A collection of checklists is provided in Appendix E. For the requirements phase, they include a safety checklist that contains standard hazards to look for when reviewing the requirements specification and a checklist of questions to think about when reviewing the PHA.

Cross references are matrices that list related items. A matrix that shows the software related hazards and hazard controls and their corresponding safety requirements should be created and maintained. This should be a living document, reviewed and updated periodically. Refreshing your mind on the hazards that software must control while working on the software design, for example, increases the likelihood that the hazard controls will be designed in correctly. Another cross reference matrix would list each requirement and the technique that will verify it (analysis, test, etc.).

You should develop a systematic checklist of software safety requirements and 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 as discussed in *Section 4.2.1 Development of Software Safety Requirements*. *Section 4.2.5 Formal Inspections*, lists some sources for starting a safety checklist.

Also, develop a hazard requirements flow-down matrix which maps safety requirements and hazard controls to system/software functions and, from there, to software modules and components. Where components are not yet defined, flow to the lowest level possible and tag for future flow-down.

Benefit-to-Cost Rating: **HIGH**

### 5.1.2 Requirements Criticality Analysis

Criticality analysis identifies program requirements that have safety implications. A method of applying criticality analysis is to analyze the hazards of the software/hardware system and identify those that could present catastrophic or critical hazards. 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. Software safety engineers and software development engineers should be included in this discussion. Software is a vital component in the whole system, and the "software viewpoint" must be part of any systems engineering activity. Requirements can be consolidated to reduce the number of critical requirements. In addition, they can be flagged for special attention during design, to reduce the criticality level.

Keep in mind that not all "safety critical" requirements are created equal. Later in the process, the concept of *risk* is used to prioritize which requirements or components are more critical than others. For now, it's best to look at everything that can cause a safety problem, even a trivial one. It's easier, and cheaper, to remove or reduce requirements later than it is to add them in.

It is probable that software components or subsystems will 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.

You perform the Requirements Criticality Analysis by doing the following:

- All software requirements are analyzed to identify additional potential system hazards that the system PHA did not reveal. A checklist of PHA hazards is a good thing to have while reviewing the software requirements. The checklist makes it easier to identify PHA-designated hazards that are not reflected in the software requirements, and new hazards missed by the PHA. In addition, look for areas where system requirements were not correctly flowed to the software. Once potential hazards have been identified, they are added to the system requirements and then flowed down to subsystems (hardware, software and operations) as appropriate.
- Review the *system* requirements to identify hardware or software functions that that receive, pass, or initiate critical signals or hazardous commands.
- Review the *software* requirements to verify that the functions from the system requirements are included. In addition, look for any new software functions or objects that receive/pass/initiate critical signals or hazardous commands.
- Look through the *software* requirements for conditions that may lead to unsafe situations. Consider conditions 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 in *Section 5.1.2.1 Critical Software Characteristics*.

The following resources are available for the Requirements Criticality Analysis:

| | |
|---|---|
| Software Development Activities Plan [Software Development Plan] Software Assurance Plan [None], Software Configuration Management Plan [Same] and Risk Management Plan [Software Development Plan] | Background information relating to safety requirements associated with the contemplated testing, manufacturing, storage, repair, installation, use, and final disposition of the system |
| System and Subsystem Requirements [System/Segment Specification (SSS), System/Segment Design Document] | Storage and timing analyses and allocations |
| Requirements Document [Software Requirements Specifications] | Program structure documents |
| External Interface Requirements Document [Interface Requirements Specifications] and other interface documents | 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 |
| Functional Flow Diagrams and related data | Historical data such as lessons learned from other systems and problem reports |

Note: documents in [parentheses] correspond to terminology from DOD-STD-2167 [2]. Other document names correspond to NASA-STD-2100.91.

Output products from this analysis are:

- Table 5-1 Subsystem Criticality Matrix
- Updated Safety Requirements Checklist
- Definition of Safety Critical Requirements

The results and findings of the Criticality Analyses should be fed back 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 Criticality 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 (FI) checklists for performing the FI process described in *Section 4.2.5 Formal Inspections of Specifications*.

Benefit-to-Cost Rating: **HIGH**

#### *5.1.2.1 Critical Software Characteristics*

Not all characteristics of the software are governed by requirements. Some characteristics are a result of the design, which may fulfill the requirements in a variety of ways. It is important that safety critical characteristics are identified and explicitly included in the requirements. "Forgotten" safety requirements often come back to bite you late in the design or coding stages.

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 (see *Section 4.2.2.2 Hazardous Commands*)
- ✓ 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 which 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.

**Table 5-1 Subsystem Criticality Matrix**

| Mission Operational Control Functions | Hazards | | |
|---|---|---|---|
| | IMI | CA | ICD |
| Communication | X | X | X |
| Guidance | | X | |
| Navigation | | X | |
| Camera Operations | | | X |
| Attitude Reference | X | X | X |
| Control | X | X | |
| Pointing | | X | |
| Special Execution | | | |
| Redundancy Management | X | | |
| Mission Sequencing | X | | |
| Mode Control | X | X | |

<u>Key</u>

| | |
|---|---|
| IMI | Inadvertent Motor Ignition |
| CA | Collision Avoidance |
| ICD | Inadvertent Component Deployment |

The above matrix is an example output of a software *5.1.2 Requirements Criticality Analysis.* Each functional subsystem is mapped against system hazards identified by the PHA. In this example, three hazards are addressed.

This matrix is an essential tool to define the criticality level of the software. Each hazard should have a risk index as described in *Section 2.4.1.2 Risk Levels* of this guidebook. The risk index is a means of prioritizing the effort required in developing and analyzing respective pieces of software.

### 5.1.3 Specification Analysis

Specification analysis evaluates the completeness, correctness, consistency, and testability of software requirements. Well-defined requirements are strong standards by which to evaluate a software component. Specification analysis should evaluate requirements individually and as an integrated set. Techniques used to perform specification analysis are:

- control-flow analysis,
- information-flow analysis
- functional simulation

These techniques are described in detail (plus background and theory) within a large, well established body of literature. Look in books on software testing and software engineering for further information on these techniques. A brief description of each technique will be given so that the analyst can determine if further study is warranted.

The safety organization should ensure the software requirements appropriately influence the software design and the development of the operator, user, and diagnostic manuals. The safety agency should review the following documents and/or data:

| | |
|---|---|
| System/segment specification and subsystem specifications | Storage allocation and program structure documents |
| Software requirements specifications | Background information relating to safety requirements |
| Interface requirements specifications and all other interface documents | Information concerning system energy, toxic and other hazardous event sources, especially those that may be controlled directly or indirectly by software |
| Functional flow diagrams and related data | Software Development Plan, Software Quality Evaluation Plan, and Software Configuration Management Plan and Historical data |

#### 5.1.3.1 *Control-flow analysis*

Control-flow analysis examines the order in which software functions will be performed. It identifies missing and inconsistently specified functions. Control-flow examines which processes are performed in series, and which in parallel (e.g., multitasking), and which tasks are prerequisites or dependent upon other tasks.

Benefit-to-Cost Rating: **HIGH**

#### 5.1.3.2 *Information-flow analysis*

Information-flow analysis examines the relationship between functions and data. Incorrect, missing, and inconsistent input/output specifications are identified. Data flow diagrams are commonly used to report the results of this activity, so this technique is best used during architectural design. However, it can also be effective during fast prototyping and/or spiral life cycle models for early, basic data and command flow.

Benefit-to-Cost Rating: **HIGH**

#### 5.1.3.3 *Functional simulation models*

Simulators are useful development tools for evaluating system performance and human interactions. You can examine the characteristics of a software component to predict performance, check human understanding of system characteristics, and assess feasibility. Simulators have limitations in that they are representational models and sometimes do not accurately reflect the real design, or make environmental assumptions which can differ from conditions in the field.

Benefit-to-Cost Rating: **MEDIUM**

### 5.1.4 Formal Inspections

Software Formal Inspections (described in *4.2.5 Formal Inspections of Specifications*) are an important activity for safety to participate in, especially during the requirements phase. Early in the life-cycle of the project is a very good time at which to express safety concerns and recommend changes. In addition, the safety representative can more thoroughly learn about the system and how it is supposed to work. As part of the inspection activity, a safety checklist should be created to use when reviewing the requirements. This checklist should be based on the safety requirements discussed in *Section 4.2 Software Requirements Phase*.

The generic requirements portion of the checklist should be tailored to emphasize those areas most relevant and those most likely to be omitted or not satisfied. Reference [6] "Targeting Safety-Related Errors During Software Requirements Analysis" contains a good checklist relevant to the NASA environment.

After the inspection, the safety representative should review the official findings of the inspection and translate any that require safety follow-up on to a worksheet such as that in *Table 4-2 Subsystem Criticality Analysis Report Form*. This form can then serve in any subsequent inspections or reviews as part of the checklist. It will also allow the safety personnel to track to closure safety specific issues that arise during the course of the inspection.

Benefit-to-Cost Rating: **HIGH**

### 5.1.5 Timing, Throughput And Sizing Analysis

Timing, throughput and sizing analysis for safety critical functions evaluates software requirements that relate to execution time, I/O data rates and memory/storage allocation. This analysis focuses on program constraints. Typical constraint requirements are maximum execution time, maximum memory usage, maximum storage size for program, and I/O data rates the program must support. The safety organization should evaluate the adequacy and feasibility of safety critical timing, throughput and sizing requirements. These analyses also evaluate whether adequate resources have been allocated in each case, under worst case scenarios. For example, will I/O channels be overloaded by many error messages, preventing safety critical features from operating.

Items to consider include:

- memory usage versus availability;
- I/O channel usage (load) versus capacity and availability;
- execution times versus CPU load and availability;
- sampling rates versus rates of change of physical parameters.
- program storage space versus executable code size
- amount of data to store versus available capacity

Quantifying timing/sizing resource requirements can be very difficult. Estimates can be based on the actual parameters of similar existing systems.

- **Memory usage versus availability**

  Assessing memory usage can be based on previous experience of software development if there is sufficient confidence. More detailed estimates should evaluate the size of the code to be stored in the memory, and the additional space required for storing data and scratch pad space for storing interim and final results of computations (heap size). As code is developed, particularly prototype or simulation code, the memory estimates should be updated.

  Consider carefully the use of Dynamic Memory Allocation in safety critical code or software that can impact on the safety critical portions. Dynamic memory allocation can lead to problems from not freeing allocated memory (memory leak), freeing memory twice (causes exceptions), or buffer overruns that overwrite code or other data areas. When data structures are dynamically allocated, they often cannot be statically analyzed to verify that arrays, strings, etc. do not go past the physical end of the structure.

- **I/O channel usage (Load) versus capacity and availability**

  Look at the amount of input data (science data, housekeeping data, control sensors) and the amount of output data (communications) generated. "I/O channel" should include internal hardware (sensors), interprocess communications (messages), and external communications (data output, command and telemetry interfaces). Check for resource conflicts between science data collection and safety critical data availability. During failure events, I/O channels can be overloaded by error messages and these important messages can be lost or overwritten. (e.g. the British "Piper Alpha" offshore oil platform disaster). Possible solutions includes additional modules designed to capture, correlate and manage lower level error messages or errors can be passed up through the calling routines until at a level which can handle the problem; thus, only passing on critical faults or combinations of faults, that may lead to a failure.

- **Execution times versus CPU load and availability**

  Investigate the time variations of CPU load and determine the circumstances that generate peak load. Is the execution time under high load conditions acceptable? Consider the timing effects from multitasking, such as message passing delays or the inability to access a needed resource because another task has it. Note that excessive multitasking can result in system instability leading to "crashes". Also consider whether the code will execute from RAM or from ROM, which is often slower to access.

- **Sampling rates versus rates of change of physical parameters**

  Design criteria for this is discussed in *Section 4.2.2.5 Timing, Sizing and Throughput Considerations*. Analysis should address the validity of the system performance models used, together with simulation and test data, if available.

- **Program storage space versus executable code size**

  Estimate the "footprint" of the executable software in the device it is stored in (EPROM, flash disk, etc.). This is may be less than the memory footprint, as only static or global variables take up space. However, if not all modules will be in memory at the same time, then the executable size may be larger. The program size includes the operating system as well as the application software.

❖ **Amount of data to store versus available capacity**

Consider how much science, housekeeping, or other data will be generated and the amount of storage space available (RAM, disk, etc.). If the data will be sent to the ground and then deleted from the storage media, then some analysis should be done to determine how often, if ever, the "disk" will be full. Under some conditions, being unable to save data or overwriting previous data that has not been downlinked could be a safety related problem.

Benefit-to-Cost Rating: **HIGH**

### 5.1.6 Software Fault Tree Analysis

It is possible for a system to meet requirements for a correct state and to also be unsafe. Complex systems increase the chance that unsafe modes will not be caught until the system is in the field. Fault Tree Analysis (FTA) is one method that focuses on how errors, or even normal functioning of the system, can lead to hazards.

The requirements phase is the time to perform a preliminary software fault tree analysis (SFTA). This is a "top down" analysis, looking for the causes of presupposed hazards. The top of the "tree" (the hazards) must be known before this analysis is applied. The Preliminary Hazard Analysis (PHA) is the primary source for hazards, along with the Requirements Criticality Analysis and other analyses described above.

The results of a fault tree analysis is a list of failures, or combination of failures, that can lead to a hazard. Some of those failures will be in software. At this top level, the failures will be very general (e.g., "computer fails to raise alarm"). When this analysis is updated in later phases, the failures can be assigned to specific functions or modules.

FTA was originally developed in the 1960's for safety analysis of the Minuteman missile system. It has become one of the most widely used hazard analysis techniques. In some cases FTA techniques may be mandated by civil or military authorities.

Most of the information presented in this section is extracted from Leveson et al. [13,14].

FTA is a complex subject, and is described further in Appendix B.

Benefit-to-Cost Rating: **MEDIUM**

### 5.1.7 Conclusion

Using the above techniques will provide some level of assurance that the software requirements will result in a design which satisfies safety objectives. The extent to which the techniques should be used depends on the degree of criticality of the system and its risk index, as discussed in Section 3. Some of these analyses will need to be updated during design and code phases as the system becomes more defined and details are specified.

The output of the Software Safety Requirements Analyses (SSRA) are used as input to follow-on software safety analyses. The SSRA is presented at the Software Requirements Review (SSR)/Software Specification Review (SSR) and system-level safety reviews. The results of the SSRA shall be provided to the ongoing system safety analysis activity.

Having developed and analyzed the baseline software requirements set, an architectural design is developed as per *Section 4.3 Architectural Design Phase*. *Section 5.2 Architectural Design Analysis* below describes analysis tasks for the architectural design.

## 5.2 Architectural Design Analysis

The software architectural design process develops the high level design that will implement the software requirements. All software safety requirements developed in *Section 4.2.1* 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 (SCCSC's) are identified.

Safety analyses are performed on the architectural design to identify potential hazards and to define and analyze SCCSC's (Safety Critical Computer Software Components). Early test plans are reviewed to verify incorporation of safety related testing. Software safety analyses from *Section 5.1* are updated as appropriate during this phase.

Analyses included in the Architectural Design Phase are:

- Update Criticality Analysis
- Conduct Hazard Risk Assessment
- Analyze Architectural Design
- Interdependence Analysis
- Independence Analysis
- Update Timing/Throughput/Sizing Analysis
- Update Software Fault Tree Analysis
- Perform preliminary Software Failure Modes and Effects Analysis

### 5.2.1 Update Criticality Analysis

At this stage of development, the software functions begin to be allocated to modules and components. Software for a system, while often subjected to a single development program, actually consists of a set of multipurpose, 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. Each module or component that implements a safety critical requirement must now be assigned a criticality index, based on the criticality analysis (See *Section 5.1.2 Requirements Criticality Analysis*). The safety activity in this phase is to relate the identified hazards from the following analyses to the Computer Software Components (CSC's) that may affect or control the hazards.

| Analysis | Guidebook Section |
|---|---|
| Preliminary Hazard Analysis (PHA) | 2.4 |
| Software Subsystem Hazard Analysis | 2.5 |
| Software Safety Requirements Analysis | 5.1 |

Develop a matrix that lists all SCCSC's and the safety requirements they relate to. Include any modules that can affect the SCCSC's as well. This would include modules that write to data in memory shared with the SCCSC's, or that provide information to the safety critical modules. The designation "Safety Critical Computer Software Component" (SCCSC) should be applied to any module, component, subroutine or other software entity identified by this analysis.

### 5.2.2 Conduct Hazard Risk Assessment

Once SCCSC's have been identified, system hazard risk assessment is done to prioritize them. Not all SCCSC's warrant further analysis beyond the architectural design level, nor do all warrant the same depth of analysis. System risk assessment of hazards, as described in the NHB 1700 series of documents, consists of ranking hazards by severity level versus probability of occurrence. This high-severity/high probability hazards are prioritized higher for analysis and corrective action than low-severity/low probability hazards.

While *Sections 5.1.2 Requirements Criticality Analysis* and *5.2.1 Update Criticality Analysis* simply assign a Yes or No to whether each component is safety critical, the Risk Assessment process takes this further. Each SCCSC's is prioritized for analysis and corrective action according to the five levels of Hazard Prioritization ranking given previously in *Table 2-2 Hazard Prioritization - System Risk Index*.

Determination of the severity and the probability of the hazards is sometimes a source of contention between the safety group and the project. It is best to sit down and work out any disagreements at an early stage. Getting the software development group's "buy in" on what is truly safety critical is vital. Software developers may ignore what they do not see as important. Getting everybody on one "side" early prevents the problem of forcing the project to add safety code or testing later in the development cycle.

Benefit-to-Cost Rating: **HIGH**

### 5.2.3 Analyze Architectural Design

Okay, you've got your list of SCCSC's that will be further analyzed. Next you analyze the Architectural (high level) design of those components to ensure all safety requirements are specified correctly and completely. In addition, review the Architectural Design, looking for places and conditions that lead to unacceptable hazards. This is done by postulating credible faults/failures and evaluating their effects on the system.

Consider the following types of events and failures:

- input/output timing
- multiple event
- out-of-sequence event
- failure of event
- wrong event
- inappropriate magnitude
- incorrect polarity
- adverse environment
- deadlocking in a multitasking system
- hardware failures

Formal Inspections (see *4.2.5 Formal Inspections of Specifications*), design reviews and prototype, animation or simulation augment this process.

#### *5.2.3.1 Design Reviews*

Design reviews are conducted to verify that the design meets the requirements. Often the reviews are formal (Preliminary Design Review (PDR) and Critical Design Review (CDR)) and for the whole system. Separate software PDR's and CDR's may be held, or the software may be a part of the system review. In all cases, software safety should be addressed. Does the design meet all the applicable software safety requirements? Does the design use "best practices" to reduce potential errors? Are safety critical modules properly separated from the rest of the software? This is the time to make design changes, if necessary, to fulfill the safety requirements.

Applicability matrices, compliance matrices, and compliance checklists are resources which can be used to assist in completing this task.

Output products from the reviews are engineering change requests, hazard reports (to capture design decisions affecting hazard controls and verification) and action items.

Benefit-to-Cost Rating: **HIGH**

#### *5.2.3.2 Prototype/Animation/Simulation*

When questions exist about the ability of the software to meet a requirement, a prototype or simulator is often used to find the answer. This software is "quick and dirty" and used only for determining if the system can do what it needs to do. Prototypes may also be used to get the customer's input into a user interface. Test cases that exercise crucial functions can be developed along with the prototype. Just run the tests and observe the system response. If the requirements can not be met, then they must be modified as appropriate.

Documented test results can confirm expected behavior or reveal unexpected behavior. Keep in mind, however, that the tests are of a prototype or simulation. The behavior of the real software may differ.

Benefit-to-Cost Rating: **MEDIUM**

### 5.2.4 Interface Analysis

#### *5.2.4.1 Interdependence Analysis*

Examine the software design to determine the interdependence among Computer Software Components (CSC's), modules, tables, variables, etc. Elements of software which directly or indirectly influence SCCSC's are also identified as SCCSC's, and as such should be analyzed for their undesired effects. For example, shared memory blocks used by an SCCSC and another CSC (safety critical or not). The inputs and outputs of each SCCSC are inspected and traced to their origin and destination.

Benefit-to-Cost Rating: **MEDIUM**

#### *5.2.4.2 Independence Analysis*

The safety critical CSC's (SCCSC's) should be independent of non-critical functions. Independence Analysis is a way to verify that.. Those CSC's that are found to affect the output of SCCSC's are designated as SCCSC's themselves. Areas where FCR (Fault Containment Region) integrity is compromised are identified.

To perform this analysis, map the safety critical functions to the software modules, and then map the software modules to the hardware hosts and FCR's. All the input and output of each SCCSC should be inspected. Consider global or shared variables, as well as the directly passed parameters. Consider "side effects" that may be included when a module is run. If a non-critical CSC modifies an SCCSC, either directly, by violation of an FCR or indirectly through shared memory, then it becomes an SCCSC itself.

Resources used in this analysis are the definition of safety critical functions (MWF and MNWF) that need to be independent (from *Section 4.2.2.2 Hazardous Commands*), design descriptions, and data and flow diagrams.

Design changes to achieve valid FCR's and corrections to SCCSC designations may be necessary.

At this point, some bottom-up analyses can be performed, like a Software FMEA. 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.

Benefit-to-Cost Rating: **MEDIUM**

### 5.2.5 Update Timing, Throughput, and Sizing Analysis

Now that initial design issues have been decided, review the Timing, Throughput, and Sizing analysis and update it as appropriate.

### 5.2.6 Update Software Fault Tree Analysis

The preliminary software fault tree generated in the requirements phase can now be expanded. Broad functions can now be specified to some module, at least at a high level. In addition, the system is now known to a greater depth. Failures that were credible during the requirements phase may no longer be possible. Additional causes for the top hazard may be added to the tree.

### 5.2.7 Formal Inspections of Architectural Design Products

The process of Formal Inspection begun in previous requirements phase (e.g. *Section5.1.4 Formal Inspections*) should continue. The preliminary (architectural) design should go through a formal review process. Create new checklists that are appropriate to the design products. Include "lessons learned" from the requirements phase.

Benefit-to-Cost Rating: **HIGH**

### 5.2.8 Formal Methods and Model Checking

During the architectural design phase, the requirements specified are "converted" into a high-level design. Standard structured methodologies (object oriented or functional) are used to create the preliminary design.

If using formal methods (formal specification), the requirements specification is "fleshed out" with increasing detail. In the example in [40], the preliminary design consisted of the state description (state variables and state transitions) expressed in the formal specification language.

The formal method "design" or model may be the complete architectural design, or it may be created in parallel with a "normal" design process. If using the parallel approach (standard software development life cycle and formal methods on separate tracks, usually with separate teams), it is important to verify that the designs created by the development team formally match those of the formal methods team.

Formal Methods Benefit-to-Cost Rating: **LOW**

Model Checking Benefit-to-Cost Rating: **MEDIUM**

## *5.3 Detailed Design Analysis*

During the Detailed Design phase, the software artifacts (design documents) are greatly enhanced. This additional detail now permits rigorous analyses to be performed. Detailed Design Analyses can make use of artifacts such as the following: 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.

- Design Logic Analysis
- Design Data Analysis
- Design Interface Analysis

- Design Constraint Analysis
- Rate Monotonic Analysis
- Dynamic Flowgraph Analysis
- Markov Modeling
- Measurement of Complexity
- Selection of Programming Languages
- Formal Methods and Safety-Critical Considerations
- Requirements State Machines
- Formal Inspections
- Updates to previous analyses

Description of each technique is provided below. Choice of techniques in terms of cost versus effectiveness was discussed earlier in *Section 3.2.3.3 Tailoring the Effort*.

### 5.3.1 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.

Benefit-to-Cost Rating: **MEDIUM**

### 5.3.2 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 and radiation affects on memory should be reviewed in conjunction with system safety.

Benefit-to-Cost Rating: **MEDIUM**

### 5.3.3 Design Interface Analysis

Design interface analysis verifies the proper design of a software component's interfaces with other components of the system. The interfaces can be with other software components, with hardware, or with human operators. This analysis will verify that the software component's interfaces, especially the control and data linkages, have been properly designed. Interface requirements specifications (which may be part of the requirements or design documents, or a separate document) are the sources against which the interfaces are evaluated.

Interface characteristics to be addressed should include interprocess communication methods, data encoding, error checking and synchronization.

The analysis should consider the validity and effectiveness of checksums, CRC's, and error correcting code. The sophistication of error checking or correction that is 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.
- Message used by sender to indicate its current state is not understood by the receiving process.
- Interface deadlock prevents data transfer (e.g., receiver ignores or cannot recognize "Ready To Send").
- User reads data from wrong address.
- Data put in shared memory by one process is in "big endian" order, while the process that will use it is expecting "little endian".
- In a language such as 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).

Benefit-to-Cost Rating: **MEDIUM**

### 5.3.4 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:

- update timing, throughput and sizing constraints as per *Section 5.1.5 Timing, Throughput And Sizing Analysis*
- 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 [21]
- 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.

Design constraint analysis evaluates the ability of the software to operate within these constraints.

Benefit-to-Cost Rating: **HIGH**

### 5.3.5 Design Functional Analysis

This analysis ensures that each safety-critical software requirement is covered and that an appropriate criticality level is assigned to each software element. Tracing the safety requirements throughout the design, code, and tests is vital to making sure that no requirements are lost, that safety is "designed in", that extra care is taken during the coding phase, and that all safety requirements are tested. A safety requirement traceability matrix is one way to implement this analysis.

Benefit-to-Cost Rating: **HIGH**

### 5.3.6 Software Element Analysis

Each software element that is **not** safety critical is examined to assure that it cannot cause a hazard. When examining a software element, consider, at a minimum, the following ideas:

- Does the element interface with hardware that can cause a hazard?
- Does the element interface with safety-critical software elements?
- Can the software element tie up resources required by any safety-critical components?
- Can the software element enter an infinite loop?
- Does the software element use the same memory as safety critical data, such that an error in addressing could lead to overwriting the safety critical information?
- Is priority inversion or deadlocking a possibility, and can it impact a safety critical task?
- Can the software element affect the system performance or timing in a way that would affect a safety critical component?
- Does the software element call any functions also called by a safety critical component? Can it change any aspect of that function, such that the safety critical component will be affected?
- Is the software element on the same platform and in the same partition as a safety critical component?

### 5.3.7 Rate Monotonic Analysis

Rate Monotonic Analysis (RMA) is a mathematical method for predicting, a priori, whether a system will meet its timing and throughput requirements when the system is operational. RMA works on systems that use static priority for the tasks. This includes nearly all commercial operating systems. RMA requires that timing information can be measured or reliably estimated for each task. For systems with hard real-time deadlines (deadlines that absolutely **must** be met), RMA is a valuable tool.

For further details on this technique, refer to publications by Sha and Goodenough, References [22] and [25]. A case study using RMA when integrating an intelligent, autonomous software system with Flight software, as part of the NASA New Millennium project, is discussed in reference [47].

Benefit-to-Cost Rating: **LOW**

### 5.3.8 Dynamic Flowgraph Analysis

Dynamic Flowgraph Analysis is a new technique that is not yet widely used and still in the experimental phase of evaluation. It does appear to offer some promise, building on the benefits of conventional *5.1.6 Software Fault Tree Analysis (SFTA)*.

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 causal relationships between physical variables and temporal characteristics of the execution of software modules.

Further description of this method is given in the paper by Garrett, Yau, Guarro and Apostolakais [20].

Benefit-to-Cost Rating: **LOW**

### 5.3.9 Markov Modeling

Markov Modeling techniques were developed for complex systems and some analysts have adapted these techniques for software intensive systems. They can provide reliability, availability and maintainability data. They model probabilistic behavior of a set of equipment as a continuous time, homogeneous, discrete state Markov Process. The statistical probability of the system being in a particular macro state can be computed. These statistics can be translated into a measure of system reliability and availability for different mission phases.

However, attempting to apply these types of reliability modeling techniques to software is questionable because, unlike hardware, software does not exhibit meaningful (random) failure statistics. Also, unlike hardware component failures, software failures are often not independent. Software errors depend on indeterminate human factors, such as alertness or programmer skill level.

Benefit-to-Cost Rating: **LOW**

### 5.3.10 Measurement of Complexity

The complexity of the software should be evaluated, because the level of complexity can affect the understandability, reliability and maintainability of the code. Highly complex data and command structures are difficult, if not impossible, to test thoroughly. Complex software is difficult to maintain, and updating the software may lead to additional errors. 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. When highly complex data and command structures are necessary, look at techniques for avoiding a high a level of programming interweaving.

Linguistic and structural metrics exist for measuring the complexity of software, and are discussed below. The following references provide a more detailed discussion of and guidance on the techniques.

1. "Software State of the Art: selected papers", Tom DeMarco, Dorset House, NY, 2000.
2. "Black-Box Testing : Techniques for Functional Testing of Software and Systems", Boris Beizer, Wiley, John & Sons Inc., 1995
3. "Applying Software Metrics", Shari Lawrence Pfleeger and Paul Oman, IEEE Press, 1997
4. "A Framework of Software Measurement", Horst Zuse, Walter deGruyter, 1998

5. "Metrics and Models in Software Quality Engineering", Stephen Kan, Addison Wesley, 1995
6. "Object-Oriented Metrics Measures of Complexity", Brian Henderson-Sellers, Prentice Hall,1996
7. "Software Metrics : A Rigorous and Practical Approach", Norman E. Fenton, PWS Publishing, 1998
8. "Function Point Analysis: Measurement Practices for Successful Software Projects", David Garmus and David Herron, Addison, 2000

Linguistic measurements assess some property of the text without regard for the contents (e.g., lines of code, function points, 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.

Apply one or more complexity estimation techniques, such as McCabe or Halstead, to the design products. 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. The references above give detailed instructions on what to look for when estimating complexity.

Resources used by these techniques are the detailed design, high level language description, any source code or pseudocode, and automated complexity measurement tool(s). The output products are the 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 provide these metrics. The level and type of complexity can indicate areas where further analysis, or testing, may be warranted. Do not take the numbers at face value, however! Sometimes a structure considered to be highly complex (such as a case statement) may actually be a simpler, more straight forward method of programming and maintenance, thus decreasing the risk of errors.

Benefit-to-Cost Rating: **HIGH**

#### *5.3.10.1 Function Points*

The most common size metric used is software lines of code (SLOC). While easy to measure, this metric has some problems. The lines of code it takes to produce a specific function will vary with the language – more lines are needed in assembly language than in C++, for instance. Counting the lines can only be done once the code is available, and pre-coding estimates are often not accurate. When SLOC is used to calculate other measures, such as defect rates (number of defects per SLOC), the numbers get worse as the programming improves!

Function Points are an alternative measurement to SLOC that focuses on the end-user, and not on the technical details of the coding. Function Point Analysis was developed by Allan Albrecht of IBM in 1979, and revised in 1983. The FPA technique quantifies the functions contained within software in terms which are meaningful to the software users. The measure relates directly to the requirements which the software is intended to address. It can therefore be readily applied throughout the life of a development project, from early requirements definition to full operational use.

The function point metric is calculated by using a weighted count of the number of the following elements:

- **User inputs** provide application-oriented data to the software.
- **User outputs** provide application-oriented information to the user. This includes reports, screens, error messages, etc. Individual data items within a report are not counted separately.
- **User inquiries** are an on-line input that results in the generation of some immediate software response in the form of an on-line output. Typing a question in a search engine would be an inquiry.
- **Files** include both physical and logical files (groupings of information).
- **External interfaces** are all machine readable interfaces used to transmit information to another system.

The weighting factors are based on the complexity of the software.

"Function Point Analysis: Measurement Practices for Successful Software Projects", by David Garmus and David Herron, provides information on calculating and using function points. The International Function Point Users Group (IFPUG, http://www.ifpug.org/) supports and promotes the use of function points.

#### 5.3.10.2 Function Point extensions

Function points (described above in *5.3.10.1 Function Points*) are business (database, transaction) oriented. Extensions are needed for systems and engineering software applications, such as real-time, process control, and embedded software.

Feature points are one such extension. This metric takes into account algorithmic complexity. A feature point value is the sum of the weighted function point factors and the weighted algorithm count. Algorithms include such actions as inverting a matrix, decoding a bit string, or handling an interrupt. Feature points were developed in 1986 by Capers Jones.

The "3D function point" was developed by Boeing for real-time and embedded systems. The Boeing approach integrates the data, functional, and control dimensions of a software system. The data dimension is essentially the standard function point. The functional dimension counts the transformations, which are the number of internal operations to transform the input data into output data. The control dimension is measured by counting the number of transitions between states.

### 5.3.11 Selection of  Programming Languages

When choosing a programming language, many factors are important. The memory size and execution speed of an algorithm developed in a particular language is one factor. The existence of tools (compiler, integrated development environment, etc.) that support the language for the specified processor and on the development platform, and the availability of software engineers who have training and experience with the language are also important. When developing safety critical applications or modules, however, the "safeness" of the programming language should be a high priority factor.

A "safe" programming language is one in which the translation from source to object code can be rigorously verified. Compilers that are designed to use safe subsets of a programming language are often certified, guaranteeing that the object code is a correct translation of the source code. In a more general sense, a "safe" language is one that enforces good programming practices, and that finds errors at compile time, rather than at run time. Safe languages have strict data types, bounds checking on arrays, and discourage the use of pointers, among other features.

*Section 6 SOFTWARE DEVELOPMENT ISSUES* contains a technical overview of safety-critical coding practices for developers and safety engineers and detailed discussion of specific programming languages. Many of the coding practices involve restricting the use of certain programming language constructs. Reading knowledge of a high level language (structural or object-oriented) is required to understand the concepts that are being discussed.

*Section 6* will provide an introduction on the criteria for evaluating the risks associated with the choice of a particular programming language. Some are well suited for safety-critical applications, and therefore engender a lower risk. Others are less "safe" and, if chosen, require additional analysis and testing to assure the safety of the software. Where appropriate, safe subsets of languages will be described. Common errors ("bugs") in software development are also included.

When choosing a language, consider the language "environment" (compiler, Integrated Development Environment (IDE), debugger, etc.) as well. Is the compiler certified? Is there a list of known defects or errors produced by the compiler? Does the code editor help find problems by highlighting or changing the color of language-specific terms? Does the compiler allow you to have warnings issued as errors, to enforce conformance? Is there a debugger that allows you to set break points and look at the source assembly code?

No programming language is guaranteed to produce safe software. The best languages enforce good programming practices, make bugs easier for the compiler to find, and incorporate elements that make the software easier to verify. Even so, the "safeness" and reliability of the software depend on many other factors, including the correctness of the requirements and the proper implementation of the requirements by the design. Humans are involved in all aspects of the process, and we are quite capable of subverting even the "safest" of languages. Select a language based on a balance of all factors, including safety.

Benefit-to-Cost Rating: **HIGH**

### 5.3.12 Formal Methods and Model Checking

During the detailed design phase, the architectural (preliminary) design is filled in with all the necessary details to fully specify the system. Standard structured methodologies (object oriented or functional) are used to create the detailed design.

If using formal methods (formal specification), the next level of detail is added to the system specification. In the example in [40], the detailed design added more information about the system, all expressed in the formal specification language.

The formal method "design" or model may be the detailed design, or it may be created in parallel with a "normal" design process. If using the parallel approach (standard software development life cycle and formal methods on separate tracks, usually with separate teams), it is important to verify that the designs created by the development team formally match those of the formal methods team.

Formal Methods Benefit-to-Cost Rating: **LOW**

Model Checking Benefit-to-Cost Rating: **MEDIUM**

### 5.3.13 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 (see Section 4.3.2.1) and Formal Methods (See *Section 4.2.3 Formal Methods - Specification Development*)).

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.

Details on using Requirements State Machines are given in Appendix D.

Benefit-to-Cost Rating: **LOW**

### 5.3.14 Formal Inspections of Detailed Design Products

The new software artifacts (detailed design) should be formally inspected. Update the inspection checklist with lessons learned, and add items appropriate to the detail now available. Pseudo-code or prototype code are often available for review at this stage.

Benefit-to-Cost Rating: **HIGH**

### 5.3.15 Software Failure Modes and Effects Analysis

A "bottom up" analysis technique is the FMEA (Failure Modes and Effects Analysis). It looks at how each component could fail, how the failure propagates through the system, and whether it can lead to a hazard. This technique requires a fairly detailed design of the system. In the Architectural Design phase, only a preliminary Software FMEA can be completed.

A Software FMEA uses the methods of a standard (hardware) FMEA, substituting software components for hardware components in each case. A widely used FMEA procedure is MIL-STD-1629, which is based on the following steps:

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.

More detailed information on SFMEA (Software Failure Modes and Effects Analysis) can be found in Appendix C.

Benefit-to-Cost Rating: **MEDIUM**

#### 5.3.16 Updates to Previous Analyses

Now that the detailed design is completed, the SFTA and Timing/Throughput/Sizing analyses should be updated. Enough detail exists to complete the Software Fault Tree Analysis, though it should be reviewed in the later phases, since things often change during coding.

The Criticality Analysis should be updated. Modules may have been subdivided into smaller components or rearranged within the design. The SCCSC's (Safety Critical Computer Software Component) should be reviewed for changes. This is also the time to look for additional modules that should be classified as safety critical.

## *5.4 Code Analysis*

Code analysis verifies that the coded program correctly implements the verified design and does not violate safety requirements. Having the code permits real measurements of size, complexity and resource usage of the software. These quantities could only be estimated during the design phase, and the estimates were often just educated guesses.

The results of code analyses may lead to significant redesign if the analyses show that the "guess" were wildly incorrect. However, the main purpose is to verify that the code meets the requirements (traceable through the design) and that it produces a safe system..

Code Analyses include the following:

- Code Logic Analysis
- Code Data Analysis
- Code Interface Analysis
- Measurement of Complexity

- Code Constraint Analysis
- Formal Code Inspections, Checklists, and Coding Standards
- Formal Methods
- Unused Code Analysis
- Interrupt Analysis
- Update to Timing/Throughput/Sizing analysis
- Update to Software Failure Modes and Effects Analysis

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.

Many of these analyses will be undergoing their second iteration, since they were 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 complete 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 to detect problems, such as erroneous jumps. The code may need to be run on an instruction set emulator which can monitor the code from the outside, without adding the instrumentation, if such problems are suspected.

### 5.4.1 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. This analysis is conducted by performing logic reconstruction, equation reconstruction and memory decoding. For complex software, this analysis is applied to all safety critical components (SCCSC's). Other software components may be analyzed if they are deemed important to the system functionality.

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.

Benefit-to-Cost Rating: **LOW**

### 5.4.2 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.

Benefit-to-Cost Rating: **MEDIUM**

### 5.4.3 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 included in the "Design Constraint Analysis" discussed in *Section 5.4.5 Update Design Constraint Analysis*.

Benefit-to-Cost Rating: **HIGH**

~~Rating:**MEDIUM**~~

### 5.4.4 Update Measurement of Complexity

Now that code exists, the complexity metrics can be recalculated. Complex code should be evaluated by a human. Some logic structures (such as case statements) may be flagged as complicated, when they really improve the comprehensibility of the software.

Complex software increases the number of errors, while making it difficult to find them. This makes the software more likely to be unstable, or suffer from unpredictable behavior. Reducing complexity is generally a good idea, whenever possible. Modularity is a useful technique to reduce complexity. Encapsulation can also be used, to hide data and functions from the "user" (the rest of the software), and prevent their unanticipated execution.

Software flagged as complex should be analyzed in more depth, even if it is not safety critical. These modules are prime candidates for formal inspections and the logic/data/constraint analyses.

### 5.4.5 Update Design Constraint Analysis

The criteria for design constraint analysis applied to the detailed design in *Section 5.3.4 Design Constraint Analysis*, 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 (see *section 5.4.10*) to ensure that computing resources and memory available are adequate for safety critical functions and processes.

Underflows/overflows in certain languages (e.g., A~~DAA~~da) 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 automatic recovery, querying the user (retry, etc.), or putting the system into a safe state.

### 5.4.6 Formal Code Inspections, Checklists, and Coding Standards

Formal Inspections, introduced in *Section 4.2.5– Formal Inspections*, should be performed on the safety critical software components, at a minimum. Consider doing Formal Inspections on other complex or critical software modules. Formal Inspections are one of the best methodologies available to evaluate the quality of code modules and program sets. Having multiple eyes and minds review the code, in a formal way, makes errors and omissions easier to find.

Checklists should be developed for use during formal inspections to facilitate inspection of the code. They should include:

- requirements information for modules under review
- design details for modules under review
- coding standards (subset/most important)
- language-independent programming errors
- language-specific programming errors

Appendix E contains a sample checklists of common errors, both independent of language and language specific.

Coding standards are based on style guides and safe subsets of programming languages. They should have been specified during the design phase (or earlier), and used throughout the coding (implementation) phase.

Benefit-to-Cost Rating: **HIGH**

### 5.4.7 Applying Formal Methods to Code

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 may be used to "extract" the logic of the code, and then compare the logic to the formal requirements expressions from the Formal Methods.

Formal Methods Benefit-to-Cost Rating: **LOW**

Model Checking Benefit-to-Cost Rating: **LOW**

### 5.4.8 Unused Code Analysis

A common real world coding error is generation of code which 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 analysis 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 to ensure that every part of the code is eventually exercised (tested) at some time, within all possible operating modes of the system.

Benefit-to-Cost Rating: **MEDIUM**

### 5.4.9 Interrupt Analysis

Interrupt Analysis looks at how interrupts are used by the software. The effect of interrupts on program flow and data corruption are the primary focus of this analysis. Can interrupts lead to priority inversion and prevent a high priority or safety critical task from completing? If interrupts are locked out for a period of time, can the system stack incoming interrupts to prevent their loss? Can a low-priority process interrupt a high-priority process and change critical data?

When performing interrupt analysis, consider the following areas of the code:

- **Program segments/modules where interrupts are inhibited (locked out).** Look at how long the interrupts are inhibited and whether the system can buffer interrupts for this period of time. The expected and maximum interrupt rates would be needed to check for buffering capacity. Identify impacts from lost interrupts. Look for possible infinite loops.
- **Re-entrant code.** Re-entrant code is designed to be interrupted without loss of state information. Check that re-entrant modules have sufficient data saved for each interruption, and that the data and system state are correctly restored. Make sure that modules that need to be re-entrant are implemented as such.
- **Interruptible code segments/modules.** Make sure that timing-critical areas are protected from interrupts, if a delay would be unacceptable. Check for sequences of instructions that should not be interrupted.
- **Priorities.** Look over the process priorities of the real-time tasks. Verify that time-critical events will be assured of execution. Also consider the operator interface. Will the interface update with important or critical information in a timely fashion?
- **Undefined interrupts.** What happens when an undefined interrupt is received? Is it ignored? Is any error processing required?

Benefit-to-Cost Rating: **HIGH**

### 5.4.~~9~~ 10 Final Timing, Throughput, and Sizing Analysis

With the completion of the coding phase, the timing, throughput, and sizing parameters can be measured. The size of the executable module (storage size) is easily measured, as is the amount of memory space used by the running software. Special tests may need to be run to determine the maximum memory used, as well as timing and throughput parameters. Some of these tests may be delayed until the testing phase, where they may be formally included in functional or load/stress tests. However, simple tests should be run as soon as the appropriate code is stable, to allow verification of the timing, throughput, and sizing requirements. The earlier a problem is discovered, the easier and cheaper it is to fix.

Benefit-to-Cost Rating: **HIGH**

### 5.4.1~~01~~1 Program Slicing

When you get a wrong answer from your software, program slicing can help. It is a technique to trace back through the program and show you all, and only, the statements that affect the variable you are interested in. In a large, complex program, slicing can cut through the extraneous (to the problem) information, focusing in on the statements of interest.

Slicing has been mainly used in debugging (finding the source of an error) and reengineering (pulling out part of a program to create a new program). It can also be used to check the "lineage" of any safety-critical data. Using a slicing tool to pull out all the statements that affect the safety-critical variable, and then examining the results, may point to errors or unexpected interactions with other, non-critical data. You may even wish to do a Formal Inspection on the sliced code.

Slicing comes in two forms: static and dynamic. Static slicing, introduced in 1982, is done on the source code (compile-time). Originally, it had to be an executable subset of the program, though that is not always necessary. Static slicing shows every statement that may have an impact of the variable of interest. Dynamic slicing first appeared around 1988, and works on programs as they operate (run-time). While static slicing shows all the statements that *may* affect the variable of interest, dynamic slicing shows only those that *do* affect the variable as the software is exercised.

Program slicing by hand would be a tedious job. Tools are beginning to be available for a variety of languages.

Benefit-to-Cost Rating: **MEDIUM**

### 5.4.1~~11~~2 Update Software Failure Modes and Effects Analysis

Review any changes to the design that developed during the coding phase. Often creating the actual coding will point out problems with the design, or elements that are missing. If the design was modified during this phase, review the Software FMEA and make any updates as necessary.

## *5.5 Test Analysis*

Two sets of analyses should be performed during the testing phase:

1. analyses before the fact to ensure validity and completeness of tests
2. analyses of the test results

Testing (as opposed to analyses) was discussed in *Section 4.6 Software Integration and Test* and will not be covered here. Analysis before the fact should, as a minimum, consider test coverage for safety critical Must-Work-Functions and Must-Not-Work-Functions.

### 5.5.1 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 which attempts to calculate the probability that a system with a certain failure probability will pass a given number of tests. This is discussed in "Evaluation of Safety Critical Software", David L. Parnas, A. John van Schouwen and Shu Po Kwan, Communications of ACM, June 1990 Vol 33 Nr 6 [43].

“White box” testing assumes that the tester has knowledge of the internal workings of the module or program to be tested. It is usually used with unit (module, file, etc.) testing. Examples of “white box” tests are path tests (all paths through the code), branch testing (verify each branch taken), checking each assignment to memory, and verifying that each statement is executed at least once.

“Black box” testing assumes that the tester has no knowledge of what happens inside the software. Only the inputs and outputs are accessible. Black box tests usually are functional tests that exercise the “normal” operations of the software. In addition, “off nominal” tests are done to verify the software operates correctly with erroneous input.

Statistical methods such as Monte Carlo simulations can be useful in planning "worst case" credible scenarios to be tested.

Test coverage analysis is best if done prior to the start of testing. At a minimum, analysis should be done to verify that the planned tests cover all paths through the program, that all branches are exercised, and that each statement is executed at least once. Verify that boundary conditions are tested for all inputs, as well as nominal and erroneous input values.

Benefit-to-Cost Rating: **HIGH**

### 5.5.2 Formal Inspections of Test Plan and Procedures

Test plans should be created early in the software development lifecycle. Once the requirements are known, a test plan that addresses how the requirements will be verified can be developed. Functional testing, acceptance testing, and off-nominal testing should be included, at a minimum.

Test procedures are the specifics of what is being tested, how to conduct the test, and what the expected results are. The procedures should reference the specific requirements verified by the test. Places to check off the steps should be provided. Important sections, including safety verification steps, have signature blocks for witnesses.

The test plan and test procedures should be reviewed by the safety engineer for a project at the safety minimum level. For higher safety levels, the plan and procedures should undergo formal inspections. (Formal inspections are discussed in *Section 4.2.5 Formal Inspections*.)

Benefit-to-Cost Rating: **HIGH**

### 5.5.3 Reliability Modeling

Software reliability contributes to software safety. If the software operates for a long period of time without a failure, then it will be "safe" for that period of time, assuming that an operational (non-failed) mode cannot lead to a hazard.

According to the ANSI standard, software reliability is defined as "the probability of failure-free operation of a computer program for a specified time in a specified environment." Reliability modeling is the process of determining what that the probability and the specified time are. The primary goal of software reliability modeling is to answer the question: Given a system, what is the probability that it will fail in a given time interval, or, what is the expected duration between successive failures?

Software reliability models come in several flavors. **Prediction models** attempt to predict what the reliability of the system will be when it is completed. Prediction models may be developed as early as the requirements phase, or in the design or implementation phase. The questions that a predictive model tries to answer are: Can we reach the reliability goals or requirements? How reliable will the system truly be? Resources that prediction models may use are the failure data of the current system (if it is in test), metrics from the software development process, and failure data from similar systems.

**Estimation models** evaluate the current software, usually during the test phase. Based on defects found and other factors, the models attempt to estimate how many defects still remain or the time between failures, once the software is in operation. Estimation models include reliability growth models, input domain models, and fault seeding models.

Over 40 varieties of prediction and estimation software reliability models exist. The accuracy of the models varies with the model, the project, and the expertise of the analyst.

Benefit-to-Cost Rating: **LOW**

#### *5.5.3.1 Criteria for Selecting a Reliability Model*

- **Model validity.** How good is the model at accurately measuring the current failure rate? At predicting the time to finish testing with associated date and cost? At predicting the operational failure rate?

- **Ease of measuring parameters.** What are the cost and schedule impacts for data (metrics) collection? How physically significant are the parameters to the software development process?

- **Quality of assumptions.** How close are the model assumptions to the real world? Is the model adaptable to a special environment?
- **Applicability.** Can the model handle program evolution and change in test and operational environments?
- **Simplicity.** Is the model simple in concept, data collection, program implementation, and validation?
- **Insensitivity to noise.** Is the model insensitive to insignificant changes in input data and parameters, without losing responsiveness to significant differences?
- **Usefulness**. Does the model estimate quantities that are useful to project personnel?

#### *5.5.3.2 Issues and Concerns*

Ideally, one simple reliability model would be available, with great tool support, that would easily and accurately predict or estimate the reliability of the software under development. The current situation, however is that

- Over 40 models have been published in the literature.
- The accuracy of the models is variable.
- You can't know ahead of time which model is best for your situation.

Some aspects of the models that are a cause for concern are:

- How accurate is the data collected during testing? How easy is it to collect that data?
- Models are primarily used during the testing phase, which is late in the development cycle.
- Estimation of parameters is not always possible, and sometimes it is mathematically intractable.
- Reliable models for multiple systems have not been developed.
- There is no well-established criteria for model selection.

#### *5.5.3.3 Tools*

In the last decade, tools have become available to aid in software reliability modeling. Most of the established models have tools that support them. Resources for information on available tools are:

1. "Applying Software Reliability Engineering in the 1990s", W. Everett, S. Keene, and A. Nikora, *IEEE Transaction on Reliability*, Vol. 47, No. 3-SP, September 1998
2. "Software Reliability Engineering Study of a Large-Scale Telecommunications Software System", Carman et. al., *Proc. 1995 International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995, pp. 350-.
3. http://rac.iitri.org/DATA/RMST/rel_model.html Links to many tools

4. MEADEP tool. http://www.meadep.com/
5. Reliability Modeling, Developed by C. Chay and W. Leyu, http://www.icaen.uiowa.edu/~ankusiak/reli.html

#### 5.5.3.4 Dissenting Views

Not everyone agrees that software reliability modeling is a useful technique. Some are concerned about the applicability of the models to real-world situations. Most models assume random failures, but is that true? The models do not usually address the fact that fixing a failure may add other errors to the software. The fact that software is often "unique" (one-of-a-kind) makes statistics about the error rates difficult to apply across a broad spectrum of programs. Unlike hardware, you are dealing with one part, not one of many identical units.

A ~~critic~~ critique of software reliability modeling is found in [46]. The authors assert that current models do not adequately deal with these factors:

- **Difficulties in estimating Operational Profiles**, such as the input distribution (what is input, when, in what order). New software may have no history or customer base to use to determine typical operations. It is non-trivial to determine how the system will be used, but such an operational profile is a key element for most reliability models.
- **Problems with reliability estimation.** Inadequate test sets, failure to exercise each feature in testing, and skewed operational profile (critical functions may not be part of the "typical" profile) make reliability difficult to estimate accurately.
- **Reliability estimation occurs near the end of development.** Individual component reliability is not known, just for the full system. There is no information to feed back that may lead to process improvement and better reliability in future projects.
- **Saturation effects lead to reliability overestimation.** Most testing techniques reach a saturation point past which they are unable to find defects. These limits can lead to an overestimate of the software reliability.

#### 5.5.3.5 Resources

The following papers and websites provide useful information on software reliability modeling:

- "Software Reliability Assurance Handbook", http://www.cs.colostate.edu/~cs630/rh/
- "Software Reliability Modeling Techniques and Tools", Michael R. Lyu and Allen P. Nikora, ISSRE'93 Tutorial, November, 1993 http://techreports.jpl.nasa.gov/1993/93-1886.pdf
- "Software Reliability: To Use or Not To Use?", a panel discussion chaired by Michael Lyu, http://www.stsc.hill.af.mil/crossTalk/1995/feb/Reliable.asp
- "Applying Software Reliability Engineering in the 1990s", W. Everett, S. Keene, and A. Nikora, *IEEE Transaction on Reliability*, Vol. 47, No. 3-SP, September 1998

- "Software Reliability: Assumptions, Realities and Data", Michel Defamie, Patrick Jacobs, and Jacques Thollembeck, *Proceedings of the IEEE International Conference on Software Maintenance*, 1998
- "Software Reliability Engineering Study of a Large-Scale Telecommunications Software System", Carman et. al., *Proc. 1995 International Symposium on Software Reliability Engineering*, Toulouse, France, Oct. 1995, pp. 350-.
- "Predicting Software Reliability", Alan Wood, *IEEE Computer*, Vol. 29, No. 11, November 1996
- "Software Metrics and Reliability", Dr. Linda Rosenberg, Ted Hammer, and Jack Shaw, http://satc.gsfc.nasa.gov/support/ISSRE_NOV98/software_metrics_and_reliability.html
- "Reliability Modeling for Safety-Critical Software", Norman F. Schneidewind, *IEEE Transactions on Reliability*, Vol. 46, No.1, March 1997, pp. 88-98
- "Handbook of Software Reliability Engineering" (Book), Edited by Michael R. Lyu, Published by IEEE Computer Society Press and McGraw-Hill Book Company, http://www.cse.cuhk.edu.hk/~lyu/book/reliability/

### 5.5.4 Checklists of Tests

The software development group (or the safety engineer) should create a list of all tests that will be done on the software. *Section 4.6 Software Integration and Test* discusses the different variety of tests that can be conducted. The test checklist should be maintained by both the development and safety personnel. That provides a cross-check, to make sure no tests are accidentally missed.

Benefit-to-Cost Rating: **HIGH**

### 5.5.5 Test Results Analysis

Once tests are conducted (and witnessed), a test report is written that describes what was done and how the results match (or differ from) the expected results. The safety engineer uses these test reports, and problem/resolution reports, to verify that all safety requirements have been satisfied. The test results 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.

Benefit-to-Cost Rating: **HIGH**

### 5.5.6 Independent Verification and Validation

For high value systems with high risk software, an IV&V organization is usually involved to oversee the software development. Verification & Validation (V&V) is a system engineering process employing a variety of software engineering methods, techniques, and tools for evaluating the correctness and quality of a software product throughout its life cycle. IV&V is performed by an organization that is technically, managerially, and financially independent of the development organization.

IV&V should supplement, not supercede, the in-house software quality/product assurance efforts. Software QA and safety should still be involved with the project from the start, reviewing documents, offering advice and suggestions, and monitoring the software development process. Depending on what is negotiated with the project, the IV&V personnel may be a second set of eyes, shadowing the software QA, conducting independent audits, witnessing testing, etc. This requires the IV&V person to be stationed with the project, or to visit frequently. A more remote form of IV&V involves reviewing the software products (plans, designs, code, test results, code review reports, etc.), with a few in-person audits to verify the software development process.

The IV&V organization should fully participate in the validation of test analyses and traceability back to the requirements.

Benefit-to-Cost Rating: **MEDIUM**

### 5.5.7 Resources

http://www.chillarege.com/authwork/TestingBestPractice.pdf provides information on the testing, and test development, process. Software Testing Hotlist, Resources for Professional Software Testers, http://www.io.com/~wazmo/qa/, is a very good reference site for testing information. The Software QA and Testing Resource Center, http://www.softwareqatest.com/, also provides useful information on testing and the QA process.

## 5.6 Operations & Maintenance

Maintenance of software is describe in *Section 4.8 Software Operations & Maintenance*.

During the operational phase of a safety critical software set, rigorous configuration control must be enforced. For every proposed software change, it is necessary to repeat each development and analysis task performed during the life cycle steps previously used for each modification, from requirements (re-)development through code (re-)test. The safety analyst must ensure that proposed changes do not disrupt or compromise pre-established hazard controls.

It is advisable to perform the final verification testing on an identical off-line analog (or simulator) of the operational software system, prior to placing it into service.

# 6. SOFTWARE DEVELOPMENT ISSUES

In this chapter, we'll look at various programming languages, operating systems, tools, and development environments being used in safety critical software. Also included are various new technologies that have particular (and usually unsolved) problems with determining their safety. Finally, a sampling good programming practices specific to safety issues is presented.

Choosing a programming language is a necessity for any project. This sections examines a subset of the available languages, as there are over a hundred languages. The languages considered are those that are commonly used in safety critical or embedded environments. Also considered are languages that might be considered, because they are popular or new. For each language, any safety-related strengths are discussed, and guidance is given on what aspects to avoid.

Safer software can be written in any language. Coding standards can designate how to program in a particular language to produce safer code. But we're human. We make mistakes, we get in a hurry, and the coding standards may not be followed. Languages that are "safer" are those that enforce the standards, that check for common errors, and that do so as early as possible!

This chapter will also look at the environment the software will be developed and run in. Issues with compilers, tools, Integrated Development Environments (IDEs), automatic code generation, and operating systems (especially Real-Time (RTOS)) will be considered.

**Why does software have bugs?** [1]

- **miscommunication or no communication** - as to specifics of what an application should or shouldn't do (the application's requirements).
- **software complexity** - the complexity of current software applications can be difficult to comprehend for anyone without experience in modern-day software development. Windows-type interfaces, client-server and distributed applications, data communications, enormous relational databases, and sheer size of applications have all contributed to the exponential growth in software/system complexity. And the use of object-oriented techniques can complicate instead of simplify a project unless it is well-engineered.
- **programming errors** - programmers, like anyone else, can make mistakes.
- **changing requirements** - the customer may not understand the effects of changes, or may understand and request them anyway - redesign, rescheduling of engineers, effects on other projects, work already completed that may have to be redone or thrown out, hardware requirements that may be affected, etc. If there are many minor changes or any major changes, known and unknown dependencies among parts of the project are likely to interact and cause problems, and the complexity of keeping track of changes may result in errors. Enthusiasm of engineering staff may be affected. In some fast-changing business environments, continuously modified requirements may be a fact of life. In this case, management must understand the resulting risks, and QA and test engineers must

adapt and plan for continuous extensive testing to keep the inevitable bugs from running out of control - see 'What can be done if requirements are changing continuously?' in Part 2 of the FAQ.

- **time pressures** - scheduling of software projects is difficult at best, often requiring a lot of guesswork. When deadlines loom and the crunch comes, mistakes will be made.
- **egos** - people prefer to say things like:

  'no problem'
  'piece of cake'
  'I can whip that out in a few hours'
  'it should be easy to update that old code'

  instead of:

  'that adds a lot of complexity and we could end up making a lot of mistakes'
  'we have no idea if we can do that; we'll wing it'
  'I can't estimate how long it will take, until I take a close look at it'
  'we can't figure out what that old spaghetti code did in the first place'

  If there are too many unrealistic 'no problem's', the result is bugs.
- **poorly documented code** - it's tough to maintain and modify code that is badly written or poorly documented; the result is bugs. In many organizations management provides no incentive for programmers to document their code or write clear, understandable code. In fact, it's usually the opposite: they get points mostly for quickly turning out code, and there's job security if nobody else can understand it ('if it was hard to write, it should be hard to read').
- **software development tools** - visual tools, class libraries, compilers, scripting tools, etc. often introduce their own bugs or are poorly documented, resulting in added bugs.

[1]The list above was taken with permission from the Software QA and Testing Frequently Asked Questions. http://www.softwareqatest.com © 1996-2000 by Rick Hower

## 6.1 Safe Subsets of Languages

A safe subset of a language is one that restricts certain features that are error-prone or are undefined or poorly defined. In some cases, a subset may be created by a particular vendor, or may grow out of the user community. In many cases, a standard subset does not exist, but "coding standards" are used to create the subset. Using coding standards means that the compiler will not enforce the subset, however.

There are two primary reasons for restricting a language definition to a subset:

1) some features are defined in an ambiguous manner

2) some features are excessively complex or error-prone.

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.

Compilers for "safer" language subsets are often certified to provide correct translation from the source code to object code. The subset usually undergoes vigorous study and verification before it is accepted by the user community.

Besides formal language subsets, safety specific coding standards are used to specify requirements for annotation of safety-critical code and prohibit use of certain language features which can reduce software safety. Avoid including programming style requirements in a coding standard. Put those in a separate coding style document. While you want programmers to use the same style, it is far more important that they following the safety-related coding standards. A "style war" can lead to programmers ignoring the whole document, if style and standards are mixed.

## 6.2 Insecurities Common to All Languages

All programming languages have insecurities either in their definition or their implementation. Newer languages (or updates to existing language standards) try to correct the shortfalls of older generation languages, while adding additional functionality. In reality, they often add new insecurities as well.

Some common problems are:

- **Use of uninitialized variables.** Uninitialized variables are the most common error in practically all programming languages. In particular, uninitialized or improperly initialized pointers (in languages that support them) often cause insidious errors. 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.
- **Memory management concerns**. Calls to deallocate memory should be examined to make sure that not only is the pointer released but that the memory used by the structure is released. Also, it is important to verify that only one deallocation call is made for a particular memory block. On the other side of the problem, memory that is not deallocated when no longer used will lead to a memory leak, and perhaps to an eventual system crash.
- **Unspecified compiler behavior.** The order operands are evaluated in is often not defined by the language standard, and is left up to the compiler vendor. Depending on the order of evaluation for certain "side effects" to be carried out is poor programming practice! The order of evaluation may be understood for this compiler and this version only. If the program is compiled with a different vendor, or a different version, the side effects may well change. Other unspecified behavior may include the order of initialization of global or static variables.

## 6.3 Method of Assessment

When comparing programming languages, we will not deal with differences among vendor implementations. Compiler implementations, by and large, do not differ significantly from the intent of the standard. However, standards are not unambiguous and they are interpreted by the vendor. Be aware that implementations will not adhere 100% to the standard because of the extremely large number of states a compiler can produce. We will present information on the strengths and weaknesses of popular programming languages, and discuss safety related concerns. Common errors specific to the language will be discussed as well.

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?

*Strong typing implies an explicit data type conversion is required when transforming one type to another

## 6.4 Languages

There are over one hundred programming languages, with more being developed each year. Many are generated within academia as part of a research project. However, the subset of well established languages is more limited. The following languages will be examined in detail, focusing on the strengths and weaknesses each has with regards to producing safe software.

- Ada83, Ada95 and safe subsets
- Assembly Languages
- C

- C++
- C#
- Forth
- FORTRAN
- Java
- LabVIEW
- Pascal
- Visual Basic

### 6.4.1 Ada83 and Ada95 Languages

One of the most commonly used language in military and safety critical applications is Ada. From the inception of Ada83 until 1997, Ada was mandated by the Department of Defense for all weapons-related and mission critical programs. Though currently not mandated, Ada is still commonly used within military projects. In addition, safety critical commercial software is being written in Ada. Ada is also the primary language of the International Space Station. The Ada language was designed with safety and reliability in mind. The goal of Ada is to maximize the amount of error detection as early in the development process as possible.

The Ada standard was first released on 17th February 1983 as ANSI/MIL-STD-1815A “Reference Manual for the Ada Programming Language”. This original version is now called Ada83. The first major revision of the Ada standard was released on 21 December 1994 via ISO/IEC 8652:1995(E), and is commonly known as Ada95. Ada95 corrects many of the safety deficiencies of Ada83 and adds full object oriented capabilities to the language

The strengths of Ada95 lie in the following attributes:

- **Object orientation**

  Ada95 supports all the standard elements of object orientation: encapsulation of objects, inheritance, and polymorphism. Encapsulation “hides” information from program elements that do not need to know about it, and therefore decreases the chance of the information being altered unexpectedly. Inheritance and polymorphism contribute to the extensibility of the software.

  Software reuse is one “plus” of object orientation. A previously tested object can be extended, with new functionality, without “breaking” the original object.

- **Strongly typed**

  Ada enforces data typing. This means that you cannot use an integer when a floating point number is expected, unless you explicitly convert it. Nor can you access an integer array through a character pointer. Strong typing finds places where the programmer assumed one thing, but the source code actually lead to another implementation. Forcing conversions helps the programmer think about what she is doing, and why, rather than allowing the compiler to make implicit (and perhaps undefined) conversions.

- **Range checking**

Range checking for arrays, strings, and other dimensioned elements is included in the language. This prevents accidentally overwriting memory outside of the array. The compiler will usually find the error. If not, a Run Time Exception (RTE) will be generated. Also included is checking for references to null.

- **Support for multitasking and threads**

  Tasking is built into the language. Support is included to deal with threads and concurrency issues. Protected objects provide a low overhead, data-oriented synchronization mechanism. Asynchronous transfer of control, with "clean up" of the interrupted process, is part of the language.

- **Clarity of source code**

  Ada code is closer to "regular" language than most languages, and this makes it easy to read. Often, coming back to code you wrote awhile ago is difficult. Much of the context has been forgotten, and it may be difficult to understand why you did something. When the code is easy to read, those problems are reduced. Also, when the code is reviewed or inspected, others find it easier to understand.

- **Mixed language support**

  Ada allows modules written in other languages to be used. Usually, just a "wrapper" must be created before the non-Ada routines can be accessed. This allows well-tested, legacy code to be used with newer Ada code.

- **Real-time system support**

  Ada95 has added support for real-time systems. Hard deadlines can be defined. Protected types give a "low overhead" type of semaphore. Dynamic task priorities is the mechanism to set the priority of a task at run-time, rather than compile-time, and is supported. Priority inversion, used to prevent deadlock when a high priority task needs a resource being used by a lower priority task, can be bounded. This allows Rate Monotonic Analysis to be used.

- **Distributed systems support**

  A "unit" in an Ada95 distributed system is called a partition. A partition is an aggregation of modules that executes in a distributed target environment. Typically, each partition corresponds to a single computer (execution site). Communication among partitions of a distributed system is based upon extending the remote procedure call paradigm.

- **Exception handling**

  Exceptions are raised by built-in and standard library functions, when events such as an integer overflow or out-of-bounds check occurs. Exceptions can also be raised by the program specifically, to indicate that the software reached an undesirable state. Exceptions are handled outside of the normal program flow, and are usually used to put the software into a known, safe state. The exception handler, written by the programmer, determines how the software deals with the exception.

- **Support for non-object-oriented (traditional) software development styles**

  Though Ada95 supports object-oriented programming, the language can be used with other styles as well. Functional (structural) programming techniques can be used with the language.

Additional safety-related features are

- **Compiler validation**

  All Ada compilers must be validated. This means the compiler is put through a standard set of tests before it is declared a true Ada compiler. This does not mean that the compiler does not have defects, however. When choosing a compiler, ask for the history list of defects.

- **Language restriction ability**

  Ada95 added a restriction pragma. This allows features of the language to be "turned off". You can specify a subset of the language, removing features that are not needed or that may be deemed unsafe. If a feature is not included, it does not have to be validated, thus reducing the testing and analysis effort.

- **Validity checking of scalar values**

  Ada95 has added a Valid attribute which allows the user to check whether the bit-pattern for a scalar object is valid with respect to the object's nominal subtype. It can be used to check the *contents* of a scalar object *without formally reading* its value. Using this attribute on an uninitialized object is not an error of any sort, and is guaranteed to either return True or False (and not raise an exception). The results are based on the actual contents of the object and not on what the optimizer might have assumed about the contents of the object based on some declaration.

  Valid can also be used to check data from an unchecked conversion, a value read from I/O, an object for which pragma Import has been specified, and an object that has been assigned a value where checks have been suppressed.

- **Reviewable object code**

  Ada provides mechanisms to aid in reviewing the object code produced by the compiler. Because the compiler will have defects, it is important in safety critical applications to review the object code itself.

  The pragma Reviewable can be applied to a partition (program) so that the compiler can provide the necessary information. The compiler vender should produce information on the ordering of modules in the object code, what registers are used for an object (and how long is the assignment valid), and what machine code instructions are used. In addition, a way to extract the object code for a particular module, so that other tools can use it, is suggested. Other information should support Initialization Analysis (what is the initialization state of all variables), determining the relationship between Source and Object Code, and Exception Analysis (indicating where compiler-generated run-time checks occur in the object code and which exceptions can be raised by any statement) may also be supported by the compiler vendor.

An Inspection Point pragma provides a "hook" into the program similar to a hardware test point. At the inspection point(s) in the object code, the values of the specified objects, or all live objects, can be determined.

However, no language is perfect. Ada95 does **not** detect the use of uninitialized variables, though using the "Normalize_Scalars" pragma will help. Some aspects of the language to consider restricting, for a safety critical application, are:

- The ability to turn off the type checking and other safety features
- Garbage collection…turn it off if timing issues are important.

"Ada Programming Guidelines", by Rational Software Corporation, are available at http://www.cs.hmc.edu/tech_docs/qref/rational/DevelopmentStudioUNIX.1.1/docs/html/rup_ada/ada.htm

### 6.4.2 Assembly Languages

Assembly languages are the human-readable version of machine code. They are specific to a processor or processor family. If an operating system is used, the method to access the functions of the OS are specific to that system. Programming in assembly requires intimate knowledge of the workings of the processor.

Modern assembly languages include macros that allow "higher level" logic flow, such as if…else statements and looping. Variables can be declared and named. Subroutines (procedures) can also be declared and called. All "higher level" constructs improve the readability and maintainability of the assembly program.

Few large programs are written entirely in assembly language. Often, a small section of the software will be rewritten in assembly to increase execution speed or decrease code size. Also, the code used on bootup of a system (that loads the operating system) and BIOS-like utilities are often written in assembly. Interrupt service routines are another place you will find assembly language used. In addition, software that runs on small microcontrollers are often space-limited and therefore assembly coding is a good alternative to a high-level language.

**Why use assembly:**

- Execution Speed
- Smaller code size
- Ability to do something that higher level languages do not allow.
- Tweaking the compiler's optimization, by editing the assembly output it produces

**Problems and safety concerns**:

™ Can do anything with the processor and access any part of memory

™ No notion of data type – a sequence of bytes can be anything you want!

™ You can jump anywhere in address space

™ All higher level constructs (structures, arrays, etc.) exist only in the programmer's implementation, and not in the language.

™ Not portable between processors

Compilers can usually produce assembly source code from the higher level language. This is useful for checking what the compiler does, and verifying its translation to that level. In fact, if the compiler produces correct assembly source code but incorrect object code, creating the assembly source and then using a different assembler to generate the object code could bypass the problem.

More often, the assembly output is used to "tweak" performance in a slow routine. Use a profiling program to find out *where* the slow sections are first. The part of the program that the programmer thinks is likely to be slow is often not the actual problem. Running the program with a profiler will give hard numbers, and point to the truly sluggish sections of code.

### 6.4.3 C Language

The C language is extremely popular because of its flexibility and support environment. C is often used in embedded and real-time environments, because hardware access is fairly easy and small, compact code can be generated. In many ways, C is a higher level assembly language. This gives it great flexibility, and opens a Pandora's box of possible errors. The support environment includes a wide variety of mature and inexpensive development and verification tools. Also, the pool of experienced vendors and personnel is quite large.

However, C's definition lacks the rigor necessary to qualify it as a suitable vehicle for safety-critical applications. There are dozens of dialects of C, raising integrity concerns about code developed on one platform and used on another. Despite its problems, many safety-critical applications have been coded in C and function without serious flaws. If C is chosen, however, the burden is on the developer to provide a thorough verification of code and data sequences, and sufficient testing to verify both functionality and error handling.

One characteristic of C that decreases its reliability is that C is not a strongly typed language. That means that the language doesn't enforce the data typing, and it can be circumvented by representational viewing. (This means that by unintended use of certain language constructs, not by explicit conversion, a datum that represents an integer can be interpreted as a character.) The definition of strong typing implies an explicit conversion process when transforming one data type to another. C allows for implicit conversion of basic types and pointers. One of the features of strong typing is that sub-ranges can be specified for the data. With a judicious choice of data types, a result from an operation can be shown to be within a sub-range. In C it is difficult to show that any integer calculation cannot overflow. Unsigned integer arithmetic is modulo the word length without overflow detection and therefore insecure for safety purposes.

Another feature of C that does not restrict operations is the way C operates with pointers. C does not place any restrictions on what addresses a programmer can point to and it allows arithmetic on pointers. While C's flexibility makes it attractive it also makes it a less reliable programming language. C has other limitations which are mentioned in reference [15].

Restricting the C language to certain constructs would not be feasible because the resulting language would not have the necessary functionality. However, rigorous enforcement of coding standards will decrease certain common errors and provide some assurance that the software will function as expected. Structured design techniques should be used.

**Limitations and Problems with the C language:**

- **Pointers**

  Pointers in C allow the programmer to access anything in memory, if the operating system does not prevent it. This is good when writing a device driver or accessing memory-mapped I/O. However, a large number of C errors are due to pointer problems. Using a pointer to access an array, and then running past the end of the array, leads to "smash the stack" (see below). Pointer arithmetic can be tricky, and you can easily point outside of the data structure. Use of undefined pointers can trash memory or the stack, and lead the program to wander into undefined territory.

- **Lack of Bounds Checking**

  C does not provide any bounds checking on arrays and strings. It is left to the programmer to make sure that the array element is truly in bounds. Since the programmer is fallible, "smash the stack" and "fandango on core" often result. The problem is especially evident when passing an array to a function, which references it via a pointer. The function must know the length of the array, or it may run past the end. Calculations that determine the element to access must also be checked, as a negative or too large value can result, leading to "out of bounds" accesses.

  A "wrapper" function can be used when accessing an array or string which checks that the element is within bounds. This adds runtime overhead, but decreases the number of errors.

- **Floating Point Arithmetic**

  The ANSI C standard does not mandate any particular implementation for floating point arithmetic. As a result every C compiler implements it differently. The following test calculation can be executed:

  $x = 10^{20}+1$

  $y = x-10^{20}$

  The resulting value for y will differ greatly from compiler to compiler, none of them will be correct due to word length round-off.

- **Casting from void***

  void* points to data of any type. It is left to the programmer to recast it when the pointer is used. There is no compile time nor run time checking to verify that the pointer is cast to a valid type (based on what the pointer actually points to). This method is inherently tricky and prone to errors.

- **Commenting problems**

  The C comment /* … */ can lead to unexpected problems, by accidentally commenting out working code. Forgetting the end comment marker (*/) can cause the code that follows to be comment out, until another comment end marker is found. A good editor will often show this problem while the code is being developed, if it marks commented text in a different color. Also, compilers should be set up to generate warnings or errors to be generated if an open comment (/*) is found within a comment.

- **Global variables**

  Global variables can be considered as input parameters to any function, since a function has full access to them. So a function that takes 2 parameters, in a program with 100 global variables, actually has 102 parameters. This makes verifying the program very difficult. It is best to avoid global variables as much possible. Global variables also cause problems in a multi-threaded program, e.g. when different threads believe they have control of the variable, and both change the global.

- **Common language errors**
  - Confusing **=** with **==** (assignment with logical equality)
  - Confusing **&** vs. **&&** (Bitwise AND with logical AND)
  - premature semicolon in control structures
  - fall-through behavior in switch statements when "break" is omitted
  - Comparing signed and unsigned variables. Particularly, testing "unsigned < 0" or "unsigned < negative signed value".

- **Side effects and macros**

  Side effects, such as incrementing a variable with ++, when mixed with macros (including "functions" that are actually implemented as a macro, such as putchar()), may produce unexpected results.

- "**smash the stack**"*

  <jargon> In C programming, to corrupt the execution stack by writing past the end of a local array or other data structure. Code that smashes the stack can cause a return from the routine to jump to a random address, resulting in insidious data-dependent bugs.

  Variants include "trash the stack", "scribble the stack", "mangle the stack".

- **"precedence lossage"***

  /pre's*-dens los'*j/ A C coding error in an expression due to unintended grouping of arithmetic or logical operators. Used especially of certain common coding errors in C due to the nonintuitively low precedence levels of "&", "|", "^", "<<" and ">>". For example, the following C expression, intended to test the least significant bit of x,

  ```
  x & 1 == 0
  ```
  is parsed as
  ```
  x & (1 == 0)
  ```
  which the compiler would probably evaluate at compile-time to
  ```
  (x & 0)
  ```
  and then to 0.

  Precedence lossage can always be avoided by suitable use of parentheses. For this reason, some C programmers deliberately ignore the language's precedence hierarchy and use parentheses defensively.

- **"fandango on core"***

(Unix/C, from the Mexican dance) In C, a wild pointer that runs out of bounds, causing a core dump, or corrupts the malloc arena in such a way as to cause mysterious failures later on, is sometimes said to have "done a fandango on core". On low-end personal machines without an MMU, this can corrupt the operating system itself, causing massive lossage. Other frenetic dances such as the rhumba, cha-cha, or watusi, may be substituted.

- **"overrun screw"***

  A variety of fandango on core produced by a C program scribbling past the end of an array (C implementations typically have no checks for this error). This is relatively benign and easy to spot if the array is static; if it is auto, the result may be to smash the stack - often resulting in heisenbugs of the most diabolical subtlety. The term "overrun screw" is used especially of scribbles beyond the end of arrays allocated with malloc; this typically overwrites the allocation header for the next block in the arena, producing massive lossage within malloc and often a core dump on the next operation to use stdio or malloc itself.

- **"C Programmer's Disease"***

  The tendency of the undisciplined C programmer to set arbitrary but supposedly generous static limits on table sizes (defined, if you're lucky, by constants in header files) rather than taking the trouble to do proper dynamic storage allocation. If an application user later needs to put 68 elements into a table of size 50, the afflicted programmer reasons that he or she can easily reset the table size to 68 (or even as much as 70, to allow for future expansion) and recompile. This gives the programmer the comfortable feeling of having made the effort to satisfy the user's (unreasonable) demands, and often affords the user multiple opportunities to explore the marvelous consequences of fandango on core. In severe cases of the disease, the programmer cannot comprehend why each fix of this kind seems only to further disgruntle the user.

*These quotations were taken from Imperial College, London, UK, world wide web home page Dictionary of Computer Terminology (http://wombat.doc.ic.ac.uk/), compiled by Denis Howe. It contains graphic descriptions of common problems with C. The quotations were reproduced by permission of Denis Howe <dbh@doc.ic.ac.uk>.

Other references [24] discussed the important problem of dynamic memory management in C (Note that simply prohibiting dynamic memory management is not necessarily the best course, due to increased risk of exceeding memory limits without warning).

Programming standards for C should include at least the following:

™ Use parentheses for precedence of operation, and do not rely on the default precedence. The default may not be what you thought it was, and it will come back to bite you.

™ Use parentheses within macros, around the variable name

™ Don't use the preprocessor for defining complex macros

™ Explicitly cast or convert variables. Do not rely on the implicit conversions.

™ Avoid void* pointers when possible.

™ Check arrays and strings for "out of bounds" accesses.

™ Always use function prototypes. This allows the compiler to find problems with inconsistent types when passing variables to a function.

™ Minimize the use of global variables. Each global can be considered a parameter to every function, increasing the chance of accidentally changing the global.

™ Always include a *default* clause in a *switch…case* statement.

™ Avoid recursion when possible.

™ Make extensive use of error handling procedures and status and error logging.

**The Ten Commandments for C Programmers**
by Henry Spencer

1. Thou shalt run lint frequently and study its pronouncements with care, for verily its perception and judgment oft exceed thine.
2. Thou shalt not follow the NULL pointer, for chaos and madness await thee at its end.
3. Thou shalt cast all function arguments to the expected type if they are not of that type already, even when thou art convinced that this is unnecessary, lest they take cruel vengeance upon thee when thou least expect it.
4. If thy header files fail to declare the return types of thy library functions, thou shalt declare them thyself with the most meticulous care, lest grievous harm befall thy program.
5. Thou shalt check the array bounds of all strings (indeed, all arrays), for surely where thou typest "foo" someone someday shall type "supercalifragilisticexpialidocious".
6. If a function be advertised to return an error code in the event of difficulties, thou shalt check for that code, yea, even though the checks triple the size of thy code and produce aches in thy typing fingers, for if thou thinkest "it cannot happen to me", the gods shall surely punish thee for thy arrogance.
7. Thou shalt study thy libraries and strive not to re-invent them without cause, that thy code may be short and readable and thy days pleasant and productive.
8. Thou shalt make thy program's purpose and structure clear to thy fellow man by using the One True Brace Style, even if thou likest it not, for thy creativity is better used in solving problems than in creating beautiful new impediments to understanding.
9. Thy external identifiers shall be unique in the first six characters, though this harsh discipline be irksome and the years of its necessity stretch before thee seemingly without end, lest thou tear thy hair out and go mad on that fateful day when thou desirest to make thy program run on an old system.
10. Thou shalt foreswear, renounce, and abjure the vile heresy which claimeth that "All the world's a VAX", and have no commerce with the benighted heathens who cling to this barbarous belief, that the days of thy program may be long even though the days of thy current machine be short.

A checklist of Generic and C-specific programming standards is included in Appendix B. Additional guidelines on C programming practices are described in the book “Safer C: Developing Software for High-integrity and Safety-critical Systems” (Reference [25], and also in [27] and [28]). Included in the book are lists of undefined or implementation defined behaviors in the language.

### 6.4.4 C++ Language

The C++ programming language was created by Bjarne Stroustrup as an extension (superset) of the C programming language discussed above (*Section 6.4.3 C Language*). The goal was to add object-oriented features, while maintaining the efficiency of C. The language was standardized in

November, 1997 as ISO/IEC 14882. C++ adds Object Orientation (OO) as well as fixing or updating many C features. C++ is also more strongly typed than C. However, C++ suffers from many of the same drawbacks as C.

A standard "safe subset" of C++ does not presently exist.

**Strengths of the C++ Language**

- **Object Orientation**

  Object orientation allows data abstraction (classes), encapsulation of data and the functions that use the data, and reusable/extensible code.

- **Stronger type checking than C**

  C++ type checking can be subverted, but it is much better than C's. Most of the mechanisms that reduce the type checking were left in to support compatibility with the C language.

- ***Const* to enforce the "invariability" of variables and functions**

  Declaring a function *const* means that the function will not change any passed parameters, even if they are passed by reference. A *const* variable cannot be changed, and replaces the "#define" preprocessor directives. The programmer can get around *const* with a cast.

- **Generic programming (templates).** C++ has the ability to use generic containers (such as vectors) without runtime overhead.

- **C++ supports both Object-Oriented and Structural design and programming styles.**

- **The user-defined types (classes) have efficiencies that approach those of built-in types. C++ treats of built-in and user-defined types uniformly**

- **Exceptions and error handling**

  Exceptions allow errors to be "caught" and handled, without crashing the program. They may not be the best way to handle errors, and the software does have to be explicitly designed to generate and deal with exceptions. However, exceptions are an improvement over C's *setjmp()* and *longjmp()* means of exception handling.

- **Namespaces**

  Namespaces are most useful for libraries of functions. They prevent function names from conflicting, if they are in different libraries (namespaces). While not primarily a safety feature, namespaces can be used to clearly identify to the reader and the programmers what functions are safety related.

- **References to variables**

  A reference is like a pointer (it points to the variable), but it also simplifies the code and forces the *compiler* to create the pointer, not the programmer. Anything the compiler does is more likely to be error free than what the programmer would do.

- **Inline Functions**

Inline functions replace #define macros. They are easier to understand, and less likely to hide defects.

**Good practices to reduce C++ errors:**

™ Never use multiple inheritance, only use one to one (single) inheritance. This is because interpretations of how to implement multiple inheritance are inconsistent (Willis and Paddon, [29] 1995. Szyperski supports this view.);

™ Minimize the levels of inheritance, to reduce complexity in the program.

™ Only rely on fully abstract classes, passing interface but not implementation (suggestion by Szyperski at 1995 Safety through Quality Conference - NASA-KSC [30]).

™ Minimize the use of pointers.

™ Do not allow aliases.

™ No side-effects in expressions or function calls.

™ Make the class destructor virtual if the class can be inherited.

™ Always define a default constructor, rather than relying on the compiler to do it for you.

™ Define a copy constructor. Even if a "bitwise" copy would be acceptable (the default, if the compiler generates it), that may change in the future. If any memory is allocated by the class methods, then a copy constructor is vital. If the class objects should not be copied, make the copy constructor and assignment operator private, and do not define bodies for them.

™ Define an assignment operator for the class, or add a comment indicating that the compiler-generated one will be used.

™ Use operator overloading sparingly and in a uniform manner. This creates more readable code, which increases the chance of errors being found in inspections, and reduces errors when the code is revisited.

™ Use const when possible, especially when a function will not change anything external to the function. If the compiler enforces this, errors will be found at compile time. If not, it will aid in finding errors during formal inspections of the code.

™ Don't use the RTTI (Run-Time Type Information). It was added to support object oriented data bases. If you think it's necessary in your program, look again at your design.

™ Avoid global variables. Declare them in a structure as static data members.

™ Make sure that the destructor removes all memory allocated by the constructor and any member functions, to avoid memory leaks.

™ Use templates with care, including the Standard Template Library. The STL is not thread-safe.

™ Take special care when using delete for an array. Check that delete[] is used. Also check for deleting (freeing) a pointer that has been changed since it was allocated. For example, the following code will cause problems:

```
p = new int[10];          // allocate an array of 10 integers
p++;                      // change the pointer to point at the second integer
delete p;          // error, not array delete (delete[]) and  pointer changed
```

A review of potential problems in C++ was published by Perara (Reference [26]). The headings from that paper are as follows:

- ✓ Don't rely on the order of initialization of globals
- ✓ Avoid variable-length argument lists
- ✓ Don't return non-constant references to private data
- ✓ Remember 'The Big Three'
- ✓ Make destructors virtual
- ✓ Remember to de-allocate arrays correctly
- ✓ Avoid type-switching
- ✓ Be careful with constructor initialization lists
- ✓ Stick to consistent overloading semantics
- ✓ Be aware of the lifetimes of temporaries
- ✓ Look out for implicit construction
- ✓ Avoid old-style casts
- ✓ Don't throw caution to the wind exiting a process
- ✓ Don't violate the 'Substitution Principle'
- ✓ Remember there are exceptions to every rule.

A detailed discussion is provided on each point, in that reference.

### 6.4.5 C# Language

The C# language is "cutting edge". It has been created by Microsoft and is expected to be released in the second half of 2001. C# is loosely based on C/C++, and bears a striking similarity to Java in many ways. Microsoft describes C# as follows:

> "C# is a simple, modern, object oriented, and type-safe programming language derived from C and C++. C# (pronounced 'C sharp') is firmly planted in the C and C++ family tree of languages, and will immediately be familiar to C and C++ programmers. C# aims to combine the high productivity of Visual Basic and the raw power of C++."

C# has been created as part of Microsoft's .NET environment, and is primarily designed for it. Any execution environment will have to support aspects that are specific to the Microsoft platform. Since they must support the Win32 API, C# may be restricted to Win32 machines. However, at least one company is considering porting C# to Linux.

C# has the following features:

- Exceptions

- References can be null (not referencing real object). C# throws an exception if the reference is accessed.
- Garbage collection. You CAN'T delete memory, once it is allocated!
- Array bounds checking (throws an exception)
- Like Java, machine-independent code which runs in a managed execution environment (like the JVM)
- No pointers, except in routines marked **unsafe**
- Multi-dimensioned arrays
- Switch statements do not allow "fall through" to next case.
- Thread support, including locking of shared resources
- No global variables
- All dynamically allocated variables initialized before use.
- The compiler produces warnings if using uninitialized local variable.
- Overflow checking of arithmetic operations (which can be turned off if needed)
- "foreach" loop – simpler way to do a *for* loop on an array or string. This decreases the chance of going out of bounds, because the compiler determines how often to loop, not the programmer.
- Everything is "derived" from the base class (system class). This means that integers, for example, have access to all the methods of the base class. The following code in C# would convert an integer to a string and write it on the console:

```
int i = 5;
System.Console.WriteLine (i.ToString());
```

- Has a goto statement but it may only point anywhere within its scope, which restricts it to the same function or *finally* block, if it is declared within one. It may not jump into a loop statement which it is not within, and it cannot leave a *try* block before the enclosing finally block(s) are executed.
- Pointer arithmetic can be performed in C# within methods marked with the **unsafe** modifier.
- "Internet" oriented (like Java)

The following features are of concern in embedded or safety environments:

™ Garbage collection can lead to non-deterministic timing. It is a problem in real-time systems.

™ Portability: C# is currently only designed to work on Microsoft Windows systems.

™ Speed: C# is an interpreted language, like Java. Unlike Java, there is currently no compiler that will produce native code.

### 6.4.6 Forth Language

The Forth language was developed in the 1960's by Charles Moore. He wanted a language that would make his work of controlling telescopes both easier and more productive.

Forth is "stack based" – the language is based on **numbers** and **words**. Numbers are pushed on the stack. Words are "executed" on the stack numbers, and are essentially functions. Words are kept in a Forth dictionary. The programmer can create new words (new functions), usually using existing words.

Forth uses reverse polish notation. The last number pushed on the stack is the first off it. A simple Forth statement to add 3 and 4 is: **3 4 +** (pushes 3, then pushes 4, + pops the top two numbers off the stack and adds them, then pushes the result onto the stack.). In this case, **+** is a built-in word (function).

Forth has the benefits of "higher level" language (like C), but it is also very efficient (memory and speed-wise). It is used mainly in embedded systems. Forth can be used as the Operating System (OS) as well, and it often is in small embedded microcontrollers.

Forth has no "safety" features. The programmer can do just about anything! It is very similar to C and assembly language this way. The flexibility and speed it gives must be balanced with the need to rigorously enforce coding standards, and to inspect the safety-critical code.

The Forth programmer must know where each parameter/variable is on the stack, and what type it is. This can lead to errors, if the type or location is incorrectly understood.

One positive aspect of Forth, from a safety standpoint, is that it is very easy to unit test. Each "word" is a unit and can be thoroughly tested, prior to integration into larger "words". There is work on applying formal methods to Forth

The following quote is from Philip J. Koopman Jr.[2] The italics are added to emphasize particular aspects of concern to programming a safe system.

"Good Forth programmers strive to write programs containing very short (often one-line), well-named word definitions and reused factored code segments. The ability to pick just the right name for a word is a prized talent. Factoring is so important that it is common for a Forth program to have more subroutine calls than stack operations. Factoring also simplifies speed optimization via replacing commonly used factors with assembly language definitions….

Forth programmers traditionally value complete understanding and control over the machine and their programming environment. Therefore, what *Forth compilers don't* do reveals something about the language and its use. *Type checking*, macro preprocessing, common subexpression elimination, and other traditional compiler services are feasible, but usually not included in Forth compilers. ….

Forth supports extremely flexible and productive application development while making ultimate control of both the language and hardware easily attainable."

[2]Philip J. Koopman Jr. by permission of the Association for Computing Machinery; **A Brief Introduction to Forth**; This description is copyright 1993 by ACM, and was developed for the Second History of Programming Languages Conference (HOPL-II), Boston MA. koopman@cmu.edu

### 6.4.7 FORTRAN Language

FORTRAN was developed in the 1950's by IBM, and first standardized in the 1960's, as FORTRAN 66. It is primarily a **numerical processing** language, great for number crunching. FORTRAN has gone through several standards since the 1960's. The versions of the language considered here are FORTRAN 77 and Fortran 90.

While not usually used in embedded systems, FORTRAN can still be used in a safety critical system (or a part of the system), if the numerical results are used in safety decisions.

FORTRAN 77 is a structured, procedural language. It contains all the elements of a high level language (looping, conditionals, arrays, subroutines and functions, globals, independent compilation of modules, and input/output (formatted, unformatted, and file)). In addition, it had complex numbers, which are not part of the other languages considered here. There is **no** dynamic memory (allocate/deallocate) in FORTRAN 77.

**Elements of FORTRAN 77 related to safety**

- **Weak data typing.** The data type of a variable can be assumed, depending on the first letter of the name, if it is not explicitly defined.
- **GOTO statements**. The programmer can jump anywhere in the program.
- **Fixed-form source input.** This relates to safety only in that it can look OK (in an inspection) and be wrong (incorrect column). However, the compiler should prevent this problem from occurring.
- **Limited size variable names.** The length of the variable name was limited to 8 characters. This prevented using realistic names that described the variable. Programmers often used cryptic names that made understanding and maintaining the program difficult.
- **Lack of dynamic memory.** This prevents the problems related to dynamic memory, though it limits the language for certain applications.
- **The EQUIVALENCE statement** should be avoided, except with the project manager's permission. This statement is responsible for many questionable practices in Fortran giving both reliability and readability problems.*
- **Use of the ENTRY statement.** This statement is responsible for unpredictable behavior in a number of compilers. For example, the relationship between dummy arguments specified in the SUBROUTINE or FUNCTION statement and in the ENTRY statements leads to a number of dangerous practices which often defeat even symbolic debuggers.*
- **Use of COMMON blocks.** COMMON is a dangerous statement. It is contrary to modern information hiding techniques and if used freely, can rapidly destroy the maintainability of a package.
- Array bounds checking is not done dynamically (at run time), though compilers may have a switch that allows it at compile time.

*These elements were extracted from Appendix A of Hatton, L. (1992) "Fortran, C or C++ for geophysical software development", Journal of Seismic Exploration, 1, p77-92.

Fortran 90 is an updated version of FORTRAN 77 that provides rudimentary support for Object Oriented programming, and other features. Fortran 90 includes:

- **Dynamic memory allocation**, specifically allocatable pointers and arrays.
- **Rudimentary support for OOP.** Inheritance is not supported. Constructors simply initialize the data members. There are no destructors. It does have derived types and operator overloading.
- **Rudimentary pointers.** A FORTRAN pointer is more of an alias (reference) than a C-style pointer. It cannot point to arbitrary locations in memory, or be used with an incorrect data type. Variables that will be pointed to must declare themselves as TARGETs.
- **Free-style format and longer variable names** (31 characters). These increase readability of the code.
- **Improved array operations**, including operating on a subsection of the array and array notation (e.g. X(1:N)). Statements like A=0 and C=A+B are now valid when A and B are arrays. Also, arrays are actually array objects which contain not only the data itself, but information about their size. There is also a built-in function for matrix multiplication (matmul).
- **Better function declarations (prototyping).**
- **Modern control structures** (SELECT CASE, EXIT, ...)
- **User defined data types (modules).** Like *struct* in C, or *record* in Pascal.
- **Recursive functions** are now a part of the language.

**Problems with Fortran 90:**

- Order of evaluation in **if** statements (if (a and b)) is undefined. A compiler can evaluate *b* first, or *a* first. However, "if (present(a) .and. a)" could cause a problem, if the compiler evaluates '*a*' (right side) first, and *'a'* doesn't exist. Do not rely on order of evaluation in if statements.
- Allocatable arrays opens the door to memory leakage (not deallocating when done) and accessing the array after it has been deallocated.
- Implicit variables are still part of the language. Some compilers support the extension of declaring IMPLICIT NONE, which forces the data type to be declared.

### 6.4.8 Java Language

Java was created by Sun Microsystems in 1995, with the first development kit (JDK 1.0) released in January, 1996. Since then, Java has become a widespread language, particularly in internet applications. Java is used in embedded web systems, as the front end/GUI for other embedded systems, and for data distribution/networking systems, among many other applications.

Java was created to be platform independent. Java programs are not normally compiled down to the machine code level. They compile to “byte code”, which can then be run on Java Virtual Machines (JVM). The JVM’s contain the machine-specific coding. When a Java program is run, the JVM interprets the byte code. This interpreted mode is usually slower than traditional program execution. In addition, timing will not be deterministic.

Work is in process to create Java specifications for real-time, embedded systems. In December, 1998, the Java Real-Time Expert Group was formed to create a specification for extensions to Java platforms that add capabilities to support real-time programming in Java and to support the adoption of new methodologies as they enter into practice. The group has focused on new APIs, language, and virtual machine semantics in six key areas (the Java thread model, synchronization, memory management, responding to external stimuli, providing accurate timing, and asynchronous transfer of control). JSR-000001, Real-time Specification for Java, was released in June, 2000.

Compilers for Java programs do exist. They compile the program down to the machine level. This decreases the portability and removes the platform independence, but allows an increase in execution speed and a decrease in program size. Compiled Java programs do not need a Java Virtual Machine (JVM).

Java has the following features:

- Fully Object Oriented. This has the plusses of reusability and encapsulation
- Dynamic loading of new classes, and object/thread creation at runtime.
- No pointers allowed! No pointer arithmetic and other pointer problems common in C. However, objects can be accessed through references.
- Garbage collection to free unused memory. The programmer doesn’t have to remember to delete the memory.
- Support for threads, including synchronization primitives.
- Support for distributed systems.
- No goto statement, though labeled break/continue statements are allowed.
- Allows implicit promotion (int to float, etc.), but conversion to lower type needs explicit cast
- Variables initialized to known values (including references)
- Allows recursion
- Checks array bounds and references to null
- Java's document comments (//*) and standard documentation conventions aid in readability.
- Type safe (compile variable and run-time types must match)
- No operator overloading

- Built-in GUI, with support for events
- Built-in security features (language limits uncontrolled system access, bytecode verification is implemented at run-time, distinguishes between trusted and untrusted (foreign) classes, and restricts changing of resources. Packages – downloaded code can be distinguished from local) However, still not secure. Ways to circumvent are found, and "bug fixes" are released
- Java automatically generates specifications (prototypes) (as opposed to using redundant specifications).

Java has these limitations:

™ can't interface to hardware; must use native methods of another language to do so.

™ Uses a Java Virtual Machine, which must be tested or certified, unless compiled to native code.

™ Garbage collection to free unused memory can't be turned off! This affects determinism in real-time systems.

™ Selfish threads (those that do not call sleep()), on some OS's, can hog the entire application. Threads can interfere with each other if using the same object. Synchronization makes the thread not be interrupted until done, but deadlock can still occur.

™ Doesn't detect "out of range" values (such as integer multiplication leading to an integer value that is too large).

™ When passing arguments to functions, all objects, including arrays, are call-by-reference. This means that the function can change them!

™ Java is an interpreted language, which is often slower than a compiled language. Compilers are available, however, which will get around this problem.

™ Non-deterministic timing. Real-time extensions are being worked on, but they are not standardized yet.

™ The Java language has not been standardized by a major standards group. It is in the control of Sun Microsystems.

### 6.4.6 LabVIEW

LabVIEW is a graphical programming language produced by National Instruments. It is used to control instrumentation, usually in a laboratory setting. LabVIEW allows the user to display values from hardware (temperatures, voltages, etc.), to control the hardware, and to do some processing on the data. It is primarily used in "ground stations" that support hardware (such as space flight instruments). LabVIEW may be part of safety critical software development if the "ground station" it supports is safety critical. In addition, it may be used to support infrastructures (e.g. wind tunnel) that have safety critical aspects.

In LabVIEW, the method by which code is constructed and saved is unique. There is no text based code as such, but a diagrammatic view of how the data flows through the program. LabVIEW is a tool of the scientist and engineer (who are not always proficient programmers) who can often visualize data flow, but are unsure of how to convert that into a conventional programming language. Also, LabVIEW's graphical structure allows programs to be built quickly.

Data flow is the fundamental tenet by which LabVIEW code is written. The basic philosophy is that the passage of data through nodes within the program determines the order of execution of the functions of the program. LabVIEW VI's (Virtual Instruments) have inputs, process data and produce outputs. By chaining together VI's that have common inputs and outputs it is possible to arrange the functions in the order by which the programmer wants the data to be manipulated.

LabVIEW source code and development is supported by Windows 9x/2000/NT, Macintosh, PowerMax OS, Solaris, HP-Unix, Sun, Linux, and Pharlap RTOS (Real-Time Operating System). Executables can be compiled under their respective development systems to run on these platforms (native code). Code developed under one platform can be ported to any of the others, recompiled and run.

LabVIEW has rich data structures (For and While loops, Shift registers, Sequencing, and Arrays and clusters). It supports polymorphism and compound arithmetic. Display types include Indicators, Graphs and charts, and Instrument simulation. Strings and file handling are included in LabVIEW. Many debugging techniques, such as breakpoints, single stepping, and probes, are supported.

A real-time version of LabVIEW (LabVIEW-RT) exists for embedded processors.

Because you can't "pop the hood" of LabVIEW and review the source code, formal inspections cannot be done on it. Thorough analysis and testing are highly recommended if LabVIEW is used in safety critical systems.

### 6.4.7 Pascal Language

The Pascal language was originally designed in 1971 by Niklaus Wirth, professor at the Polytechnic of Zurich, Switzerland. Pascal was designed as a simplified version for educational purposes of the language Algol, which dates from 1960. The Pascal language was has been used as a tool to teach structured programming. While there is still a strong subset of Pascal advocates, the language is not commonly used anymore.

The original Pascal standard is ISO 7185 : 1990. The Extended Pascal standard was completed in 1989 and is a superset of ISO 7185. The Extended Pascal standard is ANSI/IEEE 770X3.160-1989 and ISO/IEC 10206 : 1991. Object Oriented Pascal was released as a Technical Report by ANSI in 1993. Object Pascal is the language used with the Delphi Rapid Applications Development (RAD) system.

SPADE Pascal* is a subset that has undergone the study and verification necessary for safety-critical applications. The major issue with Pascal is that no provision for exception handling is provided. However, if a user employs a good static code analysis tool, the question of overflow in integer arithmetic can be addressed and fixed without needing exception handlers. The SPADE Pascal subset is suited to safety-critical applications.

*SPADE PASCAL is a commercially available product and is used here only as an example to illustrate a technical point.

#### 6.4.8 Visual Basic

Visual Basic is a Microsoft version of the Basic language for use with Windows operating systems. It is oriented toward GUIs (Graphical User Interfaces), and is proprietary. However, because Visual Basic is easy to use, many programs that will run under Windows use it as the user interface, and some other language for the "meat" of the program. Visual Basic is a Rapid Application Development (RAD) tool, like Delphi (which uses Pascal).

Features of Visual Basic:

- Strongly typed, if "type checking" is turned on; weakly typed if it is not! Variable types do not have to be declared. The older style of a type suffix on the end of the name (e.g. str$ for a string variable) is still allowed.
- Has a variant data type that can contain data in various formats (numerical, string, etc.). Use of this data type subverts the attempt to enforce strong data typing.
- Component based and not true Object Oriented. A component is a binary package with a polymorphic interface. Other components in the system depend upon nothing but the interface. The underlying implementation can be completely changed, without affecting any other component in the system, and without forcing a re-link of the system. Inheritance is not supported in VB.
- Interpreted environment. The Visual Basic environment checks the syntax of each line of code as you type it in, and highlights these errors as soon as you hit the enter key. Compilers are now available for VB, which speeds up program execution speed.
- Trapping. Visual Basic lets the programmer catch runtime errors. It is possible to recover from these errors and continue program execution.
- The code is hidden from the programmer. This is a strength of Visual Basic, as it makes programming much easier (graphical, drag-and-drop). However, the code is very difficult to inspect, unless the inspectors are intimately knowledgeable about Microsoft Windows and Visual Basic. In many ways, Visual Basic is an automatic code generating program.

### *6.5 Miscellaneous Problems Present in Most Languages*

The following quotations were taken from the Imperial College, London, UK, world wide web home page Dictionary of Computer Terminology, compiled by Denis Howe. It contains graphic descriptions of common problems.

- **aliasing bug**

  <programming> (Or "stale pointer bug") A class of subtle programming errors that can arise in code that does dynamic allocation, especially via malloc or equivalent. If several pointers address (are "aliases for") a given hunk of storage, it may happen that the storage is freed or reallocated (and thus moved) through one alias and then referenced through another, which may lead to subtle (and possibly intermittent) lossage depending on the state and the allocation

history of the malloc arena. This bug can be avoided by never creating aliases for allocated memory. Use of a higher-level language, such as Lisp, which employs a garbage collector is an option. However, garbage collection is not generally recommended for real-time systems.

Though this term is nowadays associated with C programming, it was already in use in a very similar sense in the ALGOL 60 and FORTRAN communities in the 1960s.

- **spam**

  <jargon, programming> To crash a program by overrunning a fixed-size buffer with excessively large input data.

- **heisenbug**

  <jargon> /hi:'zen-buhg/ (From Heisenberg's Uncertainty Principle in quantum physics) A bug that disappears or alters its behaviour when one attempts to probe or isolate it. (This usage is not even particularly fanciful; the use of a debugger sometimes alters a program's operating environment significantly enough that buggy code, such as that which relies on the values of uninitialized memory, behaves quite differently.)

  In C, nine out of ten heisenbugs result from uninitialized auto variables, fandango on core phenomena (especially lossage related to corruption of the malloc arena) or errors that smash the stack.

- **Bohr bug**

  <jargon, programming> /bohr buhg/ (From Quantum physics) A repeatable bug; one that manifests reliably under a possibly unknown but well-defined set of conditions.

- **mandelbug**

  <jargon, programming> /man'del-buhg/ (From the Mandelbrot set) A bug whose underlying causes are so complex and obscure as to make its behaviour appear chaotic or even nondeterministic. This term implies that the speaker thinks it is a Bohr bug, rather than a heisenbug.

- **schroedinbug**

  <jargon, programming> /shroh'din-buhg/ (MIT, from the Schroedinger's Cat thought-experiment in quantum physics). A design or implementation bug in a program that doesn't manifest until someone reading source or using the program in an unusual way notices that it never should have worked, at which point the program promptly stops working for everybody until fixed. Though (like bit rot) this sounds impossible, it happens; some programs have harboured latent schroedinbugs for years.

- **bit rot**

  (Or bit decay). Hypothetical disease the existence of which has been deduced from the observation that unused programs or features will often stop working after sufficient time has passed, even if "nothing has changed". The theory explains that bits decay as if they were radioactive. As time passes, the contents of a file or the code in a program will become increasingly garbled.

  There actually are physical processes that produce such effects (alpha particles generated by trace radionuclides in ceramic chip packages, for example, can change the contents of a computer memory unpredictably, and various kinds of subtle media failures can corrupt files in mass storage), but they are quite rare (and computers are built with error-detecting circuitry to compensate for them). The notion long favoured among hackers that cosmic rays are among the causes of such events turns out to be a myth; see the cosmic rays entry for details.

Bit rot is the notional cause of software rot.

- **software rot**

  Term used to describe the tendency of software that has not been used in a while to lose; such failure may be semi-humourously ascribed to bit rot. More commonly, "software rot" strikes when a program's assumptions become out of date. If the design was insufficiently robust, this may cause it to fail in mysterious ways.

  For example, owing to endemic shortsightedness in the design of COBOL programs, most will succumb to software rot when their 2-digit year counters wrap around at the beginning of the year 2000. Actually, related lossages often afflict centenarians who have to deal with computer software designed by unimaginative clods. One such incident became the focus of a minor public flap in 1990, when a gentleman born in 1889 applied for a driver's licence renewal in Raleigh, North Carolina. The new system refused to issue the card, probably because with 2-digit years the ages 101 and 1 cannot be distinguished.

  Historical note: Software rot in an even funnier sense than the mythical one was a real problem on early research computers (eg. the R1). If a program that depended on a peculiar instruction hadn't been run in quite a while, the user might discover that the opcodes no longer did the same things they once did. ("Hey, so-and-so needs an instruction to do such-and-such. We can snarf this opcode, right? No one uses it.")

  Another classic example of this sprang from the time an MIT hacker found a simple way to double the speed of the unconditional jump instruction on a PDP-6, so he patched the hardware. Unfortunately, this broke some fragile timing software in a music-playing program, throwing its output out of tune. This was fixed by adding a defensive initialization routine to compare the speed of a timing loop with the real-time clock; in other words, it figured out how fast the PDP-6 was that day, and corrected appropriately.

- **memory leak**

  An error in a program's dynamic store allocation logic that causes it to fail to reclaim discarded memory, leading to eventual collapse due to memory exhaustion. Also (especially at CMU) called core leak. These problems were severe on older machines with small, fixed-size address spaces, and special "leak detection" tools were commonly written to root them out.

  With the advent of virtual memory, it is unfortunately easier to be sloppy about wasting a bit of memory (although when you run out of virtual memory, it means you've got a *real* leak!).

- **memory smash**

  (XEROX PARC) Writing to the location addressed by a dangling pointer.

## 6.6 Programming Languages: Conclusions

The "safest" languages are Ada95 (and Ada83) and the SPADE Pascal subset. Ada was specifically created with safety in mind. However, Ada is not the most popular language, and finding and keeping good Ada programmers can be difficult. For this reason, other languages are often chosen.

If choosing any of the other languages, especially C, assembly language, or Forth, be aware of the limitations. Create and enforce a coding standard. Devote extra time to inspections, analysis and test. Educate the developers on the "best" programming practices for that language, and on the pitfalls of the language chosen. Take a proactive approach to reducing errors up front, then test the stuffing out of the software!

## 6.7 Compilers, Editors, Debuggers, IDEs and other Tools

The minimal set of tools (programs) that a software developer needs is:

- **Editor** to create the software (source code) with.
- **Compiler** (or cross-compiler) to create object code with, from the source code.
- **Linker** to create an executable application from the object code.
- **Debugger** to find the location of defects in the software.

Often these tools come bundled in an Integrated Development Environment (IDE), where the developer can shift from editing to compiling/linking to debugging, and back to editing, without leaving the programming environment. Many IDE's have additional tools, or have the ability to add in tools from other vendors. How well the tools can be integrated is something the developer should look at when choosing an IDE. In an embedded environment, the IDE can include simulators for the target hardware, the ability to download the software generated into the target hardware, and sophisticated debugging and monitoring capabilities.

Some IDE's are designed for safety critical software development. For example, DDC-I, a company that specialized in safety-critical software development and tools, has an IDE called SCORE (Safety-Critical Object-oriented Real-time Embedded). "The SCORE development environment has been designed to address the needs of safety-critical, real-time, embedded systems", according to their website. http://www.ddci.com/products/SCORoot.htm

Humans make mistakes, and programmers are only human (despite what some may claim). The goal of a tool is to find as many of the errors as quickly as possible. Some tools help enforce good programming practice. Others make life difficult for the programmer and lead to additional errors, because the programmer is annoyed or is actively subverting the intent of the tool!

In general, look for tools that are:

- Easy to learn.
- Well integrated (if in an IDE) or easy to integrate with other tools. Well integrated means that it is easy to switch between the different tools.
- Default to enforcing standards, rather than relying on the programmer to set the right switches
- Well documented. This includes not only documentation on how to use the tool, but limitations and problems with using the tool. Knowing what the tool *cannot* do is as important as what it *can* do.

A good article on choosing tools for embedded software development is "Choosing The Right Embedded Software Development Tools"[6]

Editors can be a simple text editor (such as Windows NotePad), a text editor designed for programmers (that handles indenting, etc.) or a sophisticated, graphical-interfaced editor. Whatever kind is chosen, look for these features:

- ✓ Can the "style" (such as indentation) be set to match that chosen for the project?

- ✓ Does the editor show language keywords and constructs (including comments) in a different way (e.g. various colors), to help the programmer catch errors such as mistyping a keyword or forgetting to close out a multi-line comment?
- ✓ What kinds of errors can the editor flag?
- ✓ Can the editor support multiple files and search/replace among all of them? Can a variable be tracked across multiple files?

Compilers and linkers usually come together as one piece of software. Cross-compilers run on one system (usually a desktop computer) and produce object code for another processor. When choosing a compiler, consider the following:

- ✓ Can warnings (possible problems) be treated as errors? Can this compiler switch be set as the default mode?
- ✓ Is there a list of defects (bugs) in the compiler? Is there a list of historical defects, and the versions/patches they were corrected in?
- ✓ Can the compiler produce assembly language output? What assembler program is the output targeted to?
- ✓ Does the compiler support integration with a debugging tool? Does it include the option of symbolic information, that allows the debugger to reference high-level language source code?
- ✓ Does the compiler (or cross-compiler) support the particular processor being used? Does it optimize for that processor? For example, if the software will run on a Pentium II, the compiler should optimize the code for that processor, and not treat it like an 80386.
- ✓ Does the compiler offer optimization options that allow you to choose size over speed (or vice versa), or no optimization at all?
- ✓ If used in an embedded system, does the compiler support packed bitmaps (mapping high-level structures to the hardware memory map), in-line assembly code, and writing interrupt handlers in the high-level language?
- ✓ How optimized is the run-time library that comes with the compiler? Can you use “stubs” to eliminate code you don’t need? Is the source code available, so that unneeded code can be stripped out, or for formal verification or inspection?

Debuggers are a vital tool to finding errors, once they’ve reared their ugly little heads. Software debuggers run the defective software within the debugging environment. This can lead to some problems, if the problem is memory violations (out of bounds, etc.), since the debug environment and the normal runtime environment differ. Hardware debuggers (e.g. In-Circuit Emulators) run the code on a simulated processor, with the ability to stop and trace at any instruction.

Debuggers operate by stopping program execution at breakpoints. Breakpoints can be a particular instruction, a variable going to a specific value, or a combination of factors. Once the program is stopped, the environment can be interrogated. You can look at the values of variables, the stack, values in specific memory locations, for example. From the breakpoint, you can single-step through the program, watching what happens in the system, or run until the next breakpoint is triggered.

Debuggers usually need some symbolic information to be included in the object/executable code for them to be able to reference the source code. When debugging, you usually want to be able to see the source code, and not the assembly equivalent. The debugger and the compiler/linker must know how to talk to each other for this to happen.

When evaluating debuggers, consider the following:

- ✓ How well does the debugger and the compiler get along? Will they talk to each other?
- ✓ How much of the system will the debugger let you get at? Can you see memory locations, variable values, and the stack trace? Can you change what's in a memory location or variable?
- ✓ Does the debugger allow you to trace back through procedure calls?
- ✓ Can you trigger on multiple events simultaneously, such as a variable being set to a value while another variable is at a defined value? Can you stop at a memory location only if the value of a variable matches a preset condition?
- ✓ Does the debugger support debugging at the high level language, mixed high level/assembly language, and at the assembly language level?
- ✓ Can the debugger display the high-level data structures used?

In addition to "the basics", these tools are very useful in creating good (and safe) code:

- Lint – finds problems in the code that compilers might miss. Not everything is a true problem, but should be evaluated. If it's "non standard", treat it as an error!
- Profiler – checks speed of program. Good for finding routines that take the most time. Points to areas to where optimization may be useful.
- Memory check programs – find memory leaks, writing outside of bounds.
- Locator – needed for embedded environments, when you must separate what parts go in ROM (program) and what go in RAM (variables, stack, etc.)

## *6.8 CASE tools and Automatic Code Generation*

### 6.8.1 Computer-Aided Software Engineering (CASE)

Computer-aided software engineering (CASE) is a collection of automated tools that support the process of software engineering. CASE can include:

- Structured Analysis (SA)
- Structured Design (SD)
- Code Generators
- Documentation Generators
- Defect Tracking
- Requirements Tracing
- Structured Discourse and Collaboration tools
- Integrated Project Support Environments (IPSEs)

- Inter-tool message systems
- Reverse Engineering
- Metric Generators/Analyzers.

Tools such as editors, compilers, debuggers, Integrated Development Environments may technically be CASE tools, but are usually considered separately. Project management tools (scheduling and tracking) and Configuration Management (Release Management, Change Management (CM)) may also be considered CASE tools.

When CASE was first promoted in the 1980's, the quality of the tools provided was not very good. CASE tools did not cover enough of the software development cycle, did not integrate well with other tools, and were very expensive for what you actually got out of them. While CASE tools are still rather expensive, their quality, reliability, and interoperability have greatly improved. There are even efforts to produce free CASE tool suites.

CASE tools are now classified in three types that describe their functionality and usage. Upper CASE is used during the early phases of software development when initial design, data and process requirements, and the interface design are determined. Requirements analysis and traceability tools, and design tools are included in this classification. Lower CASE tools are primarily those that generate code from the design (output of the Upper CASE tools). These tools can also modify an existing application into a new application with minimal new programming. The third category is integrated CASE (I-CASE), which joins the Upper and Lower CASE tools and helps in the complete software development process.

CASE tools include:

- Analyzers for software plans, requirements and designs
- Methodology support (design, state charts, etc.)
- Model Analysis (consistency checking, behavior analysis, etc.)
- Source code static analyzers (auditors, complexity measurers, cross-referencing tools, size measurers, structure checkers, syntax and semantics analyzers)
- Requirements Tracing
- Design tools (UML modeling, etc.)
- Configuration Management
- System/Prototype simulators
- Requirements-based Test Case Generators
- Test Planning tools
- Test Preparation Tools (data extractors, test data generators)
- Test Execution Tools (dynamic analyzers-assertion analyzers, capture-replay tools, coverage/frequency analyzers, debuggers, emulators, network analyzers, performance/timing analyzers, run-time error checkers, simulators, test execution managers, validation suites)\
- Test evaluators (comparators, data reducers and analyzers, defect/change trackers)
- Reengineering tools

### 6.8.2 Automatic Code Generation

Automatic code generation is one aspect of CASE. It has the advantages of allowing the software to be designed at a higher level then translated, without human error, into source code. The design becomes the "source code".

The downside to automatic code generation is that the tools are only now becoming mature. While human error is eliminated in the translation from design to code, tool error still exists. The correct translation from design to code must be verified for safety critical software. Keep in mind that in some environments, the source code may not be accessible. In addition, how well the code is optimized may affect performance or size criteria.

Code can be automatically generated in several ways:

- Visual languages, such as LabVIEW, have the developer "design" the program graphically. The underlying source code is not visible (or accessible) to the programmer.
- Visual programming environments (e.g. Visual Basic) provide graphical programming, with access to the source code produced. Wizards automatically generate applications or parts of applications based on feedback about the desired features from the programmer. The wizards automatically generate code based on this feedback.
- Generating code from design models. These models usually use a set of diagrams that depict the structure, communication, and behavior of the system. The model may be supplemented with text-based specifications of system actions, such as computations. Design methodologies that can be used for code generation include the following. Not all tools or approaches will support all design modeling methodologies.
    - Unified Modeling Language (UML)
    - Object Modeling Technique (Rumbaugh)
    - Object-Oriented Software Engineering (Jacobson)
    - Object-Oriented Analysis and Design (Booch)
    - Specification and Description Language (SDL)
    - Real-time Object-Oriented Method (ROOM)
    - Object-Oriented Analysis (OOA – Shlaer and Mellor)
    - Harel's hierarchical statecharts

#### *6.8.2.1 Visual Languages*

A visual language is one that uses a visual syntax, such as pictures ore forms, to express programs. Text can be part of a visual syntax as well. LabVIEW by National Instruments, VEE by Hewlett Packard, and PowerBuilder (Austin Software Foundry) are examples of visual languages.

Visual languages are wonderful for prototyping applications, especially when the user interface is important. The development can be "participatory", with the users and developers sitting down at a machine and designing the application interface together.

A problem with visual languages in safety critical applications is the inability to inspect the "code". What happens between the graphical program creation and the operations of the program is a black box. In addition, little formal development is done when visual languages are used. Formal specifications are usually lacking or non-existent. Configuration control is often not considered, and configuration management tools may have problems with the visual representations ("language").

### *6.8.2.2 Visual Programming Environments*

A visual programming environment (VPE) uses a visual representation of the software and allows developers to create software through managing and manipulating objects on a visual palette. Examples are Visual Basic (Visual C++, and other "visual" languages) and Delphi (by Borland).

A visual programming environment uses a graphical interface to allow the developer to construct the software. From the visual elements (often the user interface), code is generated in the appropriate language. The developer must hand-code the interactions between elements, and must hand-code all the "guts" of the program. This is very close to traditional programming, with the addition of easily creating graphical user interfaces. In fact, VPE's can be used to create "regular" programs without the fancy user interface, or to hand-code the user interface if desired.

Since VPE's produce source code, it can be formally inspected and analysis tools can be used with it. However, since the code was not generated by the developers, it may not follow the style or coding standards of the development team. The source code may be difficult to follow or understand, and its relationship back to the graphical environment may not always be obvious.

### *6.8.2.3 Code Generation from Design Models*

Model-based code generation produces application source code automatically from graphical models (designs) of system behavior or architecture. One advantage of model-based development is to raise the level of abstraction at which developers can work. The design (model) becomes the program, and only the design has to be maintained. "Code Generation from Object Models" [3] discusses the various approaches to code generation for object-oriented systems, and gives some of the plusses and minuses of using each approach.

In many ways, the moved to model-based code generation parallels the move from assembly to high-level languages. Each move along the path is a step up the abstraction ladder. Each step frees the developer from some of the gritty details of programming. However, each step also brings with it challenges in verifying that the program is safe!

The methodology and tools go hand in hand. Some tools support multiple design methodologies, some only support one. When choosing a methodology (and tool), consider:

- The suitability of the modeling language for representing the problem. (How good is the modeling methodology for your particular problem?)
- The sufficiency of modeling constructs for generating code (how much of the code can it generate, how much will have to be hand coded?)

- The maturity of translators for generating quality code (Have the translators been used for years, or created yesterday? How much analysis has been done to verify the software that is produced by the translators?)
- Tools for development tasks related to code generation (Does it integrate with the debugger?)
- Methodologies for employing code generation effectively (What method does the tool use to translate your design into code?)
- The selection of tools and methods appropriate to the application (What's the right method for the problem? What's the right tool for the development environment? Do they match (best tool works with best methodology)?)
- The language the tool produces source code in (Is it Ada? C++?).

For object-oriented systems, there are three approaches to model-based code generation:

- The **Structural** approach is based on state machines. It can create an object framework from the model design. Dynamic behavior and object communication is added by the programmer (hand coded). This includes hard deadlines and interrupt handling. This approach is used with UML and the Rumbaugh, Jacobson, and Booch OO design techniques. Most tool vendors support this approach. The tools usually can be used to reverse engineer can be done on existing code as well.
- The **Behavioral** approach is based on state machines augmented with action specifications. It includes both static and dynamic behavior (state transitions). Specification and Description Language (SDL – telecommunications standard) and UML (among other methods) support this approach. What needs to be hand coded are event handlers and performance optimizations. Developers must adopt a state-machine view of the system functionality in addition to an object view of the system structure. Because the behavior is fully specified, a simulated model for test/debug can be created prior to code completion.
- The **Translative** approach is based on application and architecture models that are independent of one another. The application model uses Object-Oriented Analysis (OOA) by Shlaer and Mellor. This approach can simulate the system before developing code (same as behavioral). The architecture model is a complete set of translation rules that map OOA constructs onto source code and implementation (run-time) mechanisms.

Some tools support other design methodologies. Structured analysis and design can be used to create code frames for the system structure. The frames have to be fleshed out by the developer, however. Also, data flow diagrams can be used by several tools. Once can produce code for Digital Signal Processors (DSP). The code generated implements the flow of system. Processing steps are either hand-coded or standard library routines. Data flow diagrams are used in specific application tools for control systems, instruments (e.g. LabVIEW), and parallel data processing.

In an ideal world, the CASE tool would be certified to some standard, and the code generated by it would be accepted without review, in the same way that the object code produced by a compiler is often accepted. However, compilers produce errors in the object code. Automatically generated code is in its infancy. When the code is safety critical, or resides in an unprotected partition with safety critical code, the automatically generated code should be subjected to the same rigorous inspection, analysis, and test as hand-generated code.

## 6.9 Software Configuration Management

Software configuration management (SCM) is often considered a part of project management and not software development or testing. It is a vital part of the development process, however, that should not be overlooked. It is very unlikely that you can produce "safe" software without it. You certainly cannot convince the Quality/Product Assurance/Safety personnel that the software is safe if you haven't implemented SCM!

Software Configuration Management (SCM) is much more than just version control of source code. It is a process to maintain and monitor the software development process as well. SCM includes:

- Identification: an identification scheme is needed to reflect the structure of the product. This involves identifying the structure and kinds of components, making them unique and accessible in some form by giving each component a name, a version identification, and a configuration identification.
- Control: controlling the release of a product and changes to it throughout the lifecycle by having controls in place that ensure consistent software via the creation of a baseline product.
- Status Accounting: recording and reporting the status of components and change requests, and gathering vital statistics about components in the product.
- Audit and review: validating the completeness of a product and maintaining consistency among the components by ensuring that components are in an appropriate state throughout the entire project life cycle and that the product is a well defined collection of components.

Often, project documentation (specifications, plans, etc.) are maintained by one person, and the source code version control is handled by the programmers. This is not a good idea unless the tools used are well integrated, and someone has control of the process. Having everything accessible in one location facilitates the status accounting and audit/review process. Having someone else in control of source code may force developers to document the changes completely. Simply having an "outside" eye on the process improves the chance of finding errors or potential problems (two developers working on the same module but for different change requests, for example).

Software Configuration Management is usually performed using a tool (program). However, a file or folder should be maintained, to collect information that is not in electronic form. This information could include the design notes scribbled on a napkin or a fax that only exists in hard-copy. The point is to collect all pertinent information in one place. It would be good if the information is cataloged in the electronic SCM system, so that it can be found again when needed.

### 6.9.1 Change control

Change control is an important part of developing safe software. Arbitrary changes, because a developer thought they would be more efficient, for example, should be avoided. Once a piece of software has reached a level of maturity, it should be subject to a formal change control process. What that level of maturity is will vary by group. It could be when the module compiles, when the CSCI (which may contain several modules) is completed, or when the whole program is at its first baseline.

Formal change control usually includes a form to request a change (Software Change Request, Engineering Change Request, etc.). The form is filled out by the developer, the customer, or someone else involved in the project. The form should include both what should be changed and why. A Change Control Board (also called an Engineering Review Board, and by other names) is convened to evaluate the change request. The board consists of several people, including someone from Software Quality Assurance. When safety is an issue, someone from safety or risk management should also be included on the board. The requestor may be at the CCB meeting, or the board may just evaluate the submitted form. The board may approve the change, reject it, combine it with other requests, or suggest a modification.

Another way software changes occur is through a problem reporting/corrective action (PRACA) process. A PRACA is issued during the operation of the software, usually during testing. If the software isn't operating as it should, a PRACA is written. The problem report goes to the developers, who must find out what the problem is. If the fix to the problem involves a change to the software, it must go through the Change Control Board.

All the paperwork from the Change Control process should be archived in the configuration management system. This includes software requests, PRACAs, notes from CCB meetings, and any other pertinent information. The configuration management system provides a repository for storing this data for later retrieval.

In addition, a cross-index should be created between software changes, requirements, code module versions, and tests. This could be a database, a spreadsheet, or some other format. Being able to know what modules a software change affected impacts what tests need to be run. The change may also indicate that a requirement changed, and that the software requirements document needs to be updated.

### 6.9.2 Versioning

Versioning is the part of software configuration management that most people think of first. It involves archiving the source code, keeping previous versions when a new version is added to the SCM tool. Sometimes a complete previous version is kept, other tools use a "delta" (difference) from the previous version to the new version.

Each module will have a version number associated with it. A "release" will consist of all the modules and their associated version numbers. Some SCM tools allow branching, where a release will go down two or more paths (perhaps a "freeware" version and a complete version, for example). Versioning keeps the changes straight, and also allows "roll back" to previous versions if a bug is found down the road.

Most SCM tools also have a check-in/check-out policy, to prevent changes by multiple programmers on the same module. Some will allow only one programmer to work on the module at one time. Other SCM tools will do a "merge" when multiple developers check in the same module.

One weakness of many SCM tools is that the programmer can get away without good documentation on what changes were made and why. The tool keeps the changes, but the reasoning behind it usually is added as a comment upon check-in of the module. Some tools force the developer to say *something*, but not necessarily something useful! At a minimum, when a module is changed the following should be done:

- Clearly identify the area of code that is changed (within the source code). Use a comment with some character string (such as *****) that is easy to spot when flipping through the source code. Identify the end of the changed area the same way.
- Have a header at the top of the altered code that includes why the change occurred (change request, PRACA, etc.), what was changed (in English, not code-ese), when it was changed, and by whom.
- Include the what/when/why/who information in the module check-in comment. This information can be extracted for status accounting (see below).

### 6.9.3 Status Accounting

According to MIL-STD-482A, configuration status accounting is "The recording and reporting of the information that is needed to manage configuration effectively, including a listing of the approved configuration identification, the status of proposed changes to configuration, and the implementation status of approved changes."

Status Accounting answers the question "how completely ready is the software?". Decide what stages of incompleteness, correctness and obsoleteness need to be known about each item and to what audience, give each stage a status name (e.g. draft, under review, ready for integration/delivery, operational, superseded), and collect the status of each item. Collate the information into a human-understandable format.

Part of status accounting is the ability to create reports that show the status of each document (version, whether it is checked-out, when it was last updated, who made the changes, and what was changed). The status of change requests and PRACAs are also included in status accounting.

While the status information can be compiled by hand, it can be a tedious process. Many tools exist that provide an integrated configuration management system for all kinds of documents, including source code, and that can generate the status reports when requested. Some of these tools are free or low-priced.

The configuration management system need to be "audited" occasionally. The audit can be a formal affair, or an informal look at the system by someone other than the configuration manager, such as Software Product Assurance or Software QA. The purpose of the audit is to verify that what the status accounting says about the project is actually true, and to look for holes in the process that can lead to problems in the future.

### 6.9.4 Defect Tracking

Defect (bug) tracking is sometimes handled outside of Software Configuration Management. However, integrating it with the SCM process facilitates the "keep it all in one place" philosophy. When it's the middle of the night and you're trying to find information on a bug you thought you had killed a week ago, you'll appreciate a well-ordered system.

Defect tracking has several purposes. One is to record all the defects for future reference. This can be simply for historical purposes, or to have something to reference when you think you've seen this defect before. Having defect information from previous projects can be a big plus when debugging the next project.

Recording the defects allows metrics to be determined. One of the easiest ways to judge whether a program is ready for serious safety testing is to measure its defect density—the number of defects per line of code. If testing has found the majority of defects, then the software is likely to be stable. Safety testing would "put the software through its paces", usually by generating error conditions and verifying graceful behavior by the program. You don't want to stress the software until you're pretty sure most of the bugs have been found.

To determine the defects per lines of code, you need to know two pieces of information, both of which can be extracted from a good configuration management system: lines of code and number of defects. You also need a "history" from other projects on defects/lines of code (from your projects, or general industry numbers). If the average defects/KLOC (thousand lines of code) is 6, and the software is 10,000 LOC, then about 60 defects exist in the software. If testing has only found 10, a lot more testing needs to be done. The software in the example has a high risk, because many more defects linger in the code.

One question in defect tracking is whether to use bugs found during unit testing by developers. It would be best if those defects are documented. The developer can see if he has a tendency to a certain kind of bug. Other programmers can learn from the experience of the developer and avoid similar defects. And a better idea of the number of defects (per KLOC, per complexity, etc.) can be determined. Developers often resist documenting the unit test bugs, however. Make the process easy on them, and make sure management does not have access to the information for a specific developer!

### 6.9.5 Metrics from your SCM system

Keeping tabs on the various elements of your software development project can show when the project is getting into trouble (cost, schedule, cannot meet delivery date, etc.) and can aid in planning future projects. Items to track, if possible, are:

- Lines of code* for the project (total)
- Lines of code per module, average module size, distribution of sizes
- Complexity per module, average complexity, distribution of complexities
- Estimated and actual time to complete development (coding, unit testing, etc.) for a change request or PRACA
- Estimated and actual time to code a module.

- Estimated and actual time to unit test a module.
- Estimated and actual time for integration tests (black box) and system tests
- Number of defects found per test type (unit, integration, system, etc.) You may wish to categorize the defects for further breakdown.

* Function points can be substituted for Lines of Code, or both numbers can be collected.

From these raw inputs, you can determine (among other things):

- Number of defects per lines of code for your team/organization
- How good your estimations are for completion of a software change
- How much time it takes to unit testing. Correlate it with the defects/LOC to see if you need to spend more, or less, time unit testing.
- How much time to estimate for the various development phases (design, coding, testing) for your next project.
- How much time it will take to update the software for a future change request.
- Where to put extra resources in testing. If you are finding the majority of your defects in system testing, spend more time in unit and integration testing to find the defects earlier.
- If you made a software development process change, the numbers may show how much of an improvement the change made.

### 6.9.6 What to include in your SCM system

- Documents and Plans (specifications, formal design documents, verification matrix, presentation packages, etc.)
- Design information (data flow charts, UML or object-oriented design products, inputs to automatic code generation programs, etc.) Also include any miscellaneous related information.
- Interface information (Interface Control Documents, flow charts, message formats, data formats, etc.)
- Source Code
- Test Cases/Scenarios
- Test scripts, for manual or automated testing
- Test Reports
- Defect lists (or defect database)
- Change requests
- Problem reports/corrective actions

- Information for metrics, such as lines of code, number of defects, estimated and actual start or completion dates, and estimated/actual time to complete a change.

## 6.10 Operating Systems

### 6.10.1 Types of operating systems

Operating Systems (OS) are the software that runs the programmers applications. The OS is loaded when the system boots up. It may automatically load the application program(s), or provide a prompt to the user. Examples of operating systems are MS-DOS, Windows 9x/NT, and Macintosh OS 9.

Not all systems use an operating system. Small, embedded systems may load the application program directly. Or a simple "scheduler" program may take the place of the OS, loading the appropriate application (task) and switching between tasks.

The types of operating systems are:

- **No operating system.** Just a boot loader (BIOS) program that loads the application directly and gives it control. The application deals with the computer hardware (processor) as well as any attached hardware, directly. This is sometimes used in small, embedded systems.
- **Simple task scheduler**. Usually written by the software developer. Acts as a mini-OS, switching between different applications (tasks). No other OS services provided.
- **Embedded OS.** This will be a fully-functional operating system, designed with small systems in mind. It will take a minimal amount of storage space (both RAM and disk). It provides task switching, some method for inter-task communications, shared resources, and other basic OS functions.
- **Real-time Operating System.** An RTOS is usually designed for embedded systems (small size), but this is not necessary for it to be "real-time". An RTOS has timing constraints it must meet. If it cannot respond to events (interrupts, task switches, completing a calculation, etc.) within a specific time, then the result is **wrong**. "Soft" real-time systems have some flexibility in the timing. "Hard" real-time systems have no flexibility for critical deadlines.
- **"Regular" Operating System**s. These systems have no timing constraints and are designed for systems that do not have limited resources. Examples are the Windows variants that run on PCs, Macintosh's OS 9, and main-frame operating systems.

### 6.10.2 Do I really need a real-time operating system (RTOS)?

If an operating system is selected for use in the safety critical system, it will most likely be an RTOS. Even if the timing aspects aren't important, most non-real-time operating systems are not designed for safety critical environments. What may be acceptable on your desktop (program freezes, frequent rebooting, or the blue screen of death) is not acceptable when safety is involved.

The first question to answer is: Do I need an operating system? Small projects often use just a "boot loader" to boot up the system and load an application program. "Get by without an RTOS" by Michael Melkonian [2] describes a method that provides most operating system functionality. For small projects, such a system may be the best option. It avoids the overhead of having to learn an RTOS. And since commercial operating systems are COTS software, they would require extra analysis and testing in a safety critical application.

Once you determine that you need (or want) an operating system, the next question is: build, reuse, or buy? Do you create your own operating system, reuse an existing, proprietary one, or purchase a commercial OS? If you have an existing OS that was used in safety critical applications before, or that has been thoroughly tested, it may be best to use that. Building your own OS is not an easy option. The advantage is that you can build in only what you need, eliminate options that might affect safety, and do formal development and/or thorough testing. For many systems, purchasing a commercial OS is the most cost-effective choice. This has the disadvantages associated with Off-The-Shelf software in general, but the advantages of time and money. The developers can spend time developing the applications, and not creating the operating system.

### 6.10.3 What to look for in an RTOS

What makes an OS a RTOS?

1. An RTOS (Real-Time Operating System) has to be multi-threaded and preemptible.
2. It must support a scheduling method that guarantees response time, especially to critical tasks.
3. Threads (tasks) must be able to be given a priority (static or dynamic). An alternative would be a deadline driven OS.
4. The OS has to support predictable thread synchronization mechanisms (semaphores, etc.)
5. A system of priority inheritance has to exist.
6. OS behavior should be known. This includes the interrupt latency (i.e. time from interrupt to task run), the maximum time it takes for every system call, and the maximum time the OS and drivers mask the interrupts. The developer also need to know the system interrupt levels and device driver parameters (IRQ levels, maximum time within a device IRQ, etc.).

Every system is unique, and there is no simple universal set of criteria for selecting an operating system. Some commonly encountered issues to consider in the selection process are:

- **Memory management:**

  Operating systems which support a dedicated hardware MMU (Memory Management Unit) are superior from a safety viewpoint. An MMU guarantees protection of the designated memory space. In a multitasking application, it ensures the integrity of memory blocks dedicated to individual tasks by preventing tasks from writing into each others' memory space. It protects each task from errors such as bad pointers in other tasks.

  For small, single task systems (such as those running on a microcontroller), an MMU may not be needed. In such cases, even more than a minimal operating system may be overkill.

- **Determinism:**

  Determinism is the ability of the operating system to:

  - Meet deadlines
  - Minimize jitter, (i.e. variations in time stamping instants, the difference between the actual and believed time instant of a sample)
  - Remain steady under dynamic occurrences, e.g. off nominal occurrences,
  - Bounding of priority inversion (time the inversion is in place).

- **Priority inversion**

  Priority inversion is a temporary state used to resolve priority conflicts. A low priority task is temporarily assigned a higher level priority to ensure its orderly completion prior to releasing a shared resource requested by a higher priority task. The priority change occurs when the higher priority task raises a "semaphore" flag. It is vital that the lower priority task releases its shared resource before delay of the higher priority task causes a system problem. The period of temporary increase of priority is called the "priority inversion" time. Priority inversion time can be defined by the application.

- **Speed**

  The context switching time is the most important speed issue for real-time operating systems. Context switching is how quickly a task can be saved, and the next task made ready to run. Other speed issues are the time it takes for a system call to complete.

- **Interrupt latency**

  Interrupt latency is how fast an interrupt can be serviced.

- **Method of scheduling:**

  The method of scheduling can be predetermined logical sequences (verified by Rate Monotonic Analysis). It can be priority-based preemptive scheduling in a multitasking environment (such as UNIX, Windows NT or OS2). Another method is "Round Robin" time slice scheduling at the same priority for all tasks. “Cooperative” schedule can also be used, where the task keeps control until it completes, then relinquished control to the scheduler. Cooperative scheduling may not be a good idea in a safety critical system, as a task can “hog” the processor, keeping the safety critical code from running.

- **POSIX compliance**(1003.1b/c).

  POSIX compliance is a standard used by many operating systems to permit transportability of applications between different operating systems. For single use software, or software that will never be used on other operating systems, this is not as important.

- **Support for synchronization**

  What support for synchronization and communication between tasks does the OS use? How much time does each method take?

- **Support for tools**

  Does the OS have support for tools such as debuggers, ICE (In Circuit Emulation) and multi-processor debugging. Consider also the ease of use, cost and availability of tools.

- **Support for multiprocessor**

  Does the OS support a multiprocessor configuration (multiple CPU's) if required.

- **Language used to create the OS**

  Consider the language in which the operating system kernel is written, using the same criteria as selecting application programming languages.

- **Error handling in OS system calls**

  How does the OS handle errors generated within system calls? What does the application need to check to verify that a system call operated correctly? Does the OS return an error code or can it access a user-created error handler?

- **Safety Certifications**

  Some operating systems go through levels of safety-related certification, often for use in medical or aviation applications.

### 6.10.4 Commonly used Operating Systems

The following is a list of Operating Systems used in embedded or real-time systems. This is not a complete list, and nothing is meant by the ordering, inclusion, or exclusion of any OS. If the OS has been certified to any safety standard, that will be mentioned.

- VxWorks (Wind River Systems, http://www.windriver.com/) – This is a popular RTOS with a tool-rich integrated development environment. There is a version of VxWorks certified to DO-178B, a standard used for aviation software. VxWorks is available for most higher-level processors.
- OSE (http://www.enea.com) – This operating system is certified to the safety standard IEC 61508. It is also being certified to DO-178B. OSE also provides an integrated development environment with many tools useful to embedded or real-time applications. OSE supports many processors, including DSPs.
- VRTX (http://www.mentor.com/embedded/vrtxos/) - VRTX is a "a high-performance, modular, DO-178B-certifiable solution for PowerPC, ARM, 68K, CPU32, CPU32+, M·CORE, and 80x86-based embedded systems." This RTOS is scalable, from a microkernel to a full-up operating system with POSIX, Java, priority inheritance, and other features.
- PSOSystem 3 (http://www.windriver.com/products/html/psosystem3.html) – This RTOS is now owned by Wind River. "pSOSystem™ 3 is a modular, high-performance, memory protected, highly reliable real-time operating system, designed specifically for embedded microprocessors" according to Wind River. pSOSystem supports the PowerPC and MIPS families of processors.
- QNX (http://www.qnx.com/) – This RTOS supports x86 (80386 and higher) processors only. It uses a microkernel with minimal, required functionality that can be extended by dynamically plugging in service-providing processes. A free, non-commercial version is available for download and evaluation.

- CMX (http://www.cmx.com/) - CMX provides both a full-featured RTOS and a "tiny" one that runs on lower-level microcontrollers (with as little as 512 bytes of RAM). The RTOS supports a wide range of microprocessors.
- OS-9 (http://www.microware.com/Products/Software/OS9.html) – According to Microware, "OS-9® is a system-secure, fault-tolerant RTOS with high availability and reliability. Users can dynamically add and replace modules while the system is up and running." OS-9 supports many higher-level processors.
- AMX (http://www.kadak.com/) - This small, compact RTOS runs on x86, 68K family, Coldfire, ARM, PowerPC, and Z80 architectures. Depending on the processor, AMX can fit in 12K to 36K of ROM, with 2K to 4K of RAM needed. Besides the standard RTOS services, AMX claims rapid task context switching and fast interrupt response. Timing information is given on their website.
- LynxOS (http://www.lynuxworks.com/products/whatislos.html) - LynxOS is a Linux-compatible real-time operating system that "is a hard RTOS that combines performance, reliability, openness, and scalability together with patented technology for real-time event handling." It supports processors from Intel, Motorola, and MIPS.
- RTEMS (http://www.rtems.com/) – RTEMS is a free, open source operating system. It was originally developed for the U.S. Army Missile Command. It contains all the basics of an RTOS, and the source code is available. RTEMS supports most commonly used processors.
- Linux (http://www.linux.org and http://www.embedded-linux.org/) - Linux is the "open source" version of Unix. It is not normally a real-time operating system, but there are versions developed for embedded systems. In addition, there are extensions to Linux for real-time systems. Linux has been ported to many processors, though it is not usually available for "cutting edge" processors. You can "roll your own" version of Linux, creating a smaller system with only the elements you need.
- Windows NT/2000 (http://www.microsoft.com) - Windows NT (and its descendent, Windows 2000) are general purpose operating systems that have many "real-time" abilities. Out of the box, neither are "hard" real-time systems, but several companies provide extensions that meet the "hard" timing criteria. Windows NT/2000 are more robust than the desktop versions of Windows (95,98), with fewer memory leaks and greater memory protection.
- Windows CE (http://www.microsoft.com) - Microsoft describes Windows CE 3.0 as "the modular, real-time, embedded operating system for small footprint and mobile 32-bit intelligent and connected devices that enables rich applications and services.". It supports many 32-bit microprocessors. It contains many "real-time" aspects, such as task priority, priority inheritance, and nested interrupts. It is not "hard" real-time, however.

## 6.11 Distributed Computing

Having multiple processors working together may "share the load" of a complex calculation, or may distribute the appropriate part of a problem to a processor optimized for that particular calculation or control. Such "multi-brained" systems constitute a distributed computing system.

Distributed systems can be defined as two or more independent processors, working together, and communicating across a medium that may have substantial transmission delays.

Distributed computing is used ~~in~~for many different purposes. In telecommunications systems, ~~for~~… For complex computational ~~problems (parallel processor), and for~~problems, parallel processors or clustered processors are used. Distributed computing is also used when high availability is required (continuing on if one processor has a hard failure)~~, among other areas~~.

Distributed systems may reside on the same processor board (multiprocessors), in the same system, or in widely separated areas. The processors in a distributed system usually use one of two main methods to communicate: shared memory or message passing. Shared memory systems have real or simulated RAM available to all the processors, and use this to communicate (pass values, signal use of resources, etc.). Shared memory is normally used in distributed systems that are physically compact (processors are near each other) and tightly coupled, such as multiprocessor systems.

Shared memory allows large or complex data structures to be easily communicated between processes. Issues with shared memory distributed systems are

- Data consistency. The consistency of the data in shared memory (accuracy at any given moment) is a problem because of network latency. Most processes will cache the shared memory to improve performance. The value in Process A's cache may be outdated, because process B has updated it, but delays may lead the update information to arrive after A has read the value. Various scenarios are implement to prevent this.

- Access synchronization. Distributed systems must also provide ways to prevent multiple processes from accessing the shared data at the same time. Usually a locking mechanism is used.

- Address space structure. A system may use a single shared distributed address space, where all the processes appear as threads within this space. The advantages is that objects appear at the same addresses on all nodes. However, security and protection are a major problem in such systems. Another approach divides each process's address space into fixed regions, some of which are shared and the rest are private. Shared data may or may not appear at the same address in each process.

- Fault tolerance. Distributed shared memory systems have some problems with fault tolerance. Most systems ignore it or maintain that it is an operating system issue. If one node that is sharing data with other processes fails, all the connected sites may fail as well.

Message passing distributed systems communicate via a network, sending and receiving messages. Messages are blocks of data, usually wrapped in a protocol layer that contains information on the sender and recipient, time stamp, priority, and other parameters. A distributed system may pass messages synchronously or asynchronously. In synchronous message passing, the system has two phases: delivery and local computation. During the delivery phase, each process may send one or more messages to its immediate neighbors. The local computation phase encompasses receiving messages, state changes, and queuing messages to be sent during the next delivery phase.

Asynchronous message passing distributed system do not have phases where messages are transmitted. Any process can send or receive messages at any time. Propagation delays are arbitrary, though in real systems they are often bounded (given a maximum delay value).

The underlying network in a distributed system may or may not guarantee that all messages are eventually delivered to the correct recipient and that the messages will be without error or duplication.

A distributed computing system may be fixed (known processors and processes that never change, except for failure) or dynamic. In a dynamic system, new processes can be added to the network and other processes can leave at arbitrary times. The protocols (message passing, usually) must adapt to the changing topology.

Nodes (processes) in a distributed system can fail completely, intermittently (where the node operates correctly some of the time and at other times fails), or randomly (Byzantine). In the Byzantine failure mode, the node behaves arbitrarily, sending valid-looking messages inconsistent with the state it is in and the messages it has received.

Failures can occur in the communications medium. Links between nodes can be broken. Intermittent problems may lead to the loss, garbling, reordering, or duplication of messages. Delays in message transfer may be interpreted as a lost message, or the data in the message, when it finally arrives, may be out of date.Breaking up software across (possibly) diverse multiple processors, communicating through some medium (serial line, network, etc.), creates a complex environment, full of potentially complex errors. Distributed systems have an *inherent* complexity resulting from the challenges of the latency of asynchronous communications, error recovery, service partitioning, and load balancing. Since distributed software is *concurrent* as well, it faces the possibility of race conditions, deadlock, and starvation problems. An excellent article that discusses the complexities of distributed computing is “Distributed Software Design: Challenges and Solutions” [13], which~~discusses~~ describes some problems inherent to distributed computing :

- **Processing site failures.** Each processor in a distributed system could fail. The developer must take this into account when building a fault-tolerant system. The failure must be detected, and the other processors must “pick up the slack”, which may involve reallocating the functionality among the remaining processors or switching to another mode of operation with limited functionality.
- **Communication media failure (total loss, or loss between links).** If the communication medium goes down, one or more processors are isolated from the others. Depending on how they are programmed, they may undertake conflicting activities.
- **Communication media failure (intermittent).** Intermittent failures include loss of messages, reordering of messages or data (arriving in a different order than when sent), and duplicating messages. They do not imply a hardware failure.
- **Transmission delays.** A delayed message may be misconstrued as a lost message, if it does not arrive before a timeout expires. Variable delays (jitter) make it hard to specify a timeout value that is neither too long nor too short. Delayed messages may also contain out-of-date information, and could lead to miscalculations or undesirable behavior if the message is acted on.

- **Distributed agreement problems.** Synchronization between the various processors poses a problem for distributed systems. It is even more difficult when failures (intermittent or complete) are present in the system.
- **Impossibility result.** It has been formally proven that it is not possible to guarantee that two or more distributed sites will reach agreement in finite time over an asynchronous communication medium, if the medium between them is lossy or if one of the distributed sites can fail.
- **Heterogeneity.** The processors and software involved in a distributed system are likely to be very different from each other. Integration of these heterogeneous nodes can create difficulties.
- **System establishment.** A major problem is how distributed sites find and synchronize with each other.

~~**[This is a placeholder for Distributed Systems. More needs to be added.]**~~More information on distributed computing problems and solutions can be found at:

1) “Distributed Software Design: Challenges and Solutions” by Bran Selic, Embedded Systems Programming, Nov. 2000

2) FINITE STATE MACHINES IN DISTRIBUTED SYSTEMS, Class 307. Speaker: Knut Odman, Telelogic

3) Distrib. Syst. Eng. **3** (1996) 86–95. Printed in the UK, Implementing configuration management policies for distributed applications, Gerald Krause y and Martin Zimmermann

### 6.12 Programmable Logic Devices

Until recently, there was a reasonably clear distinction between hardware and software. Hardware was the pieces-parts: transistors, resistors, integrated circuits, etc. Software ran on the hardware (operating systems, applications programs) or resided *inside* the hardware (firmware). The design, construction, and testing process for hardware and software differed radically.

Programmable logic devices (PLDs) blur the lines between hardware and software. Circuitry is developed in a programming language (such as VHDL or Verilog), run on a simulator, compiled, and downloaded to the programmable device. While the resulting device is “hardware”, the process of programming it is “software”. Some versions of programmable devices can even be changed “on the fly” as they are running.

Programmable logic is loosely defined as a device with configurable logic and flip-flops, linked together with programmable interconnects. Memory cells control and define the function that the logic performs and how the logic functions are interconnected. PLDs come in a range of types and sizes, from Simple Programmable Logic Devices (SPLDs) to Field Programmable Gate Arrays (FPGAs).

System safety normally includes hardware (electronic) safety. However, given the hybrid nature of programmable logic devices, software safety personnel should be included in the verification of these devices. Because PLDs are hardware equivalents, they should be able to be verified (tested) in a normal "hardware" way. However, because they are programmed devices, unused or unexpected interconnections may exist within the device as a result of software errors. These "paths" may not be tested, but could cause problems if accidentally invoked (via an error condition, single event upset, or other method). As the PLDs become more complex, they cannot be fully and completely tested. As with software, the process used to develop the PLD code becomes important as a way to give confidence that the device was programmed properly.

The variety of programmable logic devices are described in the sections below. Guidance is given on the safety aspects of verifying each type of device.

"Frequently-Asked Questions (FAQ) About Programmable Logic" [8] provides good introductory information on PLDs. An article in Embedded Systems programming [9] also gives a good introduction.

### 6.12.1 Types of Programmable Logic Devices

Simple Programmable Logic Devices (SPLDs) are the "original" Programmable Logic Devices. These are the smallest of the PLDs – each can replace only a few logic chips. Inside a PLD is a set of macrocells, each of which are composed of some amount of logic (AND gate, for example) and a flip-flop. Each macrocell is fully connected. SPLD types include PAL (Programmable Array Logic), GAL (Generic Array Logic), PLA (Programmable Logic Array), and PLD (Programmable Logic Device).

Complex Programmable Logic Devices (CPLDs) have a higher capacity than the SPLDs, typically equivalent to 2 to 64 SPLDs. The macrocells within a CPLD may not be fully connected, so not all theoretically possible designs may be a implementable in a particular CPLD. Varieties of CPLDs include EPLD (Erasable Programmable Logic Device), PEEL, EEPLD (Electrically-Erasable Programmable Logic Device) and MAX (Multiple Array matrix).

Field Programmable Gate Arrays (FPGAs) have an internal array of logic blocks, surrounded by a ring of programmable input/output blocks, connected together via programmable interconnects. These devices are more flexible than CPLDs, but may be slower for some applications because that flexibility leads to slightly longer delays within the logic.

### 6.12.2 "Program Once" Devices

"Program once" devices require an external programming mechanism and cannot be reprogrammed, once inserted on the electronics board. Included in this category are erasable/reprogrammable devices and "on-the-board" reprogrammable devices where the ability to reprogram is removed or not implemented, as well as true "write once" devices. Simple Programmable Logic Devices (SPLDs) nearly always are "program once". Depending on the underlying process technology used to create the devices, CPLDs and FPGAs may be "program once", "field reprogrammable", or fully configurable under operating conditions.

With "program once" devices, safety only needs to be concerned with the resulting final chip. Once the device is verified, it will not be changed during operations.

Simple Programmable Logic Devices (SPLDs) are fairly simple devices. Don't worry about the development process, just test as if they were "regular" electronic devices. Treat them as **hardware**.

Complex Programmable Logic Devices (CPLDs) and Field Programmable Gate Arrays (FPGAs) are complex enough that unexpected connections, unused but included logic, or other problems could be present. Besides a complete test program that exercises all the inputs/outputs of the devices, the software should be developed according to the same process used for regular software, tailored to the safety-criticality of the device/system. Requirements, design, code, and test processes should be planned, documented, and reviewed. For full safety effort, analyses should be performed on the documents from each stage of development.

### 6.12.3 "Reprogram in the Field" Devices

Both Complex Programmable Logic Devices (CPLDs) and Field Programmable Gate Arrays (FPGAs) come in a in-the-field-programmable variety. The internals of these devices is based on EEPROM (Electrically-Erasable Programmable Read Only Memory) and FLASH technology. If the circuitry is present on the board (and implemented within the chip), then these devices can be reprogrammed while on their electronics board. This is not "on-the-fly" reprogramming while in operation. Reprogramming erases what was there and totally replaces what was in the chip.

Once scenario that might be used is to hook up the CPLD/FPGA "reprogramming" circuitry to an external port, such as a serial port. During development, an external computer (laptop, etc.) is connected to the port, and the device is reprogrammed. When no computer is connected, the device cannot be reprogrammed. This scenario allows for changes in the device during development or testing, without having to physically disassemble the instrument and remove the device from the electronics board.

Another scenario could be that a new CPLD/FPGA "program" is sent to a microprocessor in the system, which would then reprogram the CPLD/FPGA. The ability to do this would have to be included in the microprocessor's software, as well as the physical circuitry being present. This scenario would allow the device to be reprogrammed in an environment where physical connection is impossible, such as in orbit around Earth.

When the device can only be reprogrammed by making a physical connection, it is relatively "safe" during operation. A software error in the main computer (processor) code, or a bad command sent by a human operator, is not going to lead to the unexpected reprogramming of the device. The main concern is that reprogramming invalidates all or most of the testing that has gone before. The later the reprogramming is done in the system development cycle, the riskier it is. A set of regression tests should be run whenever the device is reprogrammed, once the instrument is out of the development phase.

If the device can be reprogrammed by an in-system processor, then the possibility exists that it could accidentally be reprogrammed. If the device is involved in a hazard control or can cause a hazardous condition, this could be very dangerous. The legitimate command to reprogram the device would be considered a hazardous command. Commanding in general would have to be looked at closely, and safeguards put in place to make sure the reprogramming is not accidentally commanded. Other checks should be put in place to make sure that software errors do not lead to the unintentional reprogramming of the device.

### 6.12.4 Configurable Computing

Some FPGAs (and CPLDs) use SRAM (Static RAM) technology inside. These SRAM-based devices are inherently re-programmable, even in-system. However, they require some form of external configuration memory source on power-up. The configuration memory holds the program that defines how each of the logic blocks functions, which I/O blocks are inputs and outputs, and how the blocks are interconnected together. The device either self-loads its configuration memory or an external processor downloads the memory into the device. The configuration time is typically less than 200 ms, depending on the device size and configuration method.

The ability to change the internal chip logic “on the fly” can be very useful in some applications, such as pattern matching, encryption, and high-speed computing. Configurable computing’s key feature is the ability to perform computations in hardware to increase performance, while retaining much of the flexibility of a software solution. Applications that benefit the most from configurable computing solutions are those with extremely high I/O data rate requirements, repetitive operations on huge data sets, a large number of computational operations per input data point, or with a need to adapt quickly to a changing environment.

One strength of configurable computing machines (CCMs) is their inherent fault tolerance. By adding structures to detect faults, the hardware can be reconfigured to bypass the faults without having to shut down the instrument.

The downside of flexibility, however, is the difficulty in verifying the functionality and safety of a configurable system. When you can change the hardware in the middle of an operation, how do you assure its safety? That question has not been well addressed yet, as configurable computing is still a new concept. However, if your design uses CCMs, then consider very carefully how to test and verify them, as well as how to guard against internal and external errors or problems.

An article by Villasenor and Mangione-Smith [10] discusses various aspects of configurable computing.

### 6.12.5 Safety and Programmable Logic Devices

IEC 1131-3 is the international standard for programmable logic controller (PLC) programming languages. As such, it specifies the syntax, semantics and display for the following suite of PLC programming languages:

- Ladder diagram (LD)
- Sequential Function Charts (SFC)
- Function Block Diagram (FBD)
- Structured Text (ST)
- Instruction List (IL)

However, IEC 1131-3 does not address safety issues in programming PLCs. The SEMSPLC project was developed to address those issues. They have issued “SEMSPLC Guidelines: safety-related application software for programmable logic controllers” [11], available from the Institution of Electrical Engineers.

Language choice for the PLC should meet with the standard IEC 1131-3. Coding standards should be created. In addition, the language should conform to the following criteria, if possible:

- Closeness to application domain
- Definition/standardization
- Modular
- Readable/understandable
- Traceability
- Checkable
- Analyzable
- Deterministic

Standard software engineering practices should be used in the creation of PLC software. This includes requirements specification, design documentation, implementation, and rigorous testing. For safety-critical systems, Formal Inspections should be used on the products from the various stages, in particular on the requirements.

PLC development can be very formal, up to and including using Formal Methods. However, tailoring of many of the techniques for PLC development has not been done. This an emerging field that requires much more study, to determine the best development practices and the best safety verification techniques.

Besides the SEMSPLC guidelines [11], some general guidelines for better (and safer) PLC programming are:

- **Spaghetti code results in spaghetti logic.** Create a coding style/standard and stick to it. The better your code is, the faster or smaller the resulting logic will be!
- **Keep it under 85.** Don't use more than 85% of the available resources. This makes it easier to place-and-route the design and allows room for future additions or changes.
- **Modularize.** As much as possible, use modules in your PLC software. This helps avoid spaghetti code, and aids in testing and debugging.
- **Use black-and-white testing.** Use both black-box and white-box testing. Verify proper response to inputs (black-box), and also that all paths are executed (white-box), for example.
- **Research safety techniques for PLC application.** Work is being done to see which software engineering techniques are useful for PLC applications. Work by the SEMSPLC project has shown that control-flow analysis produces little information on the errors, but mutation analysis and symbolic execution have revealed a number of complex errors. [12]

## *6.13 Embedded Web Technology*

Everything is connected to everything else…through the Internet, or so it seems. Once the realm of academics sharing research data and ideas, the Internet (and its multimedia child, the World Wide Web) is now the medium for information exchange, conversation, and connectivity.

From the Embedded Web Technology site at the NASA Glenn Research Center (http://vic.lerc.nasa.gov/): "Embedded Web Technology (EWT) is the application of software developed for the World Wide Web to embedded systems. Embedded systems contain computers, software, input sensors and output actuators all of which are dedicated to the control of a specific device. Examples of devices with embedded systems include cars, household appliances, industrial machinery, and NASA Space Experiments.

EWT allows a user with a computer and Web browser to monitor and/or control a remote device with an embedded system over the Internet using a convenient, graphical user interface."

Many embedded devices are now including web servers and network hardware for communications with the outside world, instead of (or in addition to) serial or parallel ports. Some devices (Internet appliances) have no user interface hardware (keyboard, monitor). The user connects through any computer with a web browser to interact with the appliance. Embedded web servers allow remote access to the instrument (hardware) from nearly anywhere in the world.

Instruments can operate as distributed systems, with a central processor and various microcontrollers, communicating back and forth via a network. In the same way that multi-tasking operating systems "break up" the application software into various independent tasks, a distributed system "breaks up" the tasks and runs them on specialized or remote processors. A distributed instrument will have the same problems as described in *Section 6.11 Distributed Computing*.

### 6.13.1 Embedded Web Servers

Most web server software is designed for desktop systems, with a keyboard, monitor, file system, and large hard-disk. For embedded systems, the web server needs to be scaled down, as well as addressing some embedded-specific issues. Reducing the memory footprint, increasing efficiency and reliability, and source portability are important in the embedded world.

The requirements for an embedded web server include [14]:

* **Memory usage.** A small memory footprint (amount of memory used, including code, stack, and heap) is a very important requirement for an embedded web server. Memory fragmentation is also important, as frequently creating and destroying data (such as web pages) may create a myriad of tiny memory blocks that are useless when a larger memory block must be allocated. If the embedded software does not provide memory defragmentation, then embedded web servers should use only statically allocated or pre-allocated memory blocks.

* **Support for dynamic page generation.** Most of the HTML pages produced by embedded systems are generated on the fly. An embedded device will have only a few pages in memory and will often generate part or all of their content in real-time. The

current status of the device, sensor values, or other information may be displayed on the dynamically generated page.

- **Software integration.** Without source code, integrating the web server with the embedded operating system and applications code may be difficult or impossible. When source code is available, ease of integration (and good documentation!) are still important factors to consider.
- **ROMable web pages.** Embedded systems without disk drives often store their data in ROM (or flash memory), sometimes within the executable file, and sometimes external to it. The ability to "pull out" an HTML file or other data from the executable, or find it on the flash disk, is lacking in most desktop-based web servers.
- **Portability.** Nothing stays the same in the embedded world. Technology changes fast, and the processor or operating system used today may be obsolete tomorrow. The ability to port the web server to different processors or operating systems is important for long-term usage.

### 6.13.2 Testing Techniques

Some aspects of standard web-site testing do not apply to embedded web servers. However, consider checking the following areas:

- **Load Handling Capacity.** What is the total data rate the server can provide? How many transactions per second can the server handle? Are these values in line with the expected usage? What happens when the limits are exceeded?
- **User Interface.** Even if the web pages are not meant for world-wide viewing, there is a customer or two who need to view the provided data. Review the generated web pages for clarity of communication (tone, language), accessibility (load time, easy to understand and follow links), consistency ("look and feel", repeating themes), navigation (links obvious in intent and destination, standard way to move between pages), design (page length, hyperlinks), and visual presentation (use of color, easy on the eyes).
- **Data age.** Is there a way to know how fresh the data is? Is the data time-tagged? When the data refreshes, does it show up on the web page?
- **Speed of page generation.** Since most pages are generated "on the fly" in embedded web servers, the speed at which they are constructed is important.
- **Can the user "break" the system?** Check all user-input elements on the web page (buttons, etc.). Try combinations of them, strange sequences, etc. to see if the user can create problems with the web server. If you have a colleague who seems to break his web client on a regular basis, put him to work testing your system.
- **Security testing.** If you will not be on a private network, test the security provision in your web server. Can an unauthorized user get into the system?
- **Link testing.** If you provide links between pages, make sure they are all operational.
- **HTML and XML validation.** Is the HTML and/or XML standard? Will it work with all the browsers expected to interface with the web server? All browser versions?

- **Control of instrumentation.** If the embedded server provides a way to control instrumentation, test this thoroughly. Look for problems that might develop from missed commands, out of sequence commands, or invalid commands.
- **Error handling.** Does the web page handle invalid input in a graceful way? Do any scripts used anticipate and handle errors without crashing? Are "run time" handlers included? Do they work?

Good sources for information on error handling and website testing are:

- "Handling and Avoiding Web Page Errors Part 1: The Basics" (and parts 2 and 3 as well) http://msdn.microsoft.com/workshop/author/script/weberrors.asp
- "WebSite Testing", http://www.soft.com/eValid/Technology/White.Papers/website.testing.html
- "Beyond Broken Links", http://www.dbmsmag.com/9707i03.html
- "WebSite Performance Analysis", http://solo.dc3.com/white/wsperf.html
- "User Testing Techniques – A Reader-Friendliness Checklist" http://www.pantos.org/35317.html

## *6.14 AI and Autonomous Systems*

Artificial Intelligence (AI) and Autonomous Systems reside on the "cutting edge" of software technology. They are two separate entities that, combined, have the potential to create systems that can operate in changing environments without human control. Space exploration, particularly in environments far from Earth, where human intervention would come far too late, is an ideal use for Intelligent Autonomous Systems.

Artificial Intelligence encompasses any system where the software must "think" like a human. This involves information gathering, information pattern recognition, planning, decision making, and execution of the decision. That's a lot for a software system to do! Various aspects of AI includes:

- **Game Playing.** Games such as chess or checkers.
- **Expert Systems**. Systems that capture a large body of information about a domain to answer a question posed to them. Diagnosing a disease based on symptoms is one example of an expert system.
- **Agents.** A computational entity which acts on behalf of other (most often human) entities in an autonomous fashion, performs its actions with proactivity and/or reactiveness and exhibits some level of learning, co-operation and mobility. For example, an agent may perform independent searches for information, on the Internet or other sources, based on subjects needed for an upcoming technical meeting you will be attending.
- **Natural Language**. Understanding and processing natural human languages.
- **Neural Networks.** Connecting the information "nodes" in ways similar to the connections within an animal brain. Neural nets "learn" with repetitive exercising.
- **Robotics.** Controlling machines that can "see" or "hear" (via sensors) and react to their environment and input stimuli. AI robots have a "thinking" capability, unlike factory robotics that perform specific functions as programmed.

Whereas several versions of AI can exist independent of "hardware" (e.g. on a desktop computer), autonomous systems almost always control real-world systems. A robot that operates without human intervention, except for the issuance of orders ("clean the first floor on Tuesday night, the second and third on Wednesday, …") is an example of an autonomous system. One definition for an autonomous system is "a combination of a computational core, sensors and motors, a finite store for energy, and a suited control allowing, roughly speaking, for flexible stand-alone operation.[21]"

This section focuses on Intelligent Autonomous Systems that control hardware systems capable of causing hazards. As a technology on the cutting edge, methods to design, code, test, and verify such systems are not well known or understood. The issue of assuring the safety of such systems is being researched, but the surface has barely been scratch. Hopefully, much more will be learned in the coming years about creating and verifying safety critical Intelligent Autonomous Systems.

In the future, when you travel to Jupiter in cryogenic sleep, with an Intelligent Autonomous System operating the spacecraft and watching your vital signs, you want it to operate *correctly* and *safely*. HAL 9000 needed a bit more software verification!

### 6.14.1 Examples of Intelligent Autonomous Systems (IAS)

Intelligent spacecraft are one promising application of IAS. In the past, the on-board software had some "built-in intelligence" and autonomy in responding to problems, but planning for both the mission and any failures was performed by humans back on Earth. As we send probes out into the far reaches of the solar system, where communications lag time is measured in hours, having a spacecraft that can "think for itself" would be useful. Even for nearby objects, such as Mars, the communications lag time is enough to cause problems in a rover moving at speed over a varied terrain. Removing the "human" from the details of operation can increase the amount of science returned, as intelligent spacecraft and robots no longer have to wait for responses from Earth whenever a problem is encountered. The Deep Space 1 mission focused on technology validation, and contained an experiment in Intelligent Autonomous Systems. Called Remote Agent, it actually controlled the spacecraft for several days, responding to simulated faults. This experiment is described and discussed in Section 6.14.3 Case Study.

Back down to Earth, cleaning office buildings is a monotonous, dirty, dull and "low-esteem" task that does not use the higher faculties of human intelligence. Intelligent mobile cleaning robots are currently under development to automate the process [16], moving humans from "grunts" to supervisors.

"Fly-by-wire" aircraft systems have replaced hydraulic control of the aircraft with computer control (via wire to electromechanical hardware that moves the parts or surfaces). The computer keeps the aircraft stable and provides smoother motions than would be possible with a strictly mechanical system. Computers also provide information to the pilots, in the form of maps, trajectories, and aircraft status, among other items.

At this time, most of the fly-by-wire systems are not *intelligent*. Humans still direct the systems, and usually have overrides if the system misbehaves. However, the trend is to move the human pilot farther from direct control of the aircraft, leaving the details to the computerized system. At some time in the future, fly-by-wire computers could control nearly all aircraft functions, with the pilot providing guidance (where the plan should go) and oversight in the case of a malfunction.

Despite the fact that software used in aircraft is subjected to a stringent development process and thorough testing, an increasing number of accidents have “computer problems” as a contributing factor. In some cases, the computer displayed inaccurate information, which misled the flight crew. In others, the interface between the pilot and software was not well designed, leading to mistakes when under pressure. This points to the increased likelihood of aircraft accidents as computers and software become the “pilots”. Finding reliable methods for creating and verifying safe software must become a priority.

The Intelligent Transportation System (ITS) is being researched and developed under the direction of the US Department of Transportation (http://www.its.dot.gov/). Major elements of ITS include:

- Advanced Traffic Management Systems (ATMS) which monitor traffic flow and provide decision support to reduce congestion on highways.
- Advanced Traveler Information Systems (ATIS) which rovide travelers with directions, route assistance and real-time information on route conditions.
- Automated Highway Systems (AHS) which support and replace human functions in the driving process
- Intelligent Vehicle Initiative (IVI) which focuses efforts on developing vehicles with automated components
- Advanced commercial vehicle Systems (ACS) which provide support for commercial vehicle operations including logistics.

Software safety will obviously be important in developing ITS, though information on how it will be implemented has been difficult to come by. The DOT document on Software Acquisition for the ITS, which consists of over 250 pages, devotes only 4 pages to Software Safety issues.

### 6.14.2 Problems and Concerns

Like distributed systems and other complex software technologies, verifying the safety of Intelligent Autonomous Systems poses a large problem. Remember that for NASA, safety means more than just injury or death. Safety refers to the vehicle and payload as well. So even though no one can be killed by your space probe, loss of the probe would be a safety issue!

The complex interactions that occur between hardware and software must be considered for Intelligent Autonomous Systems, as for any software that controls hardware. In addition, the choices made by the software (plans and decisions based on past performance, current hardware status, and desired goals) form a subset of millions of possible “paths” the system may take. If that subset was know, it could be thoroughly tested, if not formally verified. The number of paths, and the complexities of the interactions between various software modules and the hardware, make complete testing or formal verification essentially impossible.

Various areas of concern with Intelligent Autonomous Systems (IAS) are:

- **Technology is more complicated and less mature.** Intelligent Autonomous Systems are on the cutting edge of software technology.

- **Sensitivity to the environment/context.** Traditional flight software (and other complicated embedded software) was designed to be *independent* of the system environment or software context. When a command was received, it was executed, regardless of what the spacecraft was doing at the time (but within the safety/fault tolerance checks). Whether or not the command made sense was the responsibilities of the humans who sent it. An IAS, on the other hand, must know what the environment and system context are when it generates a command. It must create a command appropriate to the system state, external environment, and software context.
- **Increased Subsystem interactions.** Traditional software systems strive for minimal interactions between subsystems. That allows each subsystem to be tested independently, with only a minimal "integrated" system testing. IAS subsystems, however, interact in multiple and complicated ways. This increases the number of system tests that must be performed to verify the system.
- **Complexity.** Intelligent Autonomous Systems are complex software. Increased complexity means increased errors at all levels of development – specification, design, coding, and testing.

New software technology often stresses the ability of traditional verification and validation techniques to adequately authenticate the system. You can't formally specify and verify the system without tremendous effort, you cannot test every interaction, and there is no way to know for certain that every possible failure has been identified and tested!

The state-of-the-art for Intelligent Autonomous System (IAS) verification has focused on two areas: Testing (primarily) and Formal Verification (Model Checking). Simmons, et. al. [20] discuss using model checking for one subset of IAS: application-specific programs written in a specialized, highly-abstracted language, such as used by Remote Agent. The application programs are verified for *internal* correctness only, which includes checks for liveness, safety, etc.

Testing issues with Remote Agent are discussed in 6.14.3.2. An additional testing strategy is described by Reinholtz and Patel [19]. They propose a four-pronged strategy, starting with *formal specifications* of correct system behavior. The software is tested against this specification, to verify correct operations. Transition zones (areas of change and interaction among the subsystems) are *identified* and *explored* to locate incorrect behavior. The fourth element of the strategy is to *manage risk* over the whole lifecycle.

### 6.14.3 Case Study – Remote Agent on Deep Space 1

Remote Agent is an experiment in Intelligent Autonomous Systems. It was part of the NASA Deep Space 1 (DS-1) mission. The experiment was designed to answer the question "Can a spacecraft function on its own nearly 120 million kilometers from Earth, without detailed instructions from the ground?"

Remote Agent was originally planned to have control of DS-1 for 6 hours (a confidence building experiment) and for 6 days. Due to various problems, the experiment was replanned for a 2 day period in May, 1999. During the experiment run, Remote Agent controlled the spacecraft and responded to various simulated problems, such as a malfunctioning spacecraft thruster. Remote Agent functioned very well, though not flawlessly, during it's two day experiment.

#### 6.14.3.1 Remote Agent Description

"Remote Agent (RA) is a model-based, reusable, artificial intelligence (AI) software system that enables goal-based spacecraft commanding and robust fault recovery[48]." To break that statement down into its component parts:

- **Model based.** A model is a general description of the behavior and structure of the component being controlled, such as a spacecraft, robot, or automobile. Each element of Remote Agent (RA) solves a problem by accepting goals, then using reasoning algorithms on the model to assemble a solution that meets the goals.
- **Reusable.** Parts of the Remote Agent were designed to be system independent and can be used in other systems without modification. Other aspects are system dependent, and would need modification before being used in a different system.
- **Artificial Intelligence.** Remote Agent *thinks* about the goals and how to reach them.
- **Goal-based commanding.** Instead of sending Remote Agent a sequence of commands (slew to this orientation, turn on camera at this time, begin taking pictures at this time, etc.), RA accepts *goals* such as "For the next week, take pictures of the following asteroids, keeping the amount of fuel used under X." Goals may not be completely achievable (parts may conflict) and Remote Agent has to sort that out.
- **Robust fault recovery.** Remote Agent can plan around failures. For example, if one thruster has failed, it can compensate with other thrusters to achieve the same maneuver.

The Remote Agent software system consists of 3 components: the Planner/Scheduler, the Executive, and the MIR (Mode Identification and Reconfiguration, also called Livingstone).

The Planner/Scheduler (PS) generates the plans that Remote Agent uses to control the spacecraft. It uses the initial spacecraft state and a set of goals to create a set of high-level tasks to achieve those goals. PS uses its model of the spacecraft, including constraints on operations or sequence of operations, to generate the plan.

The Executive requests plans from PS and executes them. It also requests/executes failure recoveries from MIR, executes goals and commands from human operators, manages system resources, configures system devices, provides system-level fault protection, and moves into safe-modes as necessary. It's a busy little program!

The Mode Identification and Reconfiguration (MIR) element diagnoses problems and provides a recovery mechanism. MIR needs to know what is happening to all components of the spacecraft, so it eavesdrops on commands sent to the hardware by the Executive. Using the commands and sensor information, MIR determines the current state of the system, which is reported to the Executive. If failures occur, MIR provides a repair or workaround scenario that would allow the plan to continue execution.

#### 6.14.3.2 Testing and Verification of Remote Agent

The main problem in testing Remote Agent was that the number of possible execution paths through the software was on the order of millions. Unlike traditional spacecraft flight software, where a *sequence* of operations was uplinked after ground verification, Remote Agent had to think for itself, identifying problems and taking corrective action, in order to achieve the goals.

It is impossible to test all these execution paths within the software, at least within the lifetime of the tester, if not the universe!

For the Remote Agent experiment, a scenario-based verification strategy was augmented with model-based verification and validation [Smith et. al. 17]. The universe of possible inputs (goals, spacecraft state, device responses, timing, etc.) is partitioned into a manageable number of scenarios. Remote Agent is exercised on each scenario and its behavior is verified against the specifications.

Going from millions or billions of possible tests down to a manageable number (200 to 300) entails adding risk. If you test everything, you know how the system will respond in any possible scenario. When you test a small subset, there is the risk that you missed something important – some scenario where the interactions among the subsystem are not what you expected. You must be able to have confidence that the tested scenarios imply success among the untested scenarios.

The effectiveness of scenario-based testing depends largely on how well the scenarios cover the requirements. This means that not only is the requirement tested, but that the selected inputs for the tests give confidence that the requirement works for all other inputs. The Remote Agent experiment used a parameter-based approach to select the scenarios and inputs to use.

Three methods were used to achieve good coverage while maintaining manageability:

- Abstracting parameter space to focus on relevant parameters and values. Parameters and parameter values were selected to focus on areas where the software was most sensitive. Equivalence classes were used to generalize from these inputs to a collection of comparable tests that would not be performed.
- Identifying independent regions of the parameter space. Areas where there is low or no interactions mean that fewer combinations of parameters/values must be tested. When there is strong interaction among parameters, more combinations must be tested.
- Using orthogonal arrays to generate minimal-sized test suites that cover the parameter combinations. Every parameter value and every pair of values appears in at least one test case. Every parameter value appears in about the same number of cases.

One difficulty encountered during testing was that it was difficult to know what parameter value lead to failure. To overcome this, a collection of test cases identical to the faulty one was generated, with each test identical except for one parameter. This allowed the value leading to the error to be identified.

Another difficulty in any form of spacecraft software testing is the lack of high-fidelity test beds. The closer the test bed is to flight fidelity, the less time you will get on it. To deal with this issue, the Remote Agent experiment performed tests on highly abstract software (software that did not care about the underlying hardware) on low-fidelity test beds, and reserved the high-fidelity test beds for hardware-specific software and general system verification.

Remote Agent automated the testing process, and in some cases "warped" the time, so that a multi-day test could be completed in hours. Software was also written to help with understanding the test results. These not only allowed more tests to be done in the limited period of time, but increased the chance of spotting errors. Reviewing log files is very tedious, and errors can be missed.

**Remember: Software which tests safety critical code must also be considered safety critical.**

Additional information on the verification and testing of the Remote Agent experiment can be found in Smith et. al. [17]. Bernard et. al. [18] discusses the testing strategy, as well as the in-flight validation of Remote Agent.

#### 6.14.3.3 In-flight Validation: How well did it work?

Even before flight, problems with some aspects of the testing strategy were noted. During the last four months before flight, after the formal testing phase had ended, a large number of new problems were discovered. Most of the problems related to the planning system operating correctly, but unable to find a plan within the time constraints. Several reasons were identified:

- o Range of some parameters differed from those assumed for testing.
- o Disappearance of slack time in going from the 6 day to 2 day scenario revealed brittleness in the Planner chronological backtracking search.
- o The test generator only considered pair-wise interactions. Some problems depended on the specific values of 3 or more parameters.

During the flight experiment, a problem developed with Remote Agent not terminating the Ion Propulsion System (IPS) thrusting as expected. Plan execution appeared to be blocked, but the Remote Agent and the spacecraft were both healthy. The cause was later identified as a missing critical section in the plan-execution code. This created a race condition between two Executive threads. If the wrong thread won, a deadlock condition would occur where each thread was waiting for an event from the other. This occurred in flight, though not on the ground, despite thousands of previous races during the ground testing.

The following is drawn from the Remote Agent Lessons Learned:

- ❖ Basic system must be thoroughly validated with a comprehensive test plan as well as formal methods, where appropriate.
- ❖ Automatic code generation of interface code, telemetry, model interfaces, and test cases was enormously helpful.
- ❖ Better model validation tools are needed. Automated test running capability helped increase the number of off-nominal tests that could be run. However, manual evaluation of the test results was laborious.
- ❖ Confidence in complex autonomous behaviors can be built up from confidence in each individual component behavior.
- ❖ Ground tools need to be created early and used to test and understand how to operate the complex flight system. For Remote Agent, the ground tools were developed very late and many of them were not well integrated.
- ❖ Ensuring sufficient visibility into the executing software requires adequate information in the telemetry. Design the telemetry early and use it as the primary way of debugging and understanding the behavior of the system during integration, test, and operations.

As the problems found in late ground operations and flight operations show, the testing strategy was not 100% successful. In particular, a timing problem that rarely occurred was missed because it never happened on the ground.

More work needs to be done on the verification and validation of Intelligent Autonomous Systems, especially if they are to have control over safety critical functions and equipment. Remote Agent had "backup" from the flight software and hardware hazard controls. It was a successful experiment that shows promise for the future. But it is not quite ready for complete control of safety-critical systems.

## *6.15 Good Programming Practices for Safety*

Besides all the practices discussed in the programming languages sections, there are some simple ways to make your software safer. The lists below come from various sources, which are referenced. In addition, they are summarized in a checklist in Appendix E.

The following list is from "Solving the Software Safety Paradox" by Doug Brown [1]. See that article for more details.

- **CPU self test.** If the CPU becomes partially crippled, it is important for the software to know this. Cosmic Radiation, EMI, electrical discharge, shock, or other effects could have damaged the CPU. A CPU self-test, usually run at boot time, can verify correct operations of the processor. If the test fails, then the CPU is faulty, and the software can go to a safe state.
- **Guarding against illegal jumps**. Filling ROM or RAM with a known pattern, particularly a halt or illegal instruction, can prevent the program from operating after it jumps accidentally to unknown memory. On processors that provide traps for illegal instructions (or a similar exception mechanism), the trap vector could point to a process to put the system into a safe state.
- **ROM tests**. Prior to executing the software stored in ROM (EEPROM, Flash disk, etc.), it is important to verify its integrity. This is usually done at power-up, after the CPU self test, and before the software is loaded. However, if the system has the ability to alter its own programming (EEPROMS or flash memory), then the tests should be run periodically.
- **Watchdog Timers.** Usually implemented in hardware, a watchdog timer resets (reboots) the CPU if it is not "tickled" within a set period of time. Usually, in a process implemented as an infinite loop, the watchdog is written to once per loop. In multitasking operating systems, using a watchdog is more difficult. Do NOT use an interrupt to tickle the watchdog. This defeats the purpose of having one, since the interrupt could still be working while all the real processes are blocked!
- **Guard against Variable Corruption.** Storing multiple copies of critical variables, especially on different storage media or physically separate memory, is a simple method for verifying the variables. A comparison is done when the variable is used, using two-out-of-three voting if they do not agree, or using a default value if no two agree. Also, critical variables can be grouped, and a CRC used to verify they are not corrupted.
- **Stack Checks.** Checking the stack guards against stack overflow or corruption. By initializing the stack to a known pattern, a stack monitor function can be used to watch the amount of available stack space. When the stack margin shrinks to some

predetermined limit, an error processing routine can be called, that fixes the problem or puts the system into a safe state.

- **Program Calculation Checks.** Simple checks can be used to give confidence in the results from calculations.

"30 Pitfalls for Real-Time Software Developers", by David B. Stewart [4][5] discusses problems faced by real-time developers. Of the problems he considers, the following are especially applicable to safety and reliability:

- **Delays implemented as empty loops.** This can create problems (and timing difficulties) if the code is run on faster or slower machines, or even if recompiled with a newer, optimizing compiler.
- **Interactive and incomplete test programs.** Tests should be planned an scripted. This prevents tests from being missed. Also, functional tests should be run after a change, to make sure that the software change did not indirectly impact other code.
- **Reusing code not designed for reuse.**~~.~~ If the code was not designed for reuse, it may have interdependencies with other modules. Usually, it will not use abstract data types (if object-oriented) or have a well-defined interface. ~~"Code that is not designed for reuse will not be in the form of an abstract data type~~
- ~~or object. The code may have interdependencies with other code, such that if all of it is taken, there is more code than needed. If only part is taken, it must be thoroughly dissected, which increases the risk of unknowingly cutting out something that is needed, or unexpectedly changing the functionality."~~
- **One big loop.** ~~"When real-time software is designed as a single big loop, we have no flexibility to modify the execution time of various parts of the code independently. Few real-time systems need to operate everything at the same rate."~~A single large loop forces all parts of the software to operate at the same rate. This is usually not desirable.
- **No analysis of hardware peculiarities before starting software design.** Different processors have peculiarities that can affect the time a calculation can take, or how long it takes to access an area of memory, for instance. Understanding the hardware before designing the software will decrease the number of "gotchas" at integration time.
- **Fine-grain optimizing during first implementation. "**Some programmers foresee anomalies (some are real, some are mythical). An example of a mythical anomaly is that multiplication takes much longer than addition."
- **Too many inter-module dependencies**. To maximize software reusability, modules should not depend on each other in a complex way.
- **Only a single design diagram. "**Most software systems are designed such that the entire system is defined by a single diagram (or, even worse, none!). When designing software, getting the entire design on paper is essential."
- **Error detection and handling are an afterthought and implemented through trial and error.** Design in the error detection and handling mechanisms *from the start*. Tailor the effort to the level of the code – don't put it everywhere! Look at critical locations

where data needs to be right or areas where the software or hardware are especially vulnerable to bad input or output.

- **No memory analysis.** Check how much memory your system uses. Estimate it from your design, so that you can adjust the design if the system is bumping up against its limits. When trying to decide between two different implementations of the same concept, knowing the memory usage of each will help in making a decision.

- **Documentation was written after implementation.**~~"Everyone knows that the system documentation for most applications is dismal. Many organizations make an effort to make sure that everything is documented, but documentation isn't always done at the right time. The problem is that documentation is often done after the code is written."~~Write what you need, and use what you write. Don't make unnecessarily verbose or lengthy documentation, unless contractually required. It is better to have short documents that the developers will actually read and use.

- **Indiscriminate use of interrupts. Use of interrupts can cause priority inversion in real-time systems if not implemented~~carefully.~~** ~~"Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements. The reason for this delay is that interrupts preempt everything else and aren't scheduled."This~~**carefully.** This can lead to timing problems and the failure to meet necessary deadlines.

- **No measurements of execution time.** "Many programmers who design real-time systems have no idea of the execution time of any part of their code."

Bill Wood, in "Software Risk Management for Medical Devices"[15], Table III, gives a list of mitigation mechanisms for various possible failures. Some of the practices that are not duplicated in the lists above are summarized below (and expanded upon):

- **Check variables for reasonableness before use.** If the value is out of range, there is a problem – memory corruption, incorrect calculation, hardware problems (if sensor), etc.

- **Use execution logging, with independent checking, to find software runaway, illegal functions, or out-of-sequence execution.** If the software must follow a known path through the modules, a check log will uncover problems shortly after they occur.

- **Come-from checks.** For safety critical modules, make sure that the correct previous module called it, and that it was not called accidentally by a malfunctioning module.

- **Test for memory leakage.** Instrument the code and run it under load and stress tests. See just how the memory usage changes, and check it against the predicted usage.

- **Use readbacks to check values.** When a value is written to memory, the display, or hardware, another function should read it back and verify that the correct value was written.

In addition to the suggestions above, consider doing the following:

- **Use a simulator or ICE (In-circuit Emulator)** system for debugging in embedded systems. These tools allow the programmer/tester to find some subtle problems more easily. Combined with some of the techniques described above, they can find memory access problems and trace back to the statement that generated the error.

- **Reduce complexity.** Calculate a complexity metric. Look at modules that are very complex and reduce them if possible. Complexity metrics can be very simple. One way to calculate McCabe's Cyclomatic Complexity is to add the number of decisions and subtract one. An "if" is a 1. A case/switch statement with 3 cases is 2. Add these up, subtract one. If the complexity is over 10, look at simplifying the routine.
- **Design for weak coupling** between modules (classes, etc.). The more independent the modules are, the less you can screw them up later in the process. "Fixes" when an error is found in testing may create problems because of misunderstood dependencies between modules.
- **Consider the stability of the requirements.** If the requirements are likely to change, design as much flexibility as possible into the system.
- **Consider compiler optimization carefully.** Debuggers may not work well with optimized code. It's hard to trace from the source code to the optimized object code. Optimization may change the way the programmer expected the code to operate (removing "unused" features that are actually used!).
- **Be careful if using multi-threaded programs.** Developing multi-threaded programs is notoriously difficult. Subtle program errors can result from unforeseen interactions among multiple threads. In addition, these errors can be very hard to reproduce since they often depend on the non-deterministic behavior of the scheduler and the environment.
- **A dependency graph** is a valuable software engineering aid. Given such a diagram, it's easy to identify what parts of the software can be reused, create a strategy for incremental testing of modules, and develop a method to limit error propagation through the entire system.
- **Follow the two person rule**. At least two people should be thoroughly familiar with the design, code, testing and operation of each software module of the system. If one person leaves the project, someone else understands what is going on.
- **Prohibit program patches.** During development, patching a program is a bad idea. Make the changes in the code and recompile instead. During operations, patching may be a necessity, but should still be carefully considered.
- **Keep Interface Control Documents up to date.** Out-of-date information usually leads to one programmer creating a module or unit that will not interface correctly with another unit. The problem isn't found until late in the testing phase, when it is expensive to fix. Besides keeping the documentation up to date, use an agreed-upon method to inform everyone of the change.
- **Create a list of possible hardware failures that may impact the software**, if they are not spelled out in the software requirements document. Have the hardware and systems engineers review the list. The software must respond properly to these failures. The list will be invaluable when testing the error handling capabilities of the software. Having a

list also makes explicit what the software can and cannot handle, and unvoiced assumptions will usually be discovered as the list is reviewed.

The following programming suggestions are derived from SSP 50038, Computer-Based Control System Safety Requirements for the International Space Station Program:

- Provide **separate authorization and separate control functions** to initiate a critical or hazardous function. This includes separate "arm" and "fire" commands for critical capabilities.
- **Do not use input/output ports for both critical and non-critical functions**.
- Provide **sufficient difference in addresses** between critical I/O ports and non-critical I/O ports, such that a single address bit failure does not allow access to critical functions or ports.
- Make sure all **interrupt priorities** and responses are defined. All interrupts should be initialized to a return, if not used by the software.
- Provide for an **orderly shutdown** (or other acceptable response) upon the detection of **unsafe conditions**. The system can revert to a known, predictable, and safe condition upon detection of an anomaly.
- Provide for an **orderly system shutdown** as the result of a **command shutdown**, power interruptions, or other failures. Depending on the hazard, battery (or capacitor) backup may be required to implement the shutdown when there is a power failure.
- Protect against **out-of-sequence transmission** of safety-critical function messages by detecting and deviation from the normal sequence of transmission. Revert to a known safe state when out-of-sequence messages are detected.
- Initialize all **unused memory** locations to a pattern that, if executed as an instruction, will cause the system to revert to a known safe state.
- **Hazardous sequences should not be initiated by a single keyboard entry**.
- Prevent **inadvertent entry into a critical routine**. Detect such entry if it occurs, and revert to a known safe state.
- **Don't use a stop or halt instruction**. The CPU should be always executing, whether idling or actively processing.
- **When possible, put safety-critical operational software instructions in nonvolatile read-only memory.**
- **Don't use scratch files** for storing or transferring safety-critical information between computers or tasks within a computer.
- When **safety interlocks** are removed/bypassed for a test, the software should verify the reinstatement of the interlocks at the completion of the testing.
- **Critical data** communicated from one CPU to another should be verified prior to operational use.

- Set a **dedicated status flag** that is updated between each step of a hazardous operation. This provides positive feedback of the step within the operation, and confirmation that the previous steps have been correctly executed.
- **Verify critical commands** prior to transmission, and upon reception. It never hurts to check twice!
- **Make sure all flags used are unique and single purpose.**
- Put the majority of **safety-critical decisions and algorithms** in a single (or few) software development module(s).
- **Decision logic** using data from hardware or other software modules should not be based on values of all ones or all zeros. Use specific binary patterns to reduce the likelihood of malfunctioning hardware/software satisfying the decision logic.
- **Safety-critical modules should have only one entry and one exit point.**
- **Perform reasonableness checks on all safety-critical inputs**.
- Perform a **status check** of critical system elements prior to executing a potentially hazardous sequence.
- Always **initialize the software into a known safe state.** This implies making sure all variables are set to an initial value, and not the previous value prior to reset.
- **Don't allow the operator to change safety-critical time limits in decision logic**.
- When the system is safed, usually in response to an anomalous condition or problem, provide the **current system configuration** to the operator.
- Safety-critical routines should include **"come from" checks** to verify that they are being called from a valid program, task, or routine.

## *6.16 Wrapping it all up*

The job of creating safe software is not an easy one. It is a balance of risks and benefits. Sometimes the benefit is a personal choice ("I like the way this editor works"), meeting a contractual requirement ("The software shall be written in Ada"), producing software that meets size or timing restrictions, or creating the software within a given schedule. Each choice carries an associated level of risk of creating "unsafe" software.

In an ideal world, all tools, programming languages, operating systems, etc. that meet your needs would also be designed for "safe" software. Such a world does not exist. Often you are lucky to find items that meet your needs without extensive modification! However, always keep the goal of producing safe software in your mind.

**Whenever possible, *select for safety.* Otherwise, take steps to mitigate the risks to safety.**

## 7. SOFTWARE ACQUISITION

Acquiring software, whether off-the-shelf, previously created, or custom made, carries with it a set of risks and rewards that differ from those related to software development. When the software will serve a safety critical function, or be integrated with in-house developed safety critical code, it becomes very important to select carefully. This section provides guidance on both purchased off-the-shelf and reused software as well as software acquired from a contractor.

Software safety is a concern with off-the-shelf (OTS), reused, and contract-developed software, and NASA safety standards apply to all types. NASA-STD-8719.13A, the Software Safety NASA Technical Standard, section 1.3, states (emphasis added):

> "This standard is appropriate for application to **software acquired or developed by NASA** that is used as a part of a system that possesses the potential of directly or indirectly causing harm to humans or damage to property external to the system. When software is acquired by NASA, this standard applies to the level specified in contract clauses or memoranda of understanding. When software is developed by NASA, this standard applies to the level specified in the program plan, software management plan, or other controlling document."

**Definitions**

| | |
|---|---|
| **Off-the-shelf (OTS)** | Software not developed in-house or by a contractor for the project. The software is general purpose, or developed for a different purpose from the current project. |
| **COTS** | Commercial-off-the-shelf software. Operating systems, libraries, applications, and other software purchased from a commercial vendor. Not customized for your project. Source code and documentation are often limited. |
| **GOTS** | Government-off-the-shelf software. This was developed in-house, but for a different project. Source code is usually available. Documentation varies. Analyses and test results, including hazard analyses, may be available. |
| **Reused software** | Software developed by the current team (or GOTS) for a different project, portions of which are reused in the current software. While it is tempting to pull out a previously written function for the new project, be aware of how it will operate in the new system. Just because it worked fine in System A does not mean it will work OK in System B. A suitability analysis should be performed. |
| **Contracted software** | Software created for a project by a contractor or sub-contractor. The project defines the requirements the software must meet. Process requirements and safety analyses may be included. This is custom-made software, but not in-house. |
| **Glueware** | Software created to connect the OTS/reused software with the rest of the system. It may take the form of "adapters" that modify interfaces or add missing functionality, "firewalls" that isolate the OTS software, or "wrappers" that check inputs and outputs to the OTS software and may modify either to prevent failures. |

NASA Policy Directive NPD 2820.1, NASA Software Policies, includes consideration of COTS and GOTS software that is part of a NASA system. Projects need to evaluate whether the use of COTS and GOTS would be more advantageous than developing the software. It expects proof that software providers are capable of delivering products that meet the requirements.

Off-the-shelf (OTS) software and reused software share many of the same benefits and concerns. They will be grouped together for convenience in section 7.1. "OTS" or "off-the-shelf software" will refer to both off-the-shelf (usually commercial) software and reused software. When a comment refers only to one or the other, the appropriate form of the software will be clearly designated. Software developed under contract will be discussed in section 7.2.

For off-the-shelf software, this section discusses the following areas:

- Pros and Cons of OTS software
- What to look for when purchasing off-the-shelf software
- Using OTS in your system
- Recommended extra testing for OTS software

For contract-developed software, guidance is provided on

- What to put in the contract regarding software development.
- What to monitor of contractor processes (insight/oversight)
- What testing is recommended

## *7.1 Off-the-Shelf Software*

The decision to use off-the-shelf (OTS) software in your system should not be made lightly. While it is becoming common to purchase software rather than create it, the process is not without pitfalls. Reusing software from other projects, even similar projects, is not a panacea either. Systems differ, and the subtle differences can lead to devastating results.

Why is OTS software use becoming more commonplace in NASA and industry? Primarily, the prevailing "wisdom" is that it will save on cost and/or schedule. If a commercial software product can be purchased that meets the needs of the project, it is usually a less expensive alternative to developing the software in-house. Or the organization may have software from a similar project that can be reused in the new project. In addition, the OTS software is often available immediately, which helps in a tight schedule. In a project strapped for money or time, OTS software looks very attractive. However, there are risks involved in using OTS software. Some of the issues are discussed below, and reference [4] provides a method for determining the risks, as well as the cost/benefit ratio, for using COTS software in your system. Reference [2] discusses concerns as well as ways to make sure the OTS software meets your needs and is safe.

Some OTS software is so common that most developers do not even consider it. Operating Systems (OS) are one example. It is very rare for a development team to create their own operating system, rather than purchasing a commercial one. Guidance on what to look for in an operating system is given in section 6.10 Operating Systems. Another example of common OTS software is language libraries, such as the C standard library.

OTS software has acquired another name recently: SOUP (Software of Uncertain Pedigree). In many ways, SOUP is a better name, because it emphasizes the potential problems and pitfalls upfront. OTS software may be developed by a team that uses good software engineering practices, or by a few developers working late nights, living on pizza and soft drinks, and banging out some code. Knowing the pedigree of the OTS software can save you headaches down the road.

Carefully consider all aspects of the OTS software under consideration. It not an easy decision, choosing between creating the software in-house (with its accompanying headaches) or purchasing OTS software/reusing software (which have a different set of headaches). You must take a systems approach and consider how the OTS software will fit into your system. You must also perform an adequate analysis of the impacts of the OTS software. Don't wait until you are deep into implementation to find out that one of the extra functions in the OTS software can cause a safety hazard!

Consider the following questions:

- Will you need glueware to connect the software to your system?
- How extensive will the glueware need to be?
- Will you have to add functionality via glueware because the OTS software doesn't provide all of it?
- Is there extra functionality in the OTS software that the rest of the system needs to be protected from?
- What extra analyses will you need to perform to verify the OTS software?
- What extra tests will you need to do?

If the glueware is going to be a significant portion of the size of the OTS software, you may want to rethink the decision to use OTS. You don't save time or money if you have to created extensive wrappers, glueware, or other code to get the OTS software working in your system. Also, in a safety critical system, the cost of extra analyses and tests may make the OTS software a costly endeavor.

In safety-critical systems, OTS software can be a burden as well as a blessing. The main problems with off-the-shelf software are:

- Inadequate documentation. Often only a user manual is provided, which describes functionality from a user point of view. In order to integrate the software into the system, more information is needed. In particular, information is required on how the software interacts within itself (between modules) and with the outside world (its application program interface (API)).
- Lack of access to source code, which precludes some safety analyses. It also precludes obtaining a better understanding of how the software actually works.
- Lack of knowledge about the software development process used to create the software.
- Lack of knowledge about the testing process used to verify the software.

™ Concern that the OTS developers may not fully understand the interactions between elements of their system or may not communicate that information fully to the purchaser.

™ Inadequate detail on defects, including known bugs not provided to purchaser.

™ Inadequate or non-existent analyses performed on the software.

™ Missing functionality. The OTS software may provide most but not all required functionality. This is one area where glueware is necessary.

™ Extra functionality. The OTS software may contain functions that are not required. Sometimes these functions can be "turn off", but unless the OTS software is recompiled, the code for these functions will remain in the system. Glueware (wrappers) may be needed to shield the rest of the system from this extra functionality.

For further information on how OTS software is handled in other industries, check references [7] (FDA) and [8] (nuclear). They provide some high-level guidance on using COTS software in medical and nuclear applications, both of which are highly safety-critical venues.

The Avionics Division of the Engineering Directorate at the NASA Lyndon B. Johnson Space Center (JSC) baselined a work instruction, EA-WI-018, "Use of Off-the-Shelf Software in Flight Projects Work Instruction" that outlines a lifecycle process for OTS software projects, including safety considerations. This work instruction is based partly on the FDA's process detailed in "Guidance for Off-the-Shelf Software Use in Medical Devices." [??]

The lifecycle described in the JSC work instruction coordinates the selection and integration of OTS software with the development and implementation of custom software. In comparing the lifecycle processes it is evident that the amount of time spent on each phase changes and the skills of the personnel need to be different. For selecting OTS products, a great deal of time is spent evaluating the functional needs of the project and the available OTS products on the market. Flexibility of requirements is needed with a clear idea of the overall system. A poor OTS selection can severely complicate or cripple a project. A series of questions are included in both the JSC work instruction and the FDA guidance document to give personnel enough information to determine whether or not to use a specific OTS software product.

The work instruction specifies that an initial determination of the criticality of the function must be accomplished. The amount of scrutiny the candidate OTS software faces is based on the criticality assessed. Experienced personnel need to determine the criticality. The JSC work instruction and the FDA guidance document list similar requirements for high criticality OTS software. A project with life threatening hazards must do the first three items of the *Checklist for Off-the-Shelf (OTS) Items* (second checklist) in Appendix E.

Some of this section, and those that follow, on Off-the-Shelf software issues, especially within NASA, comes from a whitepaper by Frances E. Simmons of JSC [11].

### 7.1.1 Purchasing or Reusing OTS Software: Recommendations

While all OTS software should be considered carefully, using OTS software in a safety critical system "ups the ante". OTS software that directly performs a safety critical function is not the only element that must be considered. **Any** OTS software that resides on the same platform as the safety critical software must be analyzed, to verify that it cannot impact the safety critical code. Since there is no independent COTS certification authority to test the safety and reliability of the COTS software, all additional analyses and tests will have to be done by you.

Using non-safety-critical OTS software on the same platform as safety-critical software is not recommended. Certain commercial programs are known to crash regularly. Do you really want to have Word on the same system that controls your air supply? If the OTS software provides a necessary function, then it must be considered in conjunction with the safety critical code. The hazard analysis should be updated (or at least reviewed) *before* you purchase the software.

This guidebook gives an introduction to the good software development processes that go into safety critical software development. As much as possible, verify that the OTS software was created using good development processes. When purchasing OTS software, or deciding to reuse existing code, the following areas should also be considered:

- ✓ Does the OTS software fill the need *in this system*? Is its operational context compatible with the system under development? Consider not only the similarities between the system(s) the OTS was designed for and the current system, but also the differences. Look carefully at how those differences affect operation of the OTS software.
- ✓ How stable is the OTS product? Are bug-fixes or upgrades released so often that the product is in a constant state of flux?
- ✓ How responsive is the vendor to bug-fixes? Does the vendor inform you when a bug-fix patch or new version is available?
- ✓ How compatible are upgrades to the software? Has the API changed significantly between upgrades in the past? Will your interface to the OTS software still work, even after an upgrade? Will you have to update your glueware with each iteration?
- ✓ How “cutting edge” is the software technology? OTS software is often market driven, and may be released with bugs (known and unknown) in order to meet an imposed deadline or to beat the competition to market.
- ✓ Conversely, is the software so well known that it is assumed to be error free and correct? Think about operating systems and language libraries. In a safety critical system, you do not want to *assume* there are no errors in the software.
- ✓ What is the user base of the software? If it is a general use library, with thousands of users, you can expect that most bugs and errors will be found and reported to the vendor. Make sure the vendor keeps this information, and provides it to the users! Small software programs will have less of a “shake down” and *may* have more errors remaining.
- ✓ What level of documentation is provided with the software? Is there more information than just a user’s manual? Can more information be obtained from the vendor (free or for a reasonable price)?
- ✓ Is source code included, or available for purchase at a reasonable price? Will support still be provided if the source code is purchased or if the software is slightly modified?
- ✓ Can you communicate with those who developed the software, if serious questions arise? Is the technical support available, adequate, and reachable? Will the vendor talk with you if you modify the product?

- ✓ Will the vendor support older versions of the software, if you choose not to upgrade? Many vendors will only support the newest version, or perhaps one or two previous versions.
- ✓ Is there a well-defined API (Application Program Interface), ICD (interface control document), or similar documentation that details how the user interacts with the software? Are there "undocumented" API functions?
- ✓ What are the error codes returned by the software? How can it fail (return error code, throw an exception, etc.)? Do the functions check input variables for proper range, or it is the responsibility of the user to implement?
- ✓ Can you obtain information on the internals of the software, such as the complexity of the various software modules or the interfaces between the modules? This information may be needed, depending on what analyses need to be performed on the OTS software.
- ✓ Can you get information about the software development process used to create the software? Was it developed using an accepted standard (IEEE 12207, for example)? What was the size of the developer team?
- ✓ What types of testing was the software subjected to? How thorough was the testing? Can you get copies of any test reports?
- ✓ Are there any known defects in the software? Are there any unresolved problems with the software, especially if the problems were in systems similar to yours? Look at product support groups, newsgroups, and web sites for problems unreported by the vendor. However, also keep in mind the source of the information found on the web – some is excellent and documented, other information is spurious and incorrect.
- ✓ Were there any analyses performed on the software, in particular any of the analyses described in section 5? Formal inspections or reviews of the code?
- ✓ How compatible is the software with your system (other software, both custom and OTS)? Will you have to write extensive glueware to interface it with your code? Are there any issues with integrating the software, such as linker incompatibility, protocol inconsistencies, or timing issues?
- ✓ Does the software provide all the functionality required? How easy is it to add any new functionality to the system, when the OTS software is integrated? Will the OTS software provide enough functionality to make it cost-effective?
- ✓ Does the OTS-to-system interface require any modification? For example, does the OTS produce output in the protocol used by the system, or will glueware need to be written to convert from the OTS to the system protocol?
- ✓ Does the software provide extra functionality? Can you "turn off" any of the functionality? If you have the source code, can you recompile with defined switches or stubs to remove the extra functionality? How much code space (disk, memory, etc.) does the extra software take up? What happens to the system if an unneeded function is accidentally invoked?

- ✓ Will the OTS software be stand-alone or integrated into your system? The level of understanding required varies with the two approaches. If stand-alone (such as an Operating System), you need to be concerned with the API/ICD primarily, and interactions with your independent software is usually minimal. If the software is to be integrated (e.g. a library), then the interaction between your code and the OTS software is more complicated. More testing and/or analyses may be needed to assure the software system.
- ✓ Does the OTS software have any "back doors" that can be exploited by others and create a security problem?
- ✓ Is the software version 1.0? If so, there is a higher risk of errors and problems. Consider waiting for at least the first bug-fix update, if not choosing another product.
- ✓ If the OTS product's interface is supposed to conform to an industry standard, verify that it does so.

Appendix E provides the above information as a checklist, and also contains another checklist of items to consider when using OTS software in your system.

IEEE 1228, the standard for Software Safety Plans, states that previously developed (reused) software and purchased software must be

- Adequately tested.
- Have an acceptable risk.
- Remains safe in the context of its planned use.

Any software that does not meet these criteria, or for which the level of risk or consequences of failure cannot be determined, should not be used in a safety critical system. In addition, IEEE 1228 provides a standard for a minimal approval process for reused or purchased software:

- ▪ Determine the interfaces to and functionality of the previously developed or purchased software that will be used in safety-critical systems.
- ▪ Identify relevant documents (e.g. product specifications, design documents, usage documents) that are available to the obtaining organization and determine their status.
- ▪ Determine the conformance of the previously developed or purchased software to published specifications.
- ▪ Identify the capabilities and limitations of the previously developed or purchased software with respect to the project's requirements.
- ▪ Following an approved test plan, test the safety-critical features of the previously developed or purchased software *independent* of the project's software.
- ▪ Following an approved test plan, test the safety-critical features of the previously developed or purchased software *with* the project's software.
- ▪ Perform a risk assessment to determine if the use of the previously developed or purchased software will result in undertaking an unacceptable level of risk.

### 7.1.2 Integrating OTS Software into your System

Okay, you've weighed all the factors (in-house development costs/time vs. OTS costs/time including glueware and extra tests/analyses), and decided to go ahead with using OTS software. Remember, OTS includes software reused from a different project as well. Now that software must be integrated into the software system, which consists of in-house developed code and/or other OTS/reused code modules. Keeping the OTS code as isolated as possible from the rest of the system is a good idea, and several approaches to doing this are presented below. Reference [1] discusses these approaches, and more.

Making sure the software system is *safe* also requires some additional tests and analyses, which are discussed in section 7.1.4.

#### *7.1.2.1 Sticky stuff: Glueware and Wrapper Functions*

It would be great if the OTS software (including reused libraries and functions from other projects) could just be plunked down into the new system with no additional work. It doesn't work that way, of course. You have to connect the OTS software to the rest of the code. The software that provides the connection is called glueware.

Glueware is a general term for any software that sits between the OTS software and the rest of the software system. Usually it is the software required to connect the two pieces (OTS and in-house) and make them work and play well together. Two specific versions of glueware are *wrappers*, which are described below, and *adapters*, which are discussed in 7.1.2.3.

Wrappers are an encapsulation mechanism, where the OTS code is isolated from the rest of the system. Wrappers can prevent certain inputs from reaching the OTS component and/or check and validate outputs from the component. Restricting inputs should be done if certain values could cause the OTS/reused software to behave badly or execute dormant code. Finding those inputs can be difficult, especially when the source code is unavailable and the documentation is barely adequate. Reference [5] discusses using software fault injection with an OTS component, to determine what undesirable outputs the component can produce, and what inputs lead to those outputs. An experiment in using fault injection with wrapper functions to test the interface robustness of the system is described in reference [3].

Wrappers have several problems when they are applied to OTS software. First, the OTS component's interface must be well understood, which requires more-than-adequate documentation. Outputs that are outside the documented understanding may slip through the wrapper. Second, wrappers may be quite complex, and can approach or exceed the size of the OTS component.

Reference [6] discusses "generic software wrappers", including the development of a "wrapper description language", for wrapping COTS software in a Unix environment. The primary focus of the article is on security issues, but is of interest to anyone considering creating OTS wrappers.

Wrappers have a use outside of the operational software. During debug/testing phase, or while evaluating the OTS software, wrappers can be used to *instrument* the software system. Essentially, the wrapper allows information about what is happening (inputs to the OTS, outputs from the OTS) to be recorded or displayed. This provides insight into what the OTS software is doing in the operational or test situations.

#### *7.1.2.2 Redundant Architecture*

The influence of the COTS component can be restricted by using a redundant architecture. Replication and multi-voting approaches, including n-version programming, can be used if the software produces consistent faults. That is, for a specific input, the same fault is always produced. However, this approach (especially n-version programming) has debatable reliability, and is not recommended.

Partitioning the system into a high-performance portion and a high-assurance kernel (or safety kernel) is another option. The high-performance portion is just that – the best, fastest, leanest, etc. code. This part of the software can contain OTS software, as well as custom-developed software. If this part fails, however, the system defaults to the high-assurance kernel. This portion maintains the system in a safe state while providing the required functionality (with reduced performance).

The Carnegie Mellon Software Engineering Institute (SEI) developed a framework of safety techniques they named Simplex Architecture. These techniques include high-assurance application-kernel technology, address-space protection mechanisms, real-time scheduling algorithms, and methods for dynamic communication among modules. This process requires using analytic redundancy to separate major functions into high-assurance kernels and high-performance subsystems. Off-The-Shelf (OTS) products can be used in the high-performance subsystem and even replaced without bringing down the system. Reference [9] describes the Simplex Architecture.

Redundant architecture is no silver bullet for OTS. It suffers from the same problems as wrapper functions: complexity, and the inability to deal effectively with unknown and unexpected functionality.

#### *7.1.2.3 Adding or Adapting Functionality*

Sometimes the OTS software is *almost* what you want, but is missing some small piece of required functionality. Or the OTS software contains what is needed, but the interfaces don't match up. In either case, a specialized form of glueware called an *adapter* can be written.

If extra functionality is required, an adapter will intercept the input (command, function call) for that functionality, execute the new function which it contains, and return the result – all without invoking the OTS software! Or, the adapter may provide some pre- or post-processing for an OTS function. For example, the OTS software has a function to control 16-bit output ports. The function takes two parameters – port address and value to write to the port. The primary software needs to access a specialize output port. This one requires writing to two consecutive 8-bit ports instead of one 16-bit port. The adapter software intercepts the function call (by matching the port address to the special one). It breaks the 16 bit value passed into 2 8-bit values, then performs two calls to the OTS function to write the values, incrementing the output port address by one between the calls.

When the interfaces between the OTS software and the rest of the code don't match up, an adapter can be written to "translate" between the two. For example, the OTS software produces messages in one format (header, message, checksum), but the standard protocol used by the rest of the system has a different header and uses a CRC instead of a checksum. The adapter would intercept messages coming from the OTS software, modify the header, and calculate the CRC before passing the message on..

#### *7.1.2.4 Dealing with Extra Functionality*

Because OTS software was originally written for another application, or written as a general set of functions, it will often contain "extra" functionality not needed by the current project. This extra code is referred to as "dormant" code, because it should sit there in the software, undisturbed and not executing. The trick is to make sure that's what it really does!

The first step is to identify if the OTS/reused software has any dormant code within it. To adequately determine this, access to the source code is necessary. If source code is unavailable, the software provider may be able to provide information, if you supply a list of the functions you will be using. Product user groups, newsgroups, and web pages may also contain useful information, though always consider the source of the information. If nothing else is available, look through the documentation for defined functions that you will not be using. The higher the ratio of used functionality to total functionality in the OTS software, the less dormant code there is.

Once the presence of dormant code is determined, look for the stimuli that activate it (cause it to execute). That will include command invocation, function calls to specific routines, and possible invocation from required functions based on the software state or parameters passed. If the source code is available, it can be examined for undefined (in the documentation) ways of entering the dormant code.

Also look at the resources the dormant code uses. How much memory space does it take up? How much disk/storage space? Will the presence of this extra code push the software system close to any prescribed limits?

In an ideal system, you will be able to identify any dormant code and verify that it cannot ever be executed. Since we never have an ideal system, contingency planning (risk mitigation) is required. Look at what happens when the dormant code is executed. What functions does it perform? Does it affect the system performance or capacity? Examine the behavior of the system if any of the dormant code is executed – can it go into an unsafe state? Will system performance be degraded, leading to a possible mission or safety issue? Can the dormant software lead to a hazard, or interfere with a hazard control or mitigation?

WARNING: Glueware Needed! Extra Work Ahead!

You have to protect your system (in particular your safety critical functions) from malfunctioning OTS software. This requires wrapping the OTS software (glueware) or providing some sort of firewall. The more dormant code there is in the OTS software, the more likely it is to "trigger" accidentally.

Depending on the issue and the product, you may be able to work with the vendor to disable the dormant code, or provide some safeguards against its unintentional execution. Procedural methods to avoid accidentally invoking the dormant code can be considered, but only as a last resort. They open up many possibilities for human error to cause problems with the system.

How much extra testing will need to be done on the software system is determined by the amount of dormant code, the level of insight into the code (what can it do), and any safety problems with the code. OTS software that could interfere with a safety control, with no source code availability, and with little or no details on its development and testing, will require extra testing! Software with good documentation, well encapsulated (not integrated with the system), and with no ability to affect safety (such as no I/O capability) may not need much extra testing. The determination of the level of testing effort should be made by the software engineers and the safety engineers, with input from project management, based on the risk to the system and to safety that the OTS software imparts.

### 7.1.3 Special Problems with Reused Software

The greatest problem with reusing software created for a different project is psychological. You *know* what the software does, how well it was tested in the previous system, and it just fits so *perfectly* into the new system. You don't need any extra analysis or testing – it's already been done. WRONG! That's what the Ariane 5 team thought when they reused Ariane 4 software! (See reference [10] for details.)

No two systems are alike. You cannot assume that the software you wrote for System A is going to function as expected in System B. The new system may be on a faster processor, and timing problems that weren't apparent in slower System A now become critical. Or the new system may have a critical task that must be executed regularly. The reused code may tie the system up long enough that the critical task is delayed.

It is very important to analyze the reused code in light of the capacity, capability, and requirements of the new system. Look for issues with timing, hogging the system, overwriting variables, using the same system resources for different purposes, and …. As well as other issues. Carefully consider the differences between the new and old systems, and how those differences can affect the functioning of the reused software.

It is a goal in modern software engineering to create reusable software. After all, why should the wheel have to be constantly reinvented? We recycle many things in our society – why not code? While a laudable goal, software reuse is still in its infancy, and all the problems and pitfalls haven't been found yet. Applying reused software to a new system requires a lot of thought *up front*.

**Think *before* you reuse!**

### 7.1.4 Who Tests the OTS? (Us vs. Them)

Hopefully, the OTS software you are about to use has been thoroughly tested, either by the vendor or by the previous project. If you're lucky, you have copies of the test reports. If you're even more lucky, you have a copy of a hazard analysis for the software. You can stop now, right?

Wrong.  Even the most thoroughly tested and analyzed OTS software must still be analyzed and tested for how it operates *in the new system*!  Think about OTS software as a child in a playground.  It may play well with the children in the sandbox and on the slide.  It gets dizzy on the merry-go-round, but still keeps playing. But put it on the swing and the rope breaks, dumping itself on the ground and creating a hazard for other children in the area.  **No two systems are identical.**  Old software must be looked at in the new context.

**Don't assume that if the software works properly,** with no hazard potentials or problems**, in the old environment it will work properly in the new system.**

Safety critical Off-The-Shelf (COTS, GOTS, or reused) software should be analyzed (up front) and tested **by the user**.  Software that resides on the same platform as safety critical software (and is not partitioned from that software), or any "high risk" OTS software are included as safety critical. These analyses and tests should be performed whether the software is modified or not.  **Remember, this is YOUR system.**  The OTS software may have been tested, even thoroughly tested, but not in your system environment.  Ariane 5 [10] demonstrated that well tested software can cause significant problems when applied to a new system (domain).

Your first step is to find out what testing has already been done.  Ideally, the software will be delivered with documentation that includes the tests performed and the results of those tests. If not, ask the vendor for any test or analysis documentation they have.  (It never hurts to ask.) If the software is government supplied or contractor developed for a government program, hazard analyses may be available.  Hazard analyses may be available for other software as well, though it is less likely for commercial software unless developed for other safety regimes (FAA, medical, automobile, etc.).

Existing hazards analyses and test results of unmodified software previously used in safety critical systems may be used as the **basis** for certification in a new system.   The existing analyses and test reports should be **reviewed for their relevance** to the reuse in the new environment.  This means that you need to look at the differences in the environment between the "old" system and the system you wish to use this software in.

If the OTS software causes or affects hazards, you need to address mitigation.  If source code is available, correct the software to eliminate or reduce to an acceptable level of risk any safety hazards discovered during analysis or test.  The corrected software must be retested under identical conditions to ensure that these hazards have been eliminated, and that other hazards do not occur.  If source code is not available, the hazards must be mitigated in another way – wrapping the OTS software, providing extra functionality in the in-house software, removing software control of a hazard cause/control, or even deciding not to use the OTS software. Thoroughly test the system to verify that the hazards have been properly mitigated.

OTS software has taken software and system development by storm, spurred on by decreasing funds and shortened schedules.  Safety engineering is trying to catch up, but the techniques and tests are still under  development.  Providing confidence in the safety of OTS software is still something of a black art.

#### 7.1.3.1 Recommended Analyses and Tests

The hazard analysis (mentioned above) **must** be updated to include the OTS software. Any new hazards that the OTS software adds to the system must be documented, as well as any ability to control a hazard. As much as possible, consider the interactions of the OTS software with the safety critical code. Look for ways that the OTS software can influence the safety critical code. For example,

- overwriting a memory location where a safety critical variable is stored
- getting into a failure mode where the OTS software uses all the available resources and prevents the safety critical code from executing
- clogging the message queue so that safety critical messages do not get through in a timely manner.

Ideally, OTS software should be thoroughly tested in a “stand alone” environment, before being integrated with the rest of the software system. This may have been already done, and the vendor may have provided the documentation. The level of software testing should be determined by the criticality or risk level of the software. High risk safety critical software should be analyzed and tested until it is completely understood.

If the OTS software is safety critical, subject it to as many tests as your budget and schedule allow. The more you know about the software, the less likely it is to cause problems down the road. Since source code is often not available, the primary testing will be black box. Test the range of inputs to the OTS software, and verify that the outputs are as expected. Test the error handling abilities of the OTS software by giving it invalid inputs. Bad inputs should be rejected or set to documented values, and the software should not crash or otherwise display unacceptable behavior. See if the software throws any exceptions, gets into infinite loops, or reaches modes where it excessively uses system resources.

Software fault injection (SFI) is a technique used to determine the robustness of the software, and can be used to understand the behavior of OTS software. It *injects* faults into the software and looks at the results (Did the fault propagate? Was the end result an undesirable outcome?). Basically, the intent is to determine if the software responds gracefully to the injected faults. Traditional software fault inject used modifications of the source code to create the faults. SFI is now being used on the interfaces between components, and can be used even when the source code is unavailable. [5], [3], and [12] discuss software fault injection with COTS software.

The following analyses, if done for the system, should be updated to include the OTS software:

- Timing, sizing and throughput – especially if the system is close to capacity/capability limits.
- Software fault tree, to include faults and dormant code in the OTS software
- Interdependence and Independence Analyses, if sufficient information available
- Design Constraint Analysis
- Code Interface Analysis

- Code Data Analysis
- Interrupt Analysis
- Test coverage Analysis

## *7.2 Contractor-developed Software*

With government downsizing and budget cutting, a large portion of the software previously developed in-house at NASA centers is now being contracted out. Usually, whole systems are developed under the contract, including the software that runs the system.

The NASA Safety Manual (NPG 8715.3), chapter 2, discusses safety and risk management requirements for NASA contracts. Responsibilities of the Project/Program manager, Contracting Officer, and Safety and Mission Assurance personnel are described.

With a contract, especially a performance-based contract, it is usually difficult to specify *how* the contractor develops the software. The end-result is the primary criteria for successful completions. However, with safety critical software and systems, the *how* is very important. The customer needs to have insight into the contractor development processes. This serves two purposes: to identify major problems early, so that they can be corrected; and to give confidence in the final system.

### 7.2.1 Contract Inclusions

Once the contract is awarded, both sides are usually stuck with it. Making sure that the delivered software is what you want starts with writing a good contract. According to NPG 8715.3, the following items should be considered for inclusion in any contract:

- Safety requirements
- Mission success requirements
- Risk management requirements
- Submission and evaluation of safety and risk management documentation from the contractor, such as corporate safety policies, project safety plans, and risk management plans.
- Reporting of mishaps, close calls, and lessons learned
- Surveillance by NASA. Performance-based contracts still have a requirement for surveillance!
- Sub-contracting – require that the safety requirements are passed on to sub-contractors!

**Clear, concise and unambiguous requirements prevent misunderstandings between the customer and the contractor.** Safety requirements should be clearly stated in the specifications. Remember that changing requirements usually involves giving the contractor more money. Do as much thinking about the system safety *up front*, and write the results into the system specification.

### *7.1.2.1 Safety Process*

NASA has a particular process for safety verification of flight projects (both shuttle and ISS). This involves creating a Safety Data Package and going through three levels of reviews at Johnson Space Center. The reviews are by phase. Phase 0/1 is the preliminary review, where the JSC Safety Panel learns about the project, reviews the safety aspects of the preliminary design, and has a chance to input any safety concerns to the project. Phase II (2) usually occurs around the Critical Design Review or during the actual implementation of the design. It is a more in-depth look at the system, hazards, and controls, as well as at the verification process to assure the hazards are mitigated. Phase III (3) occurs near the end of the project, several months before launch. The verification of safety features must be complete, or tracked on a Verification Tracking Log (VTL) if they are still outstanding. All VTL inputs must be completed before the Flight Readiness Review, prior to launch.

It is important to specify in the contract who has responsibility for preparing and presenting the Safety Data Package to the JSC Safety Panel. Software safety will need to be addressed as part of the process.

### *7.1.2.2 Analysis and Test*

The contract should also clearly state what analyses or special tests need to be done for safety and mission assurance, who will do them, and who will see the results. In a performance-based contract, usually the contractor performs all functions, including analyses. In other cases, some of the analyses may be handled from the NASA side. Regardless, the tests and analyses should be spelled out, as well as the responsible party.

Before writing the contract, review sections 4 (development) and 5 (analysis) to determine what development processes, analyses, and tests need to be included in the system specification or contract. Use the guidance on tailoring when imposing processes, analyses and tests.

### *7.1.2.3 Software Assurance and Development Process*

Software Assurance (SA) (also referred to as Software Quality Assurance or Software Product Assurance) is a vital part of successful software development. For a performance-based contract, the SA role is usually handled by a managerially independent group within the contracting company. For other contracts, SA may be performed by NASA SPA personnel, if stated in the requirements. Regardless of *who* performs the SA function, the requirement for a Software Assurance function should be included in the contract. Rather than call out specific roles and responsibilities for SA, requiring use of an accepted standard (IEEE 12207 or CMM level 3, for example), or specifying that the SA responsibilities will be called out in an SA plan that is approved by the NASA project manager, is sufficient.

The contract can also state process requirements that the contractor must meet. For example, software development according to IEEE 12207 may be required. The contractor can be required to have or obtain a certain level of the Capability Maturity Model (CMM). Local ISO requirements (local to the NASA center) may also be imposed. A special method of problem reporting may be required, or the contractor may use their own, established method. It is important that a mechanism exist for NASA to be aware of problems and their corrections, once the software and system reaches a certain level of maturity. A formal Problem

Reporting/Corrective Action process usually begins when the software has reached the first baseline version.

#### *7.1.2.3 Contractor Surveillance*

It is important when imposing requirements on the contractor that a method of monitoring their compliance is also included. Metrics might be selected that will give insight into the software status. The submittal of corroborating data might be required (such as certification to ISO 9000 or CMM level 3). Surveillance of the contractor also needs to be included, and is discussed in section 7.2.2.

#### *7.1.2.4 Software Deliverables*

Making sure you get the software you need is important, but figuring it out *what* you need at the beginning of a project can be difficult. Some of the software deliverables are obvious. Others are often forgotten or overlooked. You also need to consider what happens after the system is delivered and the contractor is no longer part of the project. You, the customer, will have to maintain the system. Make sure you have enough information to do that!

This list encompasses many commonly required or desired software deliverables. It does not include every conceivable software deliverable, but gives a good starting point.

- Operational software – the software that is part of/runs the system. This includes flight software, ground support software, and analysis software.
- Standard project documentation for software, including Software Management Plan, Software Development Plan, Software Assurance Plan, Software Requirements Specification (if not NASA-provided), Verification and Validation Plan, Software Test Plan, and Software Configuration Management Plan. The Risk Management Plan, Safety Plan and Reliability/Maintainability Plan should address software, or point where the risk management, safety, reliability, and maintainability of the software is discussed.
- Design documentation.
- Source code.
- Any development tools used, especially if they are obscure, expensive, or difficult to obtain later.
- Any configuration files, setup files, or other information required to configure the development tools for use with the project.
- Simulators or models developed for use with the operational software. These may be needed to reproduce a test, for updates to the software after delivery, or to understand aspects of the software when errors are found during operation.
- Test software used to verify portions of the software. This includes stubs and drivers from unit and integration tests. Software that generates data for use in testing also falls under this category.
- Software Assurance reports, including process audit results and discrepancy reports.

- ❖ Formal Inspection reports.
- ❖ Test procedures and reports.
- ❖ User/operator manual.

#### 7.1.2.5 Independent Verification and Validation (IV&V)

All NASA projects must complete an evaluation on the need for IV&V or Independent Assessment (IA). NPG 8730 gives the criteria and process for this evaluation. It is important that this evaluation be completed, and agreed to, before the contract is awarded. Extra contractor resources may be needed to aid the IV&V or IA activities. Contractor cooperation with IV&V or IA personnel is expected. At a minimum, management, software development, and software assurance will need to work with the IV&V or IA personnel, providing information, answers, and understanding of the software system and the development process.

Depending on the level of IV&V levied on the project, some independent tests may be performed. These may require some contractor resources to implement.

#### 7.1.2.6 Software Change Process

The requirement to implement a formal software change process should be included in the contract. The process should include a change control board to consider each request. The board should have representatives from various disciplines, including software development, software assurance, systems, safety, and management. Depending on the level of software/hardware integration, someone with an electronics or mechanical understanding may be included, permanently or as the need for such expertise arises.

A NASA/customer representative should be part of the change board, or at least review the board decisions. Some of the changes may impact the ability of the system to meet the requirements, may add or remove functionality, or may impact the safety and reliability of the system.

#### 7.1.2.7 Requirements Specification

**Formal inspection of the requirements/specification by the NASA customer should be used to ensure that the specification is complete and unambiguous.**

Problems found now, before the contract is written, will save money in the long run! Most software problems are actually specification problems, and the fixes become progressively more expensive as the software develops.

You should also have a mechanism in place to facilitate communication between the contractor and the customer. The requirements are rarely completely unambiguous, and some interpretation often occurs. Exchanges between contractors (and subcontractors) and the NASA customer will help to assure that misunderstanding are caught early, ambiguous requirements are clarified, and everyone is on the same page.

### 7.2.2 Monitoring Contractor Processes

NASA contract monitoring for Safety and Mission Assurance (S&MA) takes one of two approaches. *Oversight* is an in-line approach, where NASA personnel work with the contractor as a team member.  For the software portion, NASA personnel may act as Software Assurance, perform audits, witness tests, and perform safety analyses.  They may advise the project on "best practices" or make suggestions of new techniques or tools.

*Insight* is a more "hands off" approach and is often used with performance-based contracts.  The assumption is that the contractor knows what they are doing, and NASA only needs enough insight into their processes to make sure things are functioning properly and that no major problems are brewing.  In this mode, all SA functions are performed by the contractor.  NASA software surveillance consists of reviewing the SA records, "spot auditing" the software development process and the SA process, and participate in major reviews.  Other activities may be performed if requested by the contractor and NASA project management.

Which approach is used, and the specifics of *how* to apply the approach, are called out in a **surveillance plan**.  This plan is produced by the NASA project management once the contract is awarded.  The *who* and *what* details are included in this plan.  *Who* is the responsible party.  For example, the contract may state that NASA will perform the Preliminary Hazard Analysis, but that the contractor will perform all subsequent analyses. *What* would be the list of analyses and special tests that must be performed. *What* is also the list of audits, records and documentation reviews, and other surveillance processes that the NASA S&MA engineer will need to perform, to verify contractor compliance with the process and SA requirements of the contract.

### 7.2.3 Recommended Software Testing

In addition to tests performed by the contractor, you may wish to do additional tests after software delivery. If the environment under which the software safety verification tests were performed has changed (from engineering model to flight system, for example), or if the safety verification tests were not witnessed by NASA personnel, those tests should be rerun (depending on the criticality of the hazards).

Hopefully, all desired tests will have been included in the contract, and the software will be delivered with test reports.  If not, then the software should be subjected to the additional tests upon delivery.

**The software acceptance test should be thorough.**  It should include more than just functional testing.  All "must work" and "must not work" functions should be exercised and verified.  The error handling and fault tolerance of the software must be verified.  You don't want to "break the system", but you also want to make sure that the software can safely handle the "real world" inputs and unanticipated events.

# 8. REFERENCES

References are listed by section, numbered from [1] for each section.

## 1. INTRODUCTION

[1] NASA-STD-8719.13A NASA Software Safety Standard, September 1997

[2] NSTS 13830C Implementation Procedure for NASA Payload System Safety Requirements

[3] NASA-GB-A201 NASA Software Assurance Guidebook, 9/89

[4] Jet Propulsion Laboratory, Software Systems Safety Handbook

[5] Gowen, Lon D, and Collofello, James S. "Design Phase Considerations for Safety Critical Software Systems". Professional Safety, April 1995

## 2. SYSTEM SAFETY PROGRAM

[1] NPG 8715.3 NASA Safety Manual, Chapter-3, System Safety, and Appendix-D (Analysis Techniques)

[2] NSTS 13830C Implementation Procedure for NASA Payload System Safety Requirements

[3] NSTS-22254 Methodology for Conduct of Space Shuttle Program Hazard Analyses

[4] Department of Defense, SOFTWARE SYSTEM SAFETY HANDBOOK, *A Technical & Managerial Team Approach* , Dec. 1999, by Joint Software System Safety Committee

## 3. SOFTWARE SAFETY PLANNING

[1] NASA Software Acquisition Life Cycle SMAP/Version 4.0, 1989

[2] Leveson, Nancy G., "Safeware - System Safety and Computers", Addison-Wesley, Appendix-A Medical Devices - The Therac-25 Story.

[3] NMI 8010.1 "Classification of NASA Space Transportation (STS) Payloads".

[4] MIL-STD-882D Military Standard - Standard Practice for System Safety

[5] International Electrotechnical Committee (IEC), draft standard IEC 1508, "Software for Computers in the Application of Industrial Safety-Related Systems".

## 4. SAFETY CRITICAL SOFTWARE DEVELOPMENT

[1] The Computer Control of Hazardous Payloads - Final Report NASA/JSC/FDSD 24 July 1991

[2] SSP 50038 Computer-Based Control System Safety Requirements International Space Station Alpha

[3] NSTS 19943 Command Requirements and Guidelines for NSTS Customers

[4] STANAG 4404 (Draft) NATO Standardization Agreement (STANAG) Safety Design Requirements and Guidelines for Munitions Related Safety Critical Computing Systems

[5] WSMCR 127-1 Range Safety Requirements - Western Space and Missile Center, Attachment-3 Software System Design Requirements. This document is being replaced by EWRR (Eastern and Western Range Regulation) 127-1, Section 3.16.4 Safety Critical Computing System Software Design Requirements.

[6] AFISC SSH 1-1 System Safety Handbook - Software System Safety, Headquarters Air Force Inspection and Safety Center.

[7] EIA Bulletin SEB6-A System Safety Engineering in Software Development (Electrical Industries Association)

[8] NASA Marshall Space Flight Center (MSFC) Software Safety Standard

[9] Underwriters Laboratory - UL 1998 Standard for Safety - Safety-Related Software, January 4th, 1994

[10] Radley, Charles, 1980, M.Sc. Thesis, Digital Control of a Small Missile, The City University, London, United Kingdom.

[11] Gowen, Lon D. and Collofello, James S. "Design Phase Considerations for Safety-Critical Software Systems". PROFESSIONAL SAFETY, April 1995.

[12] Spector, Alfred and David Gifford. "The Space Shuttle Primary Computer System". Communications of the ACM 27 (1984): 874-900.

[13] Knight, John C, and Nancy G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming." IEEE Transactions on Software Engineering, SE12(1986); 96-109.

[14] Brilliant, Susan S., John C. Knight and Nancy G. Leveson , "Analysis of Faults in an N-Version Software Experiment'" IEEE Transactions on Software Engineering, 16(1990), 238-237.

[15] Brilliant, Susan S., John C. Knight and Nancy G. Leveson , "The Consistent Comparison Problem in N-Version Software." IEEE Transactions on Software Engineering, SE45 (1986); 96-109.

[16] Dahll, G., M. Barnes and P. Bishop. "Software Diversity: Way to Enhance Safety?" Informative and Software Technology. 32(1990); 677-685.

[17] Shimeall, Timotyh J. and Nancy G. Leveson. "An Empirical Comparison of Software Fault Tolerance and Fault Elimination." IEEE Transactions on Software Engineering, 17(1991); 173-182.

[18] Laprie, Jean-Claude, J. Arlat, Christian Beounes and K. Kanoun. "Definitions and Analysis of Hardware and Software, Fault-tolerant Architectures." Computer. July 1990: 39-51.

[19] Arlat, Jean, Karama Kanoun and Jean-Claude Laprie. "Dependability Modeling and Evaluation of Software Fault-tolerant Systems." IEEE Transactions on Computers. 39(1990): 504-513.

[20] Anderson, T. and P. Lee. Fault Tolerance: Principles and Practice, Englewood Cliffs, NJ: Prentice Hall, 1981.

[21] Abbott, Russell J. "Resourceful Systems for Fault Tolerance, Reliability and Safety." ACM Computing Survey. March 1990: 35-68.

[22] Neumann, Peter G. "On Hierarchical Design of Computer Systems for Critic Applications" IEEE Transactions on Software Engineering, 12(1986): 905-920.

[23] Leveson, Nancy G., Stephen S. Cha, John C. Knight and Timothy J. Shimeall, "Use of Self-Checks and Voting in Software Error Detection: An Empirical Study." IEEE Transaction on Software Engineering, SE16(1990); 432-443.

[24] Ould, M. A. "Software Development under DefStan 00-55: A Guide." Information and Software Technology 32(1990): 170-175.

[25] NASA-GB-A302 Formal Methods Specification and Verification Guidebook for Software and Computer Systems.

[26] NSTS 1700.7B Safety Policy and Requirements for Payloads Using the Space Transportation System.

[27] Lutz, Robyn R., Ampo, Yoko, "Experience Report: Using Formal Methods for Requirements Analysis of Critical Spacecraft Software", SEL-94-006 Software Engineering Laboratory Series - Proceedings of the Nineteenth Annual Software Engineering Workshop, NASA GSFC, December 1994.

[28] Butler, Ricky W.; and Finelli, George B.: The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software' IEEE Transactions on Software Engineering, vol. 19, no. 1, Jan 1993, pp 3-12.

[29] Rushby, John: Formal Methods and Digital Systems Validation for Airborne Systems, NASA Contractor Report 4551, December 1993

[30] Miller, Steven P.; and Srivas, Mandayam: Formal Verification of the AAMP5 Microprocessor: A Case Study in the Industrial Use of Formal Methods, presented at WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, April 5-8, 1995, Boca Raton, Florida, USA, pp. 30-43.

[31] Butler, Ricky; Caldwell, James; Carreno, Victor; Holloway, Michael; Miner, Paul; and Di Vito, Beb: NASA Langley's Research and Technology Transfer Program in Formal Methods, in 10th Annual Conference on Computer Assurance (COMPASS 95), Gathersburg, MD, June 1995.

[32] NUREG/CR-6263 MTR 94W0000114 High Integrity Software for Nuclear Power Plants, The MITRE Corporation, for the U.S. Nuclear Regulatory Commission.

[33] Yourdon Inc., "Yourdon Systems Method-Model Driven Systems Development:, Yourdon Press, N.J., 1993.

[34] Dermaco, Tom, "Software State of the Art: Selected Papers", Dorset House, NY, 1990.

[35] Butler, Ricky W. and Finelli, George B.: The Infeasibility of Experimental Quantification of Life-Critical Software Reliability". Proceedings of the ACM Sigsoft '91 Conference on Software for Critical Systems, New Orleans, Louisiana, Dec. 1991, pp. 66-76.

[36] NASA -STD-2202-93 Software Formal Inspections Standard

[37] Model Checking (book) E. M. Clarke, Orna Grumberg, Doron Peled; Mit Press ISBN: 0262032708; Hardcover - 314 pages (December 1999) ;Price: $50.00

[38] Stolper, Steven A. "Designs that Fly! An Approach to Software Architectural Design", Embedded Systems Conference, Fall 1998. http://www.esconline.com/98fallpapers.htm, class 443

[39] Hall, Anthony, "Seven Myths of Formal Methods", IEEE Software, 7(5):11-19, September 1990.

[40] Kemmerer, Richard A., "Integrating Formal Methods into the Development Process", IEEE Software, 7(5):37-50, September 1990.

## 5. SOFTWARE SAFETY ANALYSIS

[1] NASA-STD-2100-91, NASA Software Documentation Standard Software Engineering Program, July 29, 1991

[2] DOD-STD-2167A Military Standard Defense Systems Software Development, Feb. 29, 1988 (this document has been replaced by DOD-STD-498, which was cancelled in 1998 when IEEE 12207 was released)

[3] SSSHB 3.2/Draft JPL Software Systems Safety Handbook

[4] NASA-STD-2202-93 Software Formal Inspections Standard

[5] NASA-GB-A302 Software Formal Inspections Guidebook

[6] Targeting Safety-Related Errors During Software Requirements Analysis, Robyn R. Lutz, JPL. Sigsoft 93 Symposium on the Foundations of Software Engineering.

[7] SSP 30309 Safety Analysis and Risk Assessment Requirements Document - Space Station Freedom Program

[8] Beizer, Boris, “Software Testing Techniques”, Van Nostrand Reinhold, 1990. - (Note: Despite its title, the book mostly addresses analysis techniques).

[9] Beizer, Boris, “Software System Testing and Quality Assurance”, Van Nostrand Reinhold, 1987. (Also includes many analysis techniques).

[10] Yourdon Inc., “Yourdon Systems Method - model driven systems development”, Yourdon Press, N.J., 1993.

[11] DeMarco, Tom, “Software State of the Art: selected papers’, Dorset House, NY, 1990.

[12] Roberts, N., Vesely, W., Haasl, D., and Goldberg, F., Fault Tree Handbook, NUREG-0492, USNRC, 1/86.

[13] Leveson, N., Harvey, P., "Analyzing Software Safety", IEEE Transaction on Software Engineering, Vol. 9, SE-9, No. 5, 9/83.

[14] Leveson, N., Cha, S., and Shimeall, T., "Safety Verification of Ada Programs Using Software Fault Trees", IEEE Software, Volume 8, No. 4, 7/91.

[15] Feuer, A. and Gehani N. "A comparison of the programming languages C and Pascal"ACM Computing Surveys, 14, pp. 73-92, 1982.

[16] Carre, B., Jennings, T., Mac Lennan, F., Farrow, P., and Garnsworthy, J., SPARK The Spade Ada Kernel, 3/88.

[17] Ichbiah J. et al., Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815, 1983.

[18] Wichmann B. "Insecurities in the Ada Programming Language", NPL Report 137/89, 1/89.

[19] Leveson, N. and Stolzy, J., "Safety analysis using Petri-Nets", IEEE Trans on Software Engineering, p. 386-397, 3/87.

[20] Garrett, C., M. Yau, S. Guarro, G. Apostolakais, “Assessing the Dependability of Embedded Software Systems Using the Dynamic Flowgraph Methodology”. Fourth International Working Conference on Dependable Computing for Critical Applications, San Diego Jan 4-6, 1994

[21] BSR/AIAA R-023A-1995 (DRAFT) “Recommended Practice for Human Computer Interfaces for Space System Operations” - Sponsor: American Institute of Aeronautics and Astronautics.

[22] Sha, Liu; Goodenough, John B. “Real-time Scheduling Theory and Ada”, Computer, Vol. 23, April 1990, pp 53-62, Research Sponsored by DOD.

[23] Sha, Liu; Goodenough, John B. “Real-time Scheduling Theory and Ada”, The 1989 Workshop on Operating Systems for Mission Critical Computing, Carnegie-Mellon Univ, Pittsburgh, PA.

[24] Daconta, Michael C. “C Pointers and Dynamic Memory Management" ISBN 0-471-56152-5.

[25] Hatton, Les. “Safer C: Developing Software for High-Integrity and Safety Critical Systems.” McGraw-Hill, New York, 1995. ISBN 0-07-707640-0.

[26] Perara, Roland. “C++ Faux Pas - The C++ language has more holes in it than a string vest.” EXE: The Software Developers’ Magazine, Vol 10 - Issue 6/November 1995.

[27] Plum, Thomas. “C Programming Guidelines”, pub Plum Hall - Cardiff, NJ.\ISBN 0-911537-03-1.

[28] Plum, Thomas. “Reliable Data Structures in C”, pub Plum Hall - Cardiff, NJ. ISBN 0-911537-04-X.

[29] Willis, C. P., and Paddon, D. J. : “Machine Learning in Software Reuse”. Proceedings of the Seventh International Conference in Industrial and Engineering Application of Artificial Intelligence and Expert Systems, 1994, pp. 255-262

[30] Second Safety through quality Conference, 23rd -25th October 1995, Kennedy Space Center, Cape Canaveral, Florida

[31] McDermid, J., "Assurance in High Integrity Software,", High Integrity Software, ed C.T. Sennett, Plenum Press 1989.

[32] De Marco, T., Structured analysis and system specification, Prentice-Hall/Yourdon Inc., NY, NY, 1978.

[33] Wing, J., "A Specifier's Introduction to Formal Methods," Computer Vol. 23 No. 9, September 1990, pp. 8-24.

[34] Meyer, B., "On Formalism in Specification," IEEE Software, Jan 1985, pp. 6-26.

[35] Cullyer, J., Goodenough, S., Wichmann, B., "The choice of computer languages for use in safety-critical systems" Software Engineering Journal, March 1991, pp 51-58.

[36] Carre, B., Jennings, T., Maclennan, F., Farrow, P., Garnsworthy, J., SPARK The Spade Ada Kernel, March 1988.

[37] Boehm, B., "A spiral model of software development and enhancement", IEEE Software Engineering Project Management, 1987, p. 128-142.

[38] Unpublished seminar notes, Professor J. Cullyer, JPL, Jan. 92.

[39] De Marco, T., Structured Analysis and System Specification, Prentice Hall/Yourdon Inc., NY, NY, 1978.

[40] Jaffe , M., N. Leveson, M. Heimdahl, B. Melhart "Software Requirements Analysis for Real-Time Process Control Systems", IEEE Transactions on Software Engineering., Vol. 17, NO. 3 pp. 241-257, March 1991.

[41] Rushby, J. "Formal Specification and Verification of a Fault Masking and Transient Recovery Model for Digital Flight Systems," SRI CSL Technical Report, SRI-CSL-91-03, June 1991.

[42] Reported in "Risks to the public", P. Neumann, Ed., ACM Software Engineering Notes, Vol. 8, Issue 5, 1983.

[43] Parnas, D., van Schouwen, A., and Kwan, S., "Evaluation of Safety-Critical Software", Communications of the ACM, p. 636648, 6/90.

[44] Leveson, N., "Software Safety: Why, What, and How", Computing Surveys, p. 125-163, 6/86.

[45] IEC/65A/WG9, Draft Standard IEC 1508 "Software for Computers In the Application of Industrial Safety-Related Systems", Draft Vers. 1, Sept. 26, 1991.

[46] Horgan, et. al., “Perils of Software Reliability Modeling”, SERC Technical Report, February 3, 1995, http://www.serc.net/TechReports/abstracts/catagory/Reliability.html

[47] Kolcio, et. al., “Integrating Autonomous Fault Management with Conventional Flight Software: A case study”, IEEE, 1999, ????

## 6. SOFTWARE DEVELOPMENT ISSUES

[1] Brown, D., “Solving the Software Safety Paradox”, Embedded Systems Programming, volume 11 number 13, Dec. 1998. http://www.embedded.com/98/9812/9812feat2.htm

[2] Melkonian, M., “Get by Without an RTOS”, Embedded Systems Programming, volume 13, number 10, Sept. 2000. http://www.embedded.com/2000/0009/0009feat4.htm

[3] Bell, R. “Code Generation from Object Models”, Embedded Systems Programming, volume 11, number 3, March 1998. http://www.embedded.com/98/9803fe3.htm

[4] Stewart, D. “30 Pitfalls for Real-Time Software Developers, Part 1 “, Embedded Systems Programming, volume 12 number 10, Oct. 1999. http://www.embedded.com/1999/9910/9910feat1.htm

[5] Stewart, D. “More Pitfalls for Real-Time Software Developers“, Embedded Systems Programming, volume 12 number 11, Nov. 1999 http://www.embedded.com/1999/9911/9911feat2.htm

[6] Barbagallo, T. “Choosing The Right Embedded Software Development Tools”, Integrated Systems Design, http://www.isdmag.com/design/embeddedtools/embeddedtools.html

[7] Dart, S. “Spectrum of Functionality in Configuration Management Systems” Technical Report CMU/SEI-90-TR-11 http://www.sei.cmu.edu/publications/documents/90.reports/90.tr.011.html

[8] OptiMagic, Inc. “Frequently-Asked Questions (FAQ) About Programmable Logic“, http://www.optimagic.com/faq.html

[9] Barr, M. “Programmable Logic: What’s It To Ya?”, Embedded Systems Programming, volume 9, number 6, June 1999 http://www.embedded.com/1999/9906/9906sr.htm

[10] Villasensor, J. and Mangione-Smith, W. H., “Configurable Computing”, Scientific American, June, 1997 http://www.sciam.com/0697issue/0697villasenor.html

[11] “SEMSPLC Guidelines: Safety-related application software for programmable logic controllers". The Institution of Electrical Engineers. ISBN 0 85296 887 6

[12] Canning, et. al., “Sharing Ideas: the SEMSPLC Project”, IEE Review , Volume: 40 Issue: 2 , 17 March 1994

[13] Selic, B. “Distributed Software Design: Challenges and Solutions”, Embedded Systems Programming, Volume 13, number 12, November, 2000 http://www.embedded.com/2000/0011/0011feat5.htm

[14] O’Brien, M. “Embedded Web Servers”, Embedded Systems Programming, Volume 12, Number 11, November, 1999. http://www.embedded.com/internet/9911/9911ia2.htm

[15] Wood, B. “Software Risk Management for Medical Devices”, Medical Device & Diagnostic Industry magazine column, Jan. 1999, http://www.devicelink.com/mddi/archive/99/01/013.html

[16] Schofield, M. “Neither Master nor Slave, A Practical Case Study in the Development and Employment of Cleaning Robots”, IEEE ????, 1999

[17] Smith, et. al. “Validation and Verification of the Remote Agent for Spacecraft Autonomy”, ????

[18] Bernard, et. al. “Remote Agent Experiment DS1 Technology Validation Report”, ????

[19] Reinholz and Patel, “Testing Autonomous Systems for Deep Space Exploration”, IEEE 1998, ????

[20] Simmons, et. al. “Towards Automatic Verification of Autonomous Systems”, IEEE 2000, ????

[21] Birk, Andreas “Autonomous Systems as distributed embedded devices”, http://arti.vub.ac.be/~cyrano/AUTOSYS/

## 7. SOFTWARE ACQUISITION

[1] Lindsay, P. and Smith, G. “Safety Assurance of Commercial-Off-The-Shelf Software” Technical Report No. 00-17, May 2000, Software Verification Research Centre, School of Information Technology, The University of Queensland

[2] Besnard, J., Keene, S., and Voas, J. “Assuring COTS Products for Reliability and Safety Critical Systems”, 1999 Proceedings, Annual Reliability and Maintainability Symposium (IEEE)

[3] Voas, J. and Miller, K. “Interface Robustness for COTS-based Systems”, IEE Colloquium on Cots and Safety Critical Systems (Digest No. 1997/013), 1996, Page(s): 7/1 –712

[4] Fischman, L. and McRitchie, K. “Off-the-Shelf Software: Practical Evaluation”, Crosstalk, Jan. 2000, http://www.stsc.hill.af.mil/crosstalk/2000/jan/fischman.asp

[5] Voas, J. and Payne, J. “COTS Software Failures: Can Anything be Done?”, IEEE Workshop on Application-Specific Software Engineering Technology, 1998. ASSET-98. Proceedings, Page(s): 140 –144

[6] Fraser, T., Badger, L. and Feldman, M. "Hardening COTS Software with Generic Software Wrappers", Proceedings of the 1999 IEEE Symposium on Security and Privacy

[7] "Guidance for Industry, FDA Reviewers and Compliance on Off-the-Shelf Software Use in Medical Devices", Sept. 9, 1999, US Department of Health and Human Services, http://www.fda.gov/cdrh/ode/1252.pdf

[8] Scott, J., Preckshot, G. and Gallagher, J. "Using Commercial-Off-the-Shelf (COTS) Software in High-Consequence Safety Systems", UCRL-JC-122246, Fission Energy and Systems Safety Program (FESSP), Lawrence Livermore National Laboratory, http://www-energy.llnl.gov/FESSP/CSRC/122246.pdf

[9] Sha, L., Goodenough, J. and Pollak, B. "Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems", Crosstalk, April 1998, http://www.stsc.hill.af.mil/crosstalk/1998/apr/simplex.asp

[10] European Space Agency "ARIANE 5: Flight 501 Failure", http://www.esrin.esa.it/tidc/Press/Press96/ariane5rep.html

[11] "The Use of Commercial Off-The-Shelf Software in Safety Critical Projects" by Frances E. Simmons at Johnson Space Center, October 11, 1999. Non-published whitepaper.

## OTHER REFERENCES

### Standards and Guidebooks

NASA Standards and Guidebooks

- NPG 8715.3 NASA Safety Manual
- NASA-CM-GDBK NASA Software Configuration Management Guidebook
- NASA-GB-001-94 NASA Software Measurement Guidebook
- NASA-GB-001-95 NASA Software Process Improvement Guidebook
- NASA-GB-001-96 NASA Software Management Guidebook
- NASA-GB-002-95 Formal Methods Specification And verification Guidebook For Software And Computer Systems, Volume I: Planning And Technology Insertion
- NASA-GB-001-97 Formal Methods Specification And Analysis Guidebook For The verification Of software And computer Systems, volume II: A Practitioner's companion
- NASA-GB-A201 Software Assurance Guidebook
- NASA-GB-A301 Software Quality Assurance Audits guidebook
- NASA-GB-A302 Software Formal Inspections Guidebook
- NASA-STD-2100-91 NASA Software Documentation Standard
- NASA-STD-2201-93 NASA Software Assurance Standard
- NASA-STD-2202-93 Software Formal Inspection Process Standard
- NASA-STD-8719.13A NASA Software Safety Standard
- KHB-1700.7 Space Shuttle Payload Ground Safety Handbook
- NSTS-08126 Problem Reporting and Corrective Action (PRACA) System Requirements
- NSTS-1700-7B Safety Policy and Requirements for Payloads Using the International Space Station, Addendum
- NSTS-1700-7B Safety Policy and Requirements for Payloads Using the Space Transportation System Change No. 6

- NSTS-22206 Requirements for Preparation and Approval of Failure Modes and Effects Analysis (FMEA) and Critical Items List (CIL)
- NSTS-22254 Methodology for Conduct of Space Shuttle Program Hazard Analyses
- NSTS-5300-4(1D-2) Safety, Reliability, Maintainability and Quality Provisions for the Space Shuttle Program Change No. 2
- NSTS-5300.4 Safety, Reliability, Maintainability and Quality Provisions for Space Shuttle Program
- NSTS-ISS-18798 Interpretations of NSTS/ISS Payload Safety Requirements
- SSP-50021 Safety Requirements Document, International Space Station Program
- SSP-50038 Computer-Based Control System Safety Requirements**,** International Space Station Program

IEEE Standards

- ISO/IEC 12207 Information Technology - Software Life Cycle Processes
- EIA 12207.0, .1, .2 Industry Implementation of International Standard ISO/IEC 12207 : 1995
- IEEE 610.12 IEEE Standard Glossary of Software Engineering Terminology
- IEEE 830-1998 IEEE Recommended Practice for Software Requirements Specifications
- IEEE 982.1 IEEE Standard Dictionary Of Measures To Produce Reliable Software
- IEEE 1016-1998 IEEE Recommended Practice for Software Design Descriptions
- IEEE 1228-1994 IEEE Standard for Software Safety Plans

Military Standards

- DoD-STD-498 Software Development and Documentation, cancelled in 1998. Replaced by IEEE 12207.
- MIL-STD-882D System Safety Program Requirements
- MIL-STD-882C System Safety Program Requirements, January 19, 1993
- MIL-STD-882B System Safety Program Requirements, July 1, 1987

Other Standards

- DO-178B Software Considerations in Airborne Systems and Equipment Certification (Federal Aviation Administration).
- AIAA G-010 Reusable Software: Assessment Criteria for Aerospace Applications
- ANSI/AIAA R-013 Recommended Practice: Software Reliability R-013-1992
- ISO 9000-3 Quality Management And Quality Assurance Standards - Part 3: Guidelines For The Application Of ISO 9001: 1994 To The Development, Supply, Installation And Maintenance Of Computer Software Second Edition
- Review Guidelines on Software Languages for Use in Nuclear Power Plant Safety Systems (see http://www.sohar.com/J1030/index.htm)

To access NASA standards and other standards used by NASA: http://standards.nasa.gov/sitemap.htm

**Books**

“Software Safety and Reliability : Techniques, Approaches, and Standards of Key Industrial Sectors”, Debra S. Herrmann, et al., March 2000

“Safeware : System Safety and Computers”, Nancy Leveson, April 1995

“Safety-Critical Computer Systems”, Neil Storey, August 1996

"Software Assessment: Reliability, Safety, Testability", Michael A. Friedman and Jeffrey M. Voas (Contributor), August 16, 1995

"Semsplc Guidelines : Safety-Related Application Software for Programmable Logic Controllers", February 1999

## Websites

### *NASA Websites*

| | |
|---|---|
| NASA Lessons Learned | http://llis.nasa.gov/llis/llis/main.html |
| NASA Technical Standards | http://standards.nasa.gov/sitemap.htm |
| NASA Online Directives Information System (NODIS) Library | http://nodis.hq.nasa.gov/ |
| NASA Documents Online (HQ) | http://www.hq.nasa.gov/office/hqlibrary/books/nasadoc.htm |
| ISS Documentation (PALS) | http://iss-www.jsc.nasa.gov:1532/palsagnt/plsql/palshome |
| NASA Langley Formal Methods group: | http://atb-www.larc.nasa.gov/fm/index.html |
| GSFC Software Engineering Laboratory | http://sel.gsfc.nasa.gov/ |
| NASA Software Assurance Technology Center | http://satc.gsfc.nasa.gov/homepage.html |
| NASA IV&V Center | http://www.ivv.nasa.gov/ |
| NASA Software Working Group | http://swg.jpl.nasa.gov/index.shtml |

### *Reference Websites*

| | |
|---|---|
| Guide to the Software Engineering Body of Knowledge (SWEBOK) | http://www.swebok.org/ |
| Software metrics links: | http://www.totalmetrics.com/resource/links.htm |
| Software Methods and Tools: | http://www.methods-tools.com/html/tools.html |
| Standards (and 37 Cross References) | http://www.cmpcmm.com/cc/standards.html |

### *Software Safety*

| | |
|---|---|
| Software System Safety Working Group | http://sunnyday.mit.edu/safety-club/ |
| Safety critical systems links: | http://archive.comlab.ox.ac.uk/safety.html |
| A Framework for the Development and Assurance of High Integrity Software | http://hissa.ncsl.nist.gov/publications/sp223/ |
| Safety Critical Resources | http://www.cera2.com/WebID/realtime/safety/blank/org/a-z.htm |

### *Software QA and Testing*

| | |
|---|---|
| Society for Software Quality: | http://www.ssq.org/welcome_main.html |
| Software Testing hotlist: | http://www.io.com/~wazmo/qa/ |
| Guidance for Industry, General Principles of Software Validation, Draft Guidance Version 1.1 (FDA) | http://www.fda.gov/cdrh/comp/swareval.html |
| Software Testing Stuff: | http://www.testingstuff.com/testing2.html |
| Software QA/Test Resource Center | http://www.softwareqatest.com/ |
| SR/Institute's Software Quality HotList | http://www.testworks.com/Institute/HotList/index.9.html |
| TestingCraft – tester knowledge exchange | http://www.testingcraft.com/index.html |

### *Miscellaneous*

| | |
|---|---|
| Software Project Survival Guide | http://www.construx.com/survivalguide/chapter.htm |
| Sample Software Documents | http://www.construx.com/doc.htm |
| Software Documents, military, | http://www.pogner.demon.co.uk/mil_498/6.htm |
| Annals of Software Engineering | http://manta.cs.vt.edu/ase/ |
| Software Engineering Readings | http://www.qucis.queensu.ca/Software-Engineering/reading.html |

| | |
|---|---|
| Introduction to Software Engineering | http://www.caip.rutgers.edu/~marsic/Teaching/ISE-online.html |
| Software Engineering hotlist | http://www.cc.gatech.edu/computing/SW_Eng/hotlist.html |
| Brad Appleton's Software Engineering Links | http://www.enteract.com/~bradapp/links/swe-links.html |
| Best Manufacturing Practices guidelines | http://www.bmpcoe.org/guideline/books/index.html |
| Embedded systems programming | http://www.embedded.com/ |
| Embedded systems articles | http://www.ganssle.com/ |

[redacted]

# APPENDIX A

## Glossary of Terms

Various definitions contained in this Glossary are reproduced from IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, copyright 81990 by the Institute of Electrical and Electronic Engineers, Inc. The IEEE takes no responsibility for and will assume no liability for damages resulting from the reader's misinterpretation of said information resulting from the placement and context in this publication.

| **Terminology** | **Definition** |
|---|---|
| Access type | A value of an access type is either a null value or a value that designates an object created by an allocator. The designated object can be read and updated via the access value. The definition of an access type specifies the type of objects designated by values of the access type. |
| Accident | See Mishap. |
| Accreditation Certification | A formal declaration by the Accreditation Authority that a system is approved to operate in a particular manner using a prescribed set of safeguards. |
| Annotations | Annotations are written as Ada comments (i.e. preceded with "— " so the compiler ignores it) beginning with a special character, "#", that signals to the Static code analysis tool that special information is to be conveyed to the tool. |
| Anomalous Behavior | Behavior which is not in accordance with the documented requirements. |
| Anomaly | A state or condition which is not expected. It may or may not be hazardous, but it is the result of a transient hardware or coding error. |
| Architecture | The organizational structure of a system or CSCI, identifying its components, their interfaces and a concept of execution between them. |
| Assertions | A logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during a program execution. Types include input assertion, loop assertion, and output assertion. (IEEE Standard 610.12-1990) |
| Associate Developer | An organization that is neither prime contractor nor subcontractor to the developer, but who has a development role on the same or related project. |
| Assurance | To provide confidence and evidence that a product or process satisfies given requirements of the integrity level and applicable national and international law(s) |

| | |
|---|---|
| Audit | An independent examination of the life cycle processes and their products for compliance, accuracy, completeness and traceability. |
| Audit Trail | The creation of a chronological record of system activities (audit trail) that is sufficient to enable the reconstruction, review and examination of the sequence of environments and activities surrounding or leading to an operation, procedure or an event in a transaction from its inception to its final results. |
| Authenticate | To verify the identity of a user, device or other entity in a system, often as a prerequisite to allowing access to resources in the system. |
| Authorization | The granting of access rights to a user, program or process. |
| Automata | A machine or controlling mechanism designed to follow automatically a predetermined sequence of operations or correct errors or deviations occurring during operation. |
| Baseline | The approved, documented configuration of a software or hardware configuration item, that thereafter serves as the basis for further development and that can be changed only through change control procedures. |
| Battleshort (Safety Arc) | The capability to bypass certain safety features in a system to ensure completion of a mission without interruption due to the safety feature. Bypassed safety features include such items as circuit current overload protection, thermal protection, etc. |
| Build | (1) A version of software that meets a specific subset of the requirements that the completed software will meet.<br><br>(2) The period of time during which a version is developed.<br><br>NOTE: The relationship of the terms "build" and "version" is up to the developer; for example, it may take several versions to reach a build, a build may be released in several parallel versions (such as to different sites), or the terms may be used as synonyms. |
| Built-In Test (BIT) | A design feature of an item which provides information on the ability of the item to perform its intended functions. BIT is implemented in software or firmware and may use or control built in test equipment. |
| Built-In Test Equipment (BITE) | Hardware items that support BIT. |
| Catastrophic Hazard | A hazard which can result in a disabling or fatal personnel injury and/or loss of flight hardware or a ground facility. |
| Caution and Warning (C&W) | Function for detection, annunciation and control of impending or imminent threats to crew safety or mission success. |

| | |
|---|---|
| Certification | Legal recognition by the certification authority that a product, service, organization or person complies with the applicable requirements. Such certification comprises the activity of checking the product, service, organization or person and the formal recognition of compliance with the applicable requirements by issue of a certificate, license, approval or other document as required by national law or procedures. In particular, certification of a product involves: (a) the process of assuring the design of a product to ensure that it complies with a set of standards applicable to that type of product so as to demonstrate an acceptable level of safety; (b) the process of assessing an individual product to ensure that it conforms with the certified type design; (c) the issue of any certificate required by national laws to declare that compliance or conformity has been found with applicable standards in accordance with items (a) or (b) above. |
| Code Safety Analysis (CSA) | An analysis of program code and system interfaces for events, faults, and conditions that could cause or contribute to undesirable events affecting safety. |
| Cohesion | (1) DeMarco: Cohesion is a measure of strength of association of the elements of a module.<br><br>(2) IEEE: The manner and degree to which the tasks performed by a single software module are related to one another. Types include coincidental, communication, functional, logical, procedural, sequential and temporal. |
| Command | Any message that causes the receiving party to perform an action. |
| Computer Hardware | Devices capable of accepting and storing computer data, executing a systematic sequence of operations on computer data, or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control or other logical functions. |
| Computer Program | A combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions. |
| Computer Software Configuration Item (CSCI) | An aggregate of software that is designated for configuration management and is treated as a single entity in the configuration management process. (IEEE Standard 610.12-1990) |
| Concept/Conceptual | The period of time in the software development cycle during which the user needs are described and evaluated through documentation (for example, statement of needs, advance planning report, project initiation memo, feasibility studies, system definition, documentation, regulations, procedures, or policies relevant to the project). |
| Configuration | The requirements, design and implementation that define a particular version of a system or system component. |

| | |
|---|---|
| Configuration Control | The process of evaluating, approving or disapproving, and coordinating changes to configuration items after formal establishment of their configuration identification. |
| Configuration Item | (1) An item that is designated for configuration management.<br><br>(2) A collection of hardware or software elements treated as a unit for the purpose of configuration management.<br><br>(3) An aggregation of hardware, software or both that satisfies an end user function and is designated for separate configuration management by the acquirer. |
| Configuration Management | The process of identifying and defining the configuration items in a system, controlling the release and change of these items throughout the system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items. |
| Control Path | The logical sequence of flow of a control or command message from the source to the implementing effector or function. A control path may cross boundaries of two or more computers. |
| Controlling Application | The lower level application software that controls the particular function and its sensors and detectors. |
| Controlling Executive | The upper level software executive that serves as the overseer of the entire system including the lower level software. |
| COTS | Commercial-off-the-shelf. This refers primarily to commercial software purchased for use in a system. COTS can include operating systems, libraries, development tools, as well as complete applications. The level of documentation varies with the product. Analyses and test results are rarely available. These products are market driven, and usually contain known bugs. |
| Coupling | DeMarco: Coupling is a measure of the interdependence of modules The manner and degree of interdependence between software modules. Types include common-environment coupling, content coupling, control coupling, data coupling, hybrid coupling, and pathological coupling. |
| Coverage | A measure of achieving a complete assessment. 100% coverage is every one of the type specified, e.g. in the situation of test coverage, an assessment of 100% decision coverage is achieved when every one of the decisions in the software has been exercised. |
| Coverage Analysis | An analysis of the degree(s) of coverage assessed. |
| Credible Failure | A condition that has the potential of occurring based on actual failure modes in similar systems. |
| Critical Design Review (CDR) | A review conducted to verify that the detailed design of one or more configuration items satisfy specified requirements; to |

| | |
|---|---|
| | establish the compatibility among configuration items and other items of equipment, facilities, software, and personnel; to assess risk areas for each configuration item; and, as applicable, to assess the results of the producibility analyses, review preliminary hardware product specifications, evaluate preliminary test planning, and evaluate the adequacy of preliminary operation and support documents. (IEEE Standard 610.12-1990)<br><br>For Computer Software Configuration Items (CSCIs), this review will focus on the determination of the acceptability of the detailed design, performance, and test characteristics of the design solution, and on the adequacy of the operation and support documents. |
| Critical Hazard | A hazard which could result in a non-disabling personnel injury, damage to flight hardware or ground support equipment, loss of an emergency system, or use of unscheduled safing procedures. |
| Critical Software Command | A command that either removes a safety inhibit or creates a hazardous condition or state. - See also Hazardous Command. |
| Database | A collection of related data stored in one or more computerized files in a manner that can be accessed by users or computer programs via a database management system. |
| Data Flow Diagram<br>Control Flow Diagram<br>(DFD-CFD) | DFD and CFD diagrams are a graphical representation of a system under the Structured Analysis/Structured Design methodology. Control Flow Diagrams represent the flow of control signals in the system, while Data Flow Diagrams represent the flow of data. |
| Deactivated Code | (1) A software program or routine or set of routines, which were specified, developed and tested for one system configuration and are disabled for a new system configuration. The disabled functions(s) is (are) fully tested in the new configuration to demonstrate that if inadvertently activated the function will result in a safe outcome within the new environment.<br><br>(2) Executable code (or data) which by design is either (a) not intended to be executed (code) or used (data), or (b) which is only executed (code) or used (data) in certain configurations of the target system. |
| Dead Code | (1) Dead Code is code (1) unintentionally included in the baseline, (2) left in the system from an original software configuration, not erased or overwritten and left functional around it, or (3) deactivated code not tested for the current configuration and environment.<br><br>(2) Executable code (or data) which, as a result of design, maintenance, or installation error cannot be executed (code) or used (data) in any operational configuration of the target system and is not traceable to a requirement (e.g., embedded identifier is OK) |

| | |
|---|---|
| Deadlock | A situation in which computer processing is suspended because two or more devices or processes are each awaiting resources assigned to the other. (IEEE Standard 610.12-1990) |
| Debug | The process of locating and eliminating errors that have been shown, directly or by inference, to exist in software. |
| Degree of Demonstration | Extent to which evidence is produced to provide confidence that specified requirements are fulfilled (ISO 8402, 4.5). Note the extent depends on criteria such as economics, complexity, innovation, safety and environmental considerations. |
| Developer | The organization required to carry out the requirements of this standard and the associated contract. The developer may be a contractor or a Government agency. |
| Development Configuration | The requirements, design and implementation that define a particular version of a system or system component. |
| Document/Documentation | A collection of data, regardless of the medium on which it is recorded, that generally has permanence and can be read by humans or machines. |
| Dormant Code | Similar to dead code, it is software instructions that are included in the executable but not meant to be used. Dormant code is usually the result of COTS or reused software that include extra functionality over what is required. |
| Dynamic allocation | Dynamic allocation is the process of requesting to the operating system memory storage for a data structure that is used when required by the application program's logic. Successful allocation of memory (if memory space is available) may be from the task's heap space. |
| Emulator | A combination of computer program and hardware that mimic the instruction and execution of another computer or system. |
| Environment | (1) The aggregate of the external procedures, conditions and objects that affect the development, operation and maintenance of a system.<br>(2) Everything external to a system which can affect or be affected by the system. |
| Error | (1) Mistake in engineering, requirement specification, or design.<br>(2) Mistake in design, implementation or operation which could cause a failure. |
| Error Handling | An implementation mechanism or design technique by which software faults are detected, isolated and recovered to allow for correct runtime program execution. |
| Exception | Exception is an error situation that may arise during program execution. To raise an exception is to abandon normal program execution to signal that the error has taken place. |

| | |
|---|---|
| Fail-Safe | (1) Ability to sustain a failure and retain the capability to safely terminate or control the operation.<br><br>(2) A design feature that ensures that the system remains safe or will cause the system to revert to a state which will not cause a mishap. |
| Failure | The inability of a system or component to perform its required functions within specified performance requirements. (IEEE Standard 610.12-1990) |
| Failure Tolerance | The ability of a system or subsystem to perform its function(s) or maintain control of a hazard in the presence of failures within its hardware, firmware, or software. |
| Fault | Any change in state of an item that is considered to be anomalous and may warrant some type of corrective action. Examples of faults included device errors reported by Built-In Test (BIT)/Built-In Test Equipment (BITE), out-of-limits conditions on sensor values, loss of communication with devices, loss of power to a device, communication error on bus transaction, software exceptions (e.g., divide by zero, file not found), rejected commands, measured performance values outside of commanded or expected values, an incorrect step, process, or data definition in a computer program, etc. Faults are preliminary indications that a failure may have occurred. |
| Fault Detection | A process that discovers or is designed to discover faults; the process of determining that a fault has occurred. |
| Fault Isolation | The process of determining the location or source of a fault. |
| Fault Recovery | A process of elimination of a fault without permanent reconfiguration. |
| Fault Tree | A schematic representation, resembling an inverted tree, of possible sequential events (failures) that may proceed from discrete credible failures to a single undesired final event (failure). A fault tree is created retrogressively from the final event by deductive logic. |
| Finite State Machine | Also known as: Requirements State Machine, State of Finite Automation Transition Diagram.<br><br>A model of a multi-state entity, depicting the different states of the entity, and showing how transitions between the states can occur. A finite state machine consists of:<br>1. A finite set of states<br>2. A finite set of unique transitions. |
| Firmware | Computer programs and data loaded in a class of memory that cannot be dynamically modified by the computer during processing (e.g. ROM). |
| Flight Hardware | Hardware designed and fabricated for ultimate use in a vehicle intended to fly. |

| | |
|---|---|
| Formal Methods System | (1) The use of formal logic, discrete mathematics, and machine-readable languages to specify and verify software.<br><br>(2) The use of mathematical techniques in design and analysis of the system. |
| Formal Verification | (For Software) The process of evaluating the products of a given phase using formal mathematical proofs to ensure correctness and consistency with respect to the products and standards provided as input to that phase. |
| GOTS | Government-off-the-shelf. This refers to government created software, usually from another project. The software was not created by the current developers (see *reused software*). Usually, source code is included and all available documentation, including test and analysis results. |
| Graceful Degradation | (1) A planned stepwise reduction of function(s) or performance as a result of failure, while maintaining essential function(s)/performance.<br><br>(2) The capability of continuing to operate with lesser capabilities in the face of faults or failures or when the number or size of tasks to be done exceeds the capability to complete. |
| Graph theory | An abstract notation that can be used to represent a machine that transitions through one or more states. |
| Ground Support Equipment (GSE) | Ground-based equipment used to store, transport, handle, test, check out, service, and/or control aircraft, launch vehicles, spacecraft, or payloads. |
| Hardware Configuration Item (HWCI) | An aggregation of a hardware device and computer instructions and/or computer data that reside as read-only software on a hardware device. |
| Hazard | The presence of a potential risk situation caused by an unsafe act or condition. A condition or changing set of circumstances that presents a potential for adverse or harmful consequences; or the inherent characteristics of any activity, condition or circumstance which can produce adverse or harmful consequences. |
| Hazard Analysis | The determination of potential sources of danger and recommended resolutions in a timely manner for those conditions found in either the hardware/software systems, the person/machine relationship, or both, which cause loss of personnel capability, loss of system, loss of life, or injury to the public. |
| Hazard Cause | A component level Fault or failure which can increase the risk of, or result in a hazard. |
| Hazard Control | Design or operational features used to reduce the likelihood of occurrence of a hazardous effect. |

| | |
|---|---|
| Hazardous Command | A command whose execution (including inadvertent, out-of-sequence, or incorrectly executed) could lead to an identified critical or catastrophic hazard, or a command whose execution can lead to a reduction in the control of a hazard (including reduction in failure tolerance against a hazard or the elimination of an inhibit against a hazard). |
| Hazardous State | A state that may lead to an unsafe state. |
| Hazard Report | The output of a hazard analysis for a specific hazard which documents the hazard title, description, causes, control, verification, and status. |
| Hazard Risk Index | A combined measure of the severity and likelihood of a hazard. See Table in Section 2. |
| Hazard Severity | An assessment of the consequences of the worst credible mishap that could be caused by a specific hazard. |
| Higher Order Logic | A functional language for specifying requirements, used in Formal Methods. |
| Independent Assessment (IA) | A formal process of assessing (auditing) the development, verification, and validation of the software. An IA does not perform those functions (as in IV&V), but does evaluate how well the project did in carrying them out. |
| Independent Inhibit | Two inhibits are independent if no SINGLE failure, error, event, or environment can eliminate more than one inhibit. Three inhibits are independent if no TWO failures, errors, events or environments (or any pair of one of each) can eliminate more than two inhibits. |
| Independent Verification and Validation (IV & V) | A process whereby the products of the software development life cycle phases are independently reviewed, verified, and validated by an organization that represents the acquirer of the software and is completely independent of the provider. |
| Inhibit | A design feature that provides a physical interruption between an energy source and a function (e.g., a relay or transistor between a battery and a pyrotechnic initiator, a latch valve between a propellant tank and a thruster, etc.). |
| Interface | In software development, a relationship among two or more entities (such as CSCI-CSCI, CSCI-HWCI, CSCI-User or software unit-software unit) in which the entities share, provide or exchange data. An interface is not a CSCI, software unit or other system component; it is a relationship among them. |
| Interface Hazard Analysis | Evaluation of hazards which cross the interfaces between a specified set of components, elements, or subsystems. |
| Interlock | Hardware or software function that prevents succeeding operations when specific conditions exist. |

| | |
|---|---|
| Life Cycle | The period of time that starts when a software product is conceived and ends when the software is no longer available for use. The software life cycle traditionally has eight phases: Concept and Initiation; Requirements; Architectural Design; Detailed Design; Implementation; Integration and Test; Acceptance and Delivery; and Sustaining Engineering and Operations. |
| Machine Code | Low level language Computer software, usually in binary notation, unique to the processor object in which it is executed. The same as object code. |
| Maintainability | The ability of an item to be retained in or restored to specified condition when maintenance is performed by personnel having specified skill levels, using prescribed procedures, resources and equipment at each prescribed level of maintenance and repair. |
| Marginal Hazard | A hazard whose occurrence would result in minor occupational injury or illness or property damage. |
| Memory Integrity | The assurance that the computer program or data is not altered or destroyed inadvertently or deliberately. |
| Mishap | An unplanned event or series of events that results in death, injury, occupational illness, or damage to or loss of equipment, property, or damage to the environment; an accident. |
| N-Version Software | Software developed in two or more versions using different specifications, programmers, languages, platforms, compilers, or combinations of some of these. This is usually an attempt to achieve independence between redundant software items. Research has shown that this method usually does not achieve the desired reliability, and it is no longer recommended. |
| Negative Testing | Software Safety Testing to ensure that the software will not go to a hazardous state or generate outputs that will create a hazard in the system in response to out of bound or illegal inputs. |
| Negligible Hazard | Probably would not affect personnel safety but is a violation of specific criteria. |
| No-Go Testing | Software Safety Testing to ensure that the software performs known processing and will go to a known safe state in response to specific hazardous situations. |
| Object Code | Low level language Computer software, usually in binary notation, unique to the processor object in which it is executed. The same as machine code. |
| Objective Evidence | Information which can be proved true, based on facts obtained through observation, measurement, test or other means. |

| | |
|---|---|
| Operator Error | An inadvertent action by flight crew or ground operator that could eliminate, disable, or defeat an inhibit, redundant system, containment feature, or other design features that is provided to control a hazard. |
| Override | The forced bypassing of prerequisite checks on the operator-commanded execution of a function. Execution of any command (whether designated as a “hazardous command” or not) as an override is considered to be a hazardous operation requiring strict procedural controls and operator safing. (ISS) |
| Patch | (1) A modification to a computer sub-program that is separately compiled inserted into machine code of a host or parent program. This avoids modifying the source code of the host/parent program. Consequently the parent/host source code no longer corresponds to the combined object code.<br><br>(2) A change to machine code (object code) representation of a computer program and by-passing the compiler |
| Path | The logical sequential structure that the program must execute to obtain a specific output. |
| Peer Review | An overview of a computer program presented by the author to others working on similar programs in which the author must defend his implementation of the design.<br><br>Note: A phase does not imply the use of any specific life-cycle model, nor does it imply a period of time in the development of a software product. |
| Predicate | Predicate is any expression representing a condition of the system. |
| Preliminary Design Review (PDR) | A review conducted to evaluate the progress, technical adequacy, and risk resolution of the selected design approach for one or more configuration items; to determine each design's compatibility with the requirements for the configuration item; to evaluate the degree of definition and assess the technical risk associated with the selected manufacturing methods and processes; to establish the existence and compatibility of the physical and functional interfaces among the configuration items and other items of equipment, facilities, software, and personnel; and as appropriate, to evaluate the preliminary operation and support documents. (IEEE Standard 610.12-1990)<br><br>For CSCIs, the review will focus on:<br>(1) the evaluation of the progress, consistency, and technical adequacy of the selected architectural design and test approach,<br><br>(2) compatibility between software requirements and architectural design, and<br><br>(3) the preliminary version of the operation and support documents. |

| | |
|---|---|
| Preliminary Hazard Analysis (PHA) | Analysis performed at the system level to identify safety-critical areas, to provide an initial assessment of hazards, and to identify requisite hazard controls and follow-on actions. |
| Program Description Language (PDL) | PDL is used to describe a high level design that is an intermediate step before actual code is written. |
| Redundancy | Provision of additional functional capability (hardware and associated software) to provide at least two means of performing the same task. |
| Regression Testing | The testing of software to confirm that functions, that were previously performed correctly, continue to perform correctly after a change has been made. |
| Reliability | The probability of a given system performing its mission adequately for a specified period of time under the expected operating conditions. |
| Rendezvous | A rendezvous is the interaction that occurs between two parallel tasks when one task has called an entry of the other task, and a corresponding accept statement is being executed by the other task on behalf of the calling task. |
| Requirement(s) | (1) Condition or capability needed by a user to solve a problem or achieve an objective. |
| | (2) Statements describing essential, necessary or desired attributes. |
| Requirements, Derived | (1) Essential, necessary or desired attributes not explicitly documented, but logically implied by the documented requirements. |
| | (2) Condition or capability needed, e.g. due to a design or technology constraint, to fulfill the user's requirement(s). |
| Requirements, Safety | Those requirements which cover functionality or capability associated with the prevention or mitigation of a hazard. |
| Requirement Specification | Specification that sets forth the requirements for a system or system component. |
| Requirements State Machine | See Finite State Machine |
| Reusable Software | A software product developed for one use but having other uses, or one developed specifically to be usable on multiple project or in multiple roles on one project. Examples include, but are not limited to, commercial-off-the-shelf software (COTS) products, acquirer-furnished software products, software products in reuse libraries, and pre-existing developer software products. Each use may include all or part of the software product and may involve its modification. This term can be applied to any software product (for example, requirements, architectures, etc.), not just to software itself. |

| | |
|---|---|
| Reused Software | This is software previously written by an in-house development team and used on a different project. GOTS software would come under this category if it is supplied to another government project. Because this software was verified and validated for a previous project, it is often assumed to work correctly in the new system. Each piece of reused software should be thoroughly analyzed for its operation in the new system. Remember the problems when the Ariane 4 software was used in Ariane 5! |
| Risk | (1) As it applies to safety, exposure to the chance of injury or loss. It is a function of the possible frequency of occurrence of the undesired event, of the potential severity of resulting consequences, and of the uncertainties associated with the frequency and severity.<br><br>(2) A measure of the severity and likelihood of an accident or mishap.<br><br>(3) The probability that a specific threat will exploit a particular vulnerability of the system. |
| Safe (Safe State) | (1) The state of a system defined by having no identified hazards present and no active system processes which could lead to an identified hazard.<br><br>(2) A general term denoting an acceptable level of risk, relative freedom from and low probability of: personal injury; fatality; loss or damage to vehicles, equipment or facilities; or loss or excessive degradation of the function of critical equipment. |
| Safety | Freedom from hazardous conditions. |
| Safety Analysis | A systematic and orderly process for the acquisition and evaluation of specific information pertaining to the safety of a system. |
| Safety Architectural Design Analysis (SADA) | Analysis performed on the high-level design to verify the correct incorporation of safety requirements and to analyze the Safety-Critical Computer Software Components (SCCSCs). |
| Safety-Critical | Those software operations that, if not performed, performed out-of sequence, or performed incorrectly could result in improper control functions (or lack of control functions required for proper system operation) that could directly or indirectly cause or allow a hazardous condition to exist. |
| Safety-Critical Computer Software Component (SCCSC) | Those computer software components (processes, modules, functions, values or computer program states) whose errors (inadvertent or unauthorized occurrence, failure to occur when required, occurrence out of sequence, occurrence in combination with other functions, or erroneous value) can result in a potential hazard, or loss of predictability or control of a system. |
| Safety-Critical Computing System | A computing system containing at least one Safety-Critical Function. |

| | |
|---|---|
| Safety-Critical Computing | Those computer functions in which an error can result in a potential hazard to the user, friendly forces, materiel, third parties or the environment. |
| Safety-Critical Software | Software that:<br><br>(1) Exercises direct command and control over the condition or state of hardware components; and, if not performed, performed out-of-sequence, or performed incorrectly could result in improper control functions (or lack of control functions required for proper system operation), which could cause a hazard or allow a hazardous condition to exist.<br><br>(2) Monitors the state of hardware components; and, if not performed, performed out-of-sequence, or performed incorrectly could provide data that results in erroneous decisions by human operators or companion systems that could cause a hazard or allow a hazardous condition to exist.<br><br>(3) Exercises direct command and control over the condition or state of hardware components; and, if performed inadvertently, out-of-sequence, or if not performed, could, in conjunction with other human, hardware, or environmental failure, cause a hazard or allow a hazardous condition to exist. |
| Safety Detailed Design Analysis (SDDA) | Analysis performed on Safety-Critical Computer Software Components to verify the correct incorporation of safety requirements and to identify additional hazardous conditions. |
| Safety Kernel | An independent computer program that monitors the state of the system to determine when potentially unsafe system states may occur or when transitions to potentially unsafe system states may occur. The Safety Kernel is designed to prevent the system from entering the unsafe state and return it to a known safe state. |
| Safing | The sequence of events necessary to place systems or portions thereof in predetermined safe conditions. |
| Sensor | A transducer that delivers a signal for input processing. |
| Separate Control Path | A control path which provides functional independence to a command used to control an inhibit to an identified critical or catastrophic hazard. |
| Software | (1) Computer programs and computer databases.<br><br>Note: although some definitions of software include documentation, MIL-STD-498 limits the definition to programs and computer databases in accordance with Defense Federal Acquisition Regulation Supplement 227.401 (MIL-STD-498).<br><br>(2) Organized set of information capable of controlling the operation of a device. |

| | |
|---|---|
| Software Assurance (SA) | The process of verifying that the software developed meets the quality, safety, reliability, security requirements as well as technical and performance requirements. Assurance looks at both the process used to develop the software and the analyses and tests performed to verify the software. Software Quality Assurance (SQA) and Software Product Assurance (SPA) are sometimes used interchangeably with Software Assurance. |
| Software Controllable Inhibit | A system-level hardware inhibit whose state is controllable by software commands. |
| Software Error | The difference between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. |
| Software Fault | An incorrect step, process or data definition in a computer system. |
| Software Inhibit | A software or firmware feature that prevents a specific event function from occurring or a specific function from being available. The software may be resident in any medium. (A software inhibit is not in itself an "inhibit" in the sense of providing a physical interrupt between an energy source and a function.) |
| Software Partitioning | Separation, physically and/or logically, of (safety-critical) functions from other functionality. |
| Software Requirements Review (SRR) | A review of the requirements specified for one or more software configuration items to evaluate their responsiveness to and interpretation of system requirements and to determine whether they form a satisfactory basis for proceeding into a preliminary (architectural) design of configuration items. (IEEE Standard 610.12-1990)<br><br>Same as Software Specification Review for MIL-STD-498. |
| Software Requirements Specification (SRS) | Documentation of the essential requirements (functions, performance, design constraints, and attributes) of the software and its external interfaces. (IEEE Standard 610.12-1990) |

| | |
|---|---|
| Software Safety Requirements Analysis (SSRA) | Analysis performed to examine system and software requirements and the conceptual design in order to identify unsafe modes for resolution, such as out-of-sequence, wrong event, deadlocking, and failure-to-command modes. |
| Software Specification Review (SSR) | Same as Software Requirements Review. |
| Software Safety | The application of the disciplines of system safety engineering techniques throughout the software life cycle to ensure that the software takes positive measures to enhance system safety and that errors that could reduce system safety have been eliminated or controlled to an acceptable level of risk. |
| Software Safety Engineering | The application of System Safety Engineering techniques to software development in order to ensure and verify that software design takes positive measures to enhance the safety of the system and eliminate or control errors which could reduce the safety of the system. |
| System Safety | Application of engineering and management principles, criteria, and techniques to optimize safety and reduce risks within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle. |
| Software Specification Review (SSR) | Same as Software Requirements Review |
| State Transition Diagram | (See also Finite State Machine). Directed graph used in many Object Oriented methodologies, in which nodes represent system states and arcs represent transitions between states. |
| System | A set of components which interact to perform some function or set of functions. |
| System Safety | Application of engineering and management principles, criteria, and techniques to optimize safety and reduce risks within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle. |
| System Safety Engineering | An engineering discipline requiring specialized professional knowledge and skills in applying scientific and engineering principles, criteria, and techniques to identify and eliminate hazards, or reduce the associated risk. |
| System Safety Management | A management discipline that defines system safety program requirements and attempts to ensure the planning, implementation and accomplishment of system safety tasks and activities consistent with the overall program requirements. |
| System Specification | Document stating requirements for the system. |
| Test Case | A set of test inputs, execution conditions and expected results used to determine whether the expected response is produced. |
| Testing | The process of executing a series of test cases and evaluating the results. |

| | |
|---|---|
| Test Procedure | (1) Specified way to perform a test.<br><br>(2) Detailed instructions for the set-up and execution of a given set of test cases and instructions for the evaluation of results executing the test cases. |
| Test Readiness Review (TRR) | A review conducted to evaluate preliminary test results for one or more configuration items; to verify that the test procedures for each configuration item are complete, comply with test plans and descriptions, and satisfy test requirements; and to verify that a project is prepared to proceed to formal test of the configuration items. (IEEE Standard 610.12-1990) |
| Test, Stress | For software, this is testing by subjecting the software to extreme external conditions and anomalous situations in which the software is required to perform correctly. |
| Time to Criticality | The time between the occurrence of a failure, event or condition and the subsequent occurrence of a hazard or other undesired outcome. |
| Traceability | Traceability for software refers to documented mapping of requirements into the final product, through all development life cycles. |
| Transition | A transition is when an input causes a state machine to change state. |
| Trap | Software feature that monitors program execution and critical signals to provide additional checks over and above normal program logic. Traps provide protection against undetected software errors, hardware faults, and unexpected hazardous conditions. |
| Trigger | Triggers are one or more conditions that when all are true enable a specific action to take place. |
| Type | (As used in software design). A type characterizes both a set of values and a set of operations applicable to those values. Typing of variables can be strong or weak. Strong typing is when only defined values of a variable and defined operations are allowed. Weak typing refers to when the restrictions that are applied are very loose (i.e. a declaration of type integer with no range or operation definition). |
| Undocumented Code | Software code that is used by the flight software but is not documented in the software design. Usually this pertains to Commercial Off-the-Shelf Software(COTS) because the documentation is not always available. |
| Unsafe State | A system state that may result in a mishap. |
| Unused Code | Software code that resides in the flight software that is not intended for use during nominal or contingency situation. Examples are test code, no-oped coded (code that is bypassed), |

| | |
|---|---|
| | and code that is retained by not being used from one operational increment to the next. |
| Validation | (1) An evaluation technique to support or corroborate safety requirements to ensure necessary functions are complete and traceable.<br><br>(2) The process of evaluating software at the end of the software development process to ensure compliance with software requirements.<br><br>(3) Confirmation by examination and provision of objective evidence that the particular requirements for a specific use are fulfilled (for software). The process of evaluating software to ensure compliance with specified.<br><br>(4) The process of determining whether the system operates correctly and executes the correct functions. |
| Verification | (1) The process of determining whether the products of a given phase of the software development cycle fulfill the requirements established during the previous phase(s) (see also validation).<br><br>(2) Formal proof of program correctness.<br><br>(3) The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether items, processes, services, or documents conform to specified requirements.<br><br>(4) Confirmation by examination and provision of objective evidence that specified requirements have been fulfilled (for software).<br><br>(5) The process of evaluating the products of a given phase to determine the correctness and consistency of those products with respect to the products and standards provided as input to that phase. |
| Waiver | A variance that authorizes departure from a particular safety requirement where alternate methods are employed to mitigate risk or where an increased level of risk has been accepted by management. |
| Watchdog Timer | An independent, external timer that ensures the computer cannot enter an infinite loop. Watchdog timers are normally reset by the computer program. Expiration of the timer results in generation of an interrupt, program restart, or other function that terminates current program execution. |

### *Acronyms*

| Acronym | Term |
|---|---|
| BIT | Built-In Test |
| BITE | Built-In Test Equipment |
| C&W | Caution and Warning |
| CASE | Computer Aided Software Engineering |
| CDR | Critical Design Review |
| CFD | Control Flow Diagram |
| COTS | Commercial Off-the-Shelf |
| CSA | Code Safety Analysis |
| CSC | Computer Software Component |
| CSCI | Computer Software Configuration Item |
| CSU | Computer Software Unit |
| DFD | Data Flow Diagram |
| DID | Data Item Description |
| DLA | Design Logic Analysis |
| DoD | Department of Defense |
| FDIR | Fault Detection, Isolation, and Recovery |
| FMEA | Failure Modes and Effects Analysis |
| FQR | Formal Qualifications Review |
| GFE | Government Furnished Equipment |
| GSE | Ground Support Equipment |
| HOL | Higher Order Logic |
| HRI | Hazard Risk Index |
| ICD | Interface Control Document |
| IOS | International Organization for Standardization |
| ISO | from the Greek root isos meaning equal or standard (not an acronym). ISO standards are published by IOS |
| IV&V | Independent Verification and Validation |
| JPL | Jet Propulsion Laboratory, Pasadena, California |
| MIL-STD | Military Standard |
| NDI | Non-Developmental Item |
| NHB | NASA Handbook |
| NDS | Non-Developmental Software |
| NMI | NASA Management Instruction |
| NPD | NASA Policy Directive |
| NPG | NASA Procedures and Guidelines |
| NSTS | National Space Transportation System |
| OO | Object Oriented |
| OOA | Object Oriented Analysis |
| OOD | Object Oriented Development |
| QA | Quality Assurance |
| PDL | Program Description Language |

| | |
|---|---|
| PHA | Preliminary Hazard Analysis |
| POST | Power On Self Test |
| S&MA | Safety and Mission Assurance |
| SA | Software Assurance |
| SADA | Safety Architectural Design Analysis |
| SCCSC | Safety Critical Computer Software Component |
| SDDA | Safety Detailed Design Analysis |
| SEE | Software Engineering Environment |
| SIS | Software Interface Specification |
| SPA | Software Product Assurance |
| SQA | Software Quality Assurance |
| SRD | Software Requirements Document |
| SRR | Software Requirements Review |
| SRS | Software Requirements Specifications |
| SSR | Software Specification Review |
| SSRA | Software Safety Requirements Analysis |
| TRR | Test Readiness Review |

# APPENDIX B Software Fault Tree Analysis (SFTA)

This section is provided to assist systems safety engineers and software developers with an introductory explanation of the Software Fault Tree Analysis technique. Most of the information presented in this entry is extracted from Leveson et al.12,31.

**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.

FTA was originally developed in the 1960's for safety analysis of the Minuteman missile system. It has become one of the most widely used hazard analysis techniques. In some cases FTA techniques may be mandated by civil or military authorities.

## *B.1 Software Fault Tree Analysis Description*

The fault tree handbook from the U.S. Nuclear Regulatory Commission gives the following description of the technique [1]:

"Fault tree analysis can be simply described as an analytical technique, whereby an undesired state of the system is specified (usually a state that is critical from a safety standpoint), and the system is then analyzed in the context of its environment and operation to find all credible ways in which an undesired event can occur. The fault tree is a graphic model of the various parallel and sequential combinations of faults (or system states) that will result in the predefined undesired event. A fault tree thus depicts the logical relationships of basic events that lead to the undesired event - which is the top event of the fault tree".

A sample fault tree is shown in Figure B- 1 :- SFTA Graphical Representation Symbols.

## *B.2 Goal of Software Fault Tree Analysis*

SFTA is a technique to analyze the safety of a software design. The goal of SFTA is to show that the logic in a software design or in an implementation (actual code) will not produce a hazard. The design or code is modified to compensate for those failure conditions deemed to be hazardous threats to the system. In this manner, a system with safer operational characteristics is produced. SFTAs are most practical to use when we know that the system has relatively few states that are hazardous

Developers typically use forward inference to design a system. That is, their analysis focuses on generating a next state from a previously safe state. The software is developed with key assumptions about the state of the system prior to entering the next state. In complex systems that rely on redundancy, parallelism, or fault tolerance, it may not be feasible to go exhaustively through the assumptions for all cases.

The SFTA technique provides an alternative perspective that uses backward inference. The experience from projects that have employed SFTA shows that this change of perspective is crucial to the issue of finding safety errors. The analyst is forced to view the system from a different perspective, one that makes finding errors more apparent.

SFTA is very useful for determining the conditions under which fault tolerance and fail safe procedures should be initiated. The analysis can help guide safety engineers in the development of safety critical test cases by identifying those areas most likely to cause a hazard. On larger systems, this type of analysis can be used to identify safety critical software modules, if they have not already been identified.

SFTA is language independent and can be applied to any programming language (high level or low level) as long as the semantics are well defined. The SFTA is an axiomatic verification where the postconditions describe the hazard rather than the correctness condition [3]. This analysis shows that, if the weakest precondition is false, the hazard or postcondition can never occur and conversely, if the precondition is true, then the program is inherently unsafe and needs to be changed.

Software fault trees should not be reviewed in isolation from the underlying hardware, because to do so would deny a whole class of interface and interaction problems. Simulation of human failure such as operator mistakes can also be analyzed using the SFTA technique.

The symbols used for the graphical representation of the SFTA, to a large extent, have been borrowed from the hardware fault tree set (see Figure B-1:- SFTA Graphical Representation Symbols) [2]. This facilitates the linking of hardware and software fault trees at their interfaces to allow the entire system to be analyzed.

The SFTA makes no claim as to the reliability of the software. When reusing older modules, a new safety analysis is necessary because the fundamental safety assumptions used in the original design must be validated in the new environment. The assertion that highly reliable software is safe is not necessarily true. In fact safety and reliability at times run counter to each other. An example of this conflict can be found in the actual experience of air traffic controllers from the U.S. who attempted to port an air traffic control software application from the U.S. to Britain. The U.S. software had proved to be very reliable but certain assumptions had been made about longitude (i.e., no provision for <u>both</u> east and west coordinates) that caused the map of Britain to fold in half at the Greenwich meridian [3]).

SFTA is not a substitute for the integration and test procedures that verify functional system requirements. The traditional methods that certify that requirements are correct and complete will still need to be used. The SFTA helps provide the extra assurance that is required of systems that are either safety-critical or very costly by verifying that safety axioms have been implemented through a rigorous analysis of those software modules that are responsible for the safety controls of the system.

Two examples of the application of the SFTA technique illustrate that it is cost effective and helps improve the robustness of a design. SFTA techniques have been applied with success on the Canadian Nuclear Power Plant shutdown software. The software consisted of approximately

6000 lines of Pascal and Fortran code [3]. Although no errors were detected in SFTA's, the changes implemented improved the robustness of the system. The increased robustness was achieved by inserting run time assertions to verify safe operating conditions.

Another example of an application of SFTA was on the spacecraft called FIREWHEEL (NASA/ESA). This spacecraft had an Intel 8080 assembly language program of approximately 1200 lines of code that controlled flight and telemetry. The code had already been extensively tested when the SFTA techniques were applied. This analysis discovered that an unanticipated environment hazard could have resulted in the loss of the craft [2].

## B.3 Use of Software Fault Tree Analysis

Any SFTA must be preceded by a hazard analysis of the entire system. The information in the hazard analysis identifies those undesired events in the system that can cause serious consequences. It should be noted that in complex systems not all hazards can be predetermined. In this respect the technique does not claim to produce consistent results irrespective of the analyst. It is dependent on the judgment of the individual as to when to stop the process and which hazards to analyze.

The SFTA can be used at different stages of the software life cycle. The earliest stage where the technique should be used is Preliminary Design (if, at this point, the design still has excessive TBDs, then the technique is ineffective). In practice it will be used most frequently at the code level, preferably prior to integration and test.

The basic procedure in an SFTA is to assume that the hazard has occurred and then to determine its set of possible causes. The technique is useless if one starts with the overly generalized hazard "system fails". A more specific failure, such as those identified from the earlier hazard analysis, has to be the starting point for the analysis. The hazard is the root of the fault tree and its leaves are the necessary preconditions for the hazard to occur. These preconditions are listed in the fault tree and connected to the root of the tree via a logical AND or logical OR of the preconditions (see Figure B- 2:- Example of High Level Fault Tree). In turn, each one of the preconditions is expanded in the same fashion as the root fault (we identify the causes of each precondition). The expansion continues until all leaves describe events of computable probability or the event cannot be analyzed further. The analysis also stops when the precondition is a hardware malfunction that has no dependency on software.

The fault tree is expanded from the specified system level failure to the software interface level where we have identified the software outputs or lack of them that can adversely affect system operation. At this stage the analysis begins to take into account the behavior specific to the language. The language constructs can be transformed into templates using preconditions, postconditions and logical connectives. (For templates of Ada constructs, see Leveson et al. [3].) All the critical code must be traced until all conditions are identified as true or false or an input statement is reached.

The technique will be illustrated with an example using a Pascal like language [2]. The code will be analyzed for the occurrence of the variable Z being output with a value greater than 100. We should assume B, X, Z are integers.

```
While B>Xdo
begin B :=B- 1;
Z := Z + 10;
end
if Z ~ 100 then output Z;
```

In this piece of code there are assignment statements, an "if" and a "while" construct. The templates for these statements will be applied, starting from the occurrence of the event we are searching for "output Z with Z > 100". Refer to Figure B- 3 :- Example Code Fault Tree for the discussion that follows. The templates for the constructs will be drawn showing all the considerations that are required for the analysis to be complete. Some leaves of the tree are not expanded further because they are not relevant to the event or postcondition that we are analyzing. The "if" template shows that the event is triggered by the "then" clause. This follows from the condition in the "if" statement. At this point we need to determine the preconditions necessary for Z > 100 prior to the entry into the while construct.

In this example we have only two simple assignments within the "while" construct but they could be replaced by more complex expressions. The analysis would still be similar to that shown here in the example. The "while" construct would be analyzed as a unit and the expressions within the "while" would generate a more complex tree structure as previously described using the language templates to determine the preconditions. By analysis of the transformations in the "while" loop, we arrive at the conclusion that for the Z > 100 to be output, the weakest precondition at the beginning of the code was that for B > X, Z + 1 OB - 10X > 100. At this point we have identified the weakest condition necessary for this code to output Z with Z > 100. More detailed examples are provided in references [1] and [2]. Anyone interested in applying the technique should study the examples in the two references or other articles where the technique is illustrated.

The analysis that was shown in the section above determined the preconditions for the event to occur. One way to preclude a hazard from happening is to place an assertion in the code that verifies that the precondition for the hazard, as determined in the analysis, does not occur. SFTAs point out where to place assertions and the precondition to assert. If the preconditions do occur, some corrective action needs to take place to remedy the problem or, if a remedy is not possible, to mitigate the consequences.

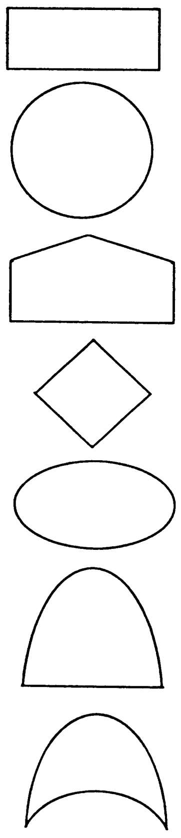
Typically a small percentage of the total software effort on projects will be spent on safety critical code. The Canadian Nuclear Power Plant safety-critical shutdown software was reviewed via the SFTA technique in three work months. The cost of this technique is insignificant considering the total amount spent on testing and verification. Full functional verification of the same software took 30 work years [3]. In cases where no problems are found, the benefits can still justify the investment. The resulting code is made more robust by the inclusion of the safety assertions and the analysis verifies that major hazardous states identified have been avoided.

Due to complexity, the figures from the example cited above (3 work months for 6K lines of code) will probably not scale up. The technique can be selectively applied to address only certain classes of faults in the case where a large body of safety-critical code requires a safety verification.

## B.4 Benefits Of Software Fault Tree Analysis

Overall, the benefits of carrying out an SFTA are well worth the small investment that is made at either the design or code stage, or at both stages. SFTAs can provide the extra assurance required of safety-critical projects. When used in conjunction with the traditional functional verification techniques, the end product is a system with safer operational characteristics than prior to the application of the SFTA technique.

**Figure B-1: - SFTA Graphical Representation Symbols**



A RECTANGLE INDICATES AN EVENT TO BE ANALYZED

A CIRCLE INDICATES A BASIC FAULT EVENT OR PRIMARY FAILURE OF A COMPONENT. IT REQUIRES NO FURTHER DEVELOPMENT

THE HOUSE IS USED FOR EVENTS WHICH NORMALLY OCCUR IN THE SYSTEM. IT REPRESENTS CONTINUED OPERATION OF THE COMPONENT.

THE DIAMOND IS USED FOR NON PRIMAL EVENTS WHICH ARE NOT DEVELOPED FURTHER FOR LACK OF INFORMATION OR INSUFFICIENT CONSEQUENCE.

THE OVAL IS USED TO INDICATE A CONDITION. IT DEFINES THE STATE OF THE SYSTEM THAT PERMITS A FAULT SEQUENCE TO OCCUR. IT MAY BE NORMAL OR RESULT FROM FAILURES.

THE AND GATE SERVES TO INDICATE THAT ALL INPUT EVENTS ARE REQUIRED IN ORDER TO CAUSE THE OUTPUT EVENT.

THE OR GATE SERVES TO INDICATE THAT ONE OR MORE OF THE INPUT EVENTS ARE REQUIRED TO PRODUCE THE GATED EVENT

**Figure B-2: - Example of High Level Fault Tree**

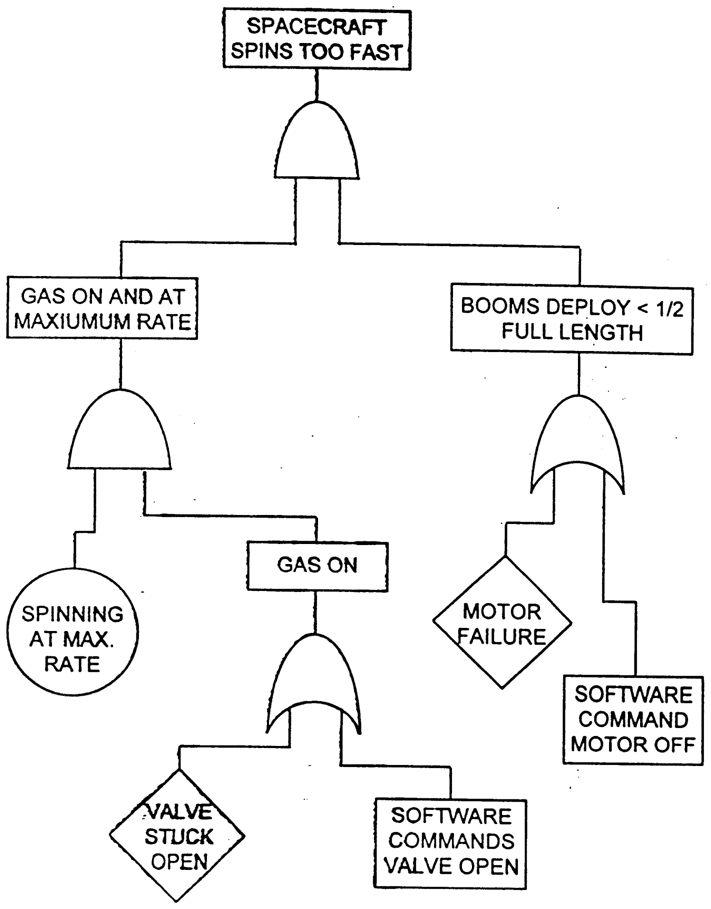
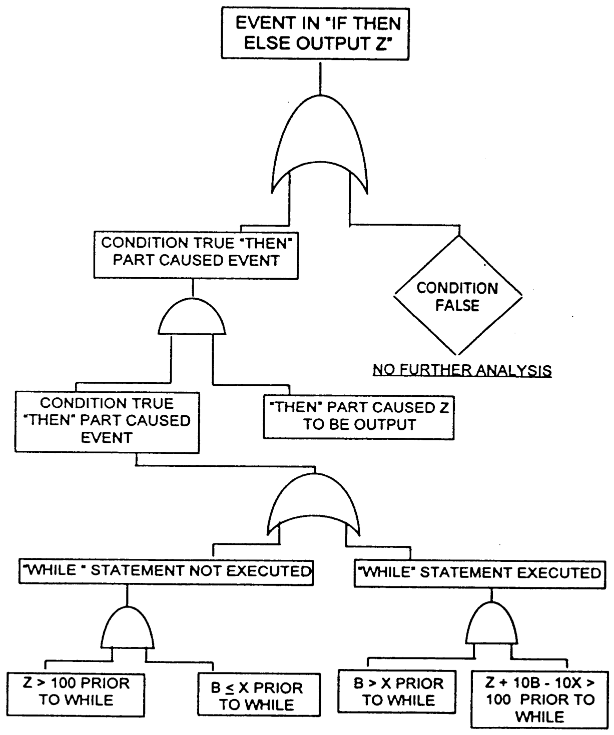

SPACECRAFT SPINS TOO FAST
GAS ON AND AT MAXIUMUM RATE
BOOMS DEPLOY < 1/2 FULL LENGTH
SPINNING AT MAX. RATE
GAS ON
MOTOR FAILURE
SOFTWARE COMMAND MOTOR OFF
VALVE STUCK OPEN
SOFTWARE COMMANDS VALVE OPEN

**Figure B-3: - Example Code Fault Tree**


EVENT IN "IF THEN ELSE OUTPUT Z"
CONDITION TRUE "THEN" PART CAUSED EVENT
CONDITION FALSE
NO FURTHER ANALYSIS
CONDITION TRUE "THEN" PART CAUSED EVENT
"THEN" PART CAUSED Z TO BE OUTPUT
"WHILE " STATEMENT NOT EXECUTED
"WHILE" STATEMENT EXECUTED
Z > 100 PRIOR TO WHILE
B ≤ X PRIOR TO WHILE
B > X PRIOR TO WHILE
Z + 10B - 10X > 100 PRIOR TO WHILE

# APPENDIX C Software Failure Modes and Effects Analysis

This section is provided to assist systems safety engineers and software developers with an introductory explanation of the Software Failure Modes and Effects Analysis technique. The information presented here is part of the NASA Safety Critical Software Analysis course.

Failure Modes and Effects Analysis (FMEA) is a bottom-up method used to find potential system problems while the project is still in the design phase. Each component in the system is examined, and all the ways it can fail are listed. Each possible failure is traced through the system to see what effects it will have, and whether it can result in a hazardous state. The likelihood of the failure is considered, as well as the severity of the system failure.

FMEA has been used by system safety and other engineering disciplines since the 1960's. The methodology has been extended to examine the software aspects of a system (SFMEA).

## *C.1 Terminology*

A **failure** is the inability of a system or component to perform its required functions within specified performance requirements. An event that makes the equipment deviate from the specified limits of usefulness or performance is also a failure. Failures can be complete, gradual, or intermittent.

A **complete** system failure is manifested as a system crash or lockup. At this juncture, the system is usually unusable in part, or in whole, and may need to be restarted as a minimum. - What precautions are needed to guard against this, if it is inevitable, then what can be done to insure the system is safe and can recover safely.

A **gradual** system failure may be manifested by decreasing system functionality. Functions may start to disappear and others follow or, the system may start to degrade (as in the speed with which functions are executed may decrease). Often resource management is a fault here, the CPU may be running out of memory or time slice availability.

**Intermittent** failures are some of the most frustrating and difficult to solve. Some of these may be cyclical or event driven or some condition periodically occurs which is unexpected and/or non-predictive. Usually an unrealized path through the software takes place under unknown conditions.

These types of failures should be kept in mind when considering failure modes (described below). Unlike most hardware failures, software faults don't usually manifest as "hard" (complete lockup of the system) type system failures. Software doesn't wear out and break. It is either functional, or already broken (but no one knows it)!

A **Failure Mode** is defined as the type of defect contributing to a failure (ASQC); the physical or functional manifestation of a failure (IEEE Std 610.12-1990). The Failure Mode is generally the manner in which a failure occurs and the degree of the failure's impact on normal required system operation. Examples of failure modes are: fracture (hardware), value of data out of limits (software), and garbled data (software).

The **Failure Effect** is the consequence(s) a failure mode has on the operation, function, or status of an item or system. Failure effects are classified as local effects (at the component), next higher level effects (portion of the system that the component resides in), and end effect (system level).

## C.2 Why do an SFMEA?

SFMEA's identify key software fault modes for data and software actions. It analyzes the effects of abnormalities on other components in the system, and on the system as a whole. This technique is used to uncover system failures from the perspective of the lowest level components. It is a "bottom-up" (or "forward") analysis, propagating problems from the lowest levels, up to a failure within the broader system.

Software Fault Tree Analysis (SFTA, *Appendix B*) is a "top down" (or "backward") approach. It identifies possible system failures and asks what could have caused them. SFTA looks backwards from the failure to the component(s) whose defects could cause or contribute to the failure.

The SFMEA asks "What is the effect if this component operates incorrectly?" Failures for the component are postulated, and then traced through the system to see what the final result will be. Not all component failures will lead to system problems. In a good defensive design, many errors will already be managed by the error-handling part of the design.

A Software FMEA takes a systems approach, analyzing the software's response to hardware failures and the effects on the hardware of anomalous software actions. Doing an FMEA is done on software can identify:

- Hidden failure modes, system interactions, and dependencies
- Unanticipated failure modes
- Unstated assumptions
- Inconsistencies between the requirements and the design

SFMEA's are not a panacea. They will not solve all of your problems! You will probably not get all of the above results, but you should be a lot closer to a clean system than if you had not done the analysis.

It's important to interact with other members of the team as you perform an SFMEA. No one person understands all components, software or hardware. Have hardware and software designers/engineers review your analysis as you are performing it. Their point of view will help uncover the hidden assumptions or clarify the thought process that led to a requirement or design element. SFMEA is not a silver bullet, but a tool to hedge your bets (reduce your risk).

## C.3 Issues with SFMEA

If SFMEA's are so wonderful, why isn't everyone doing them? The problems are the technique is:

™ Time consuming

™ Tedious

™ Manual method (for now)

- ™ Dependent on the knowledge of the analyst
- ™ Dependent on the accuracy of the documentation
- ™ Questionable benefit of incomplete failure modes list

The place to reap the greatest advantages of this technique is in requirements and design analysis. This may take some time, but it is well worth the effort in terms of the different perspectives with which you'll be able to view the project (hardware, software, operations, etc.).

The technique is considered tedious by some. However, the end result is greater and more detailed project and/or system knowledge. This is most true when used earlier (requirements and design) in the life-cycle. It is easier to use SFMEA later in the project, since components and their logical relationships are known, but at this point (i.e. detailed design and implementation) it is often too late (and expensive) to affect the requirements or design. Early in the project, lower level components are conjecture and may be wrong, but this conjecture can be used to drive out issues early. There must be balance in the approach. There is no value in trying to perform analysis on products that are not ready for examination.

The technique is dependent on how much the analyst knows and understands about the system. However, as mentioned earlier, the technique should be helpful in bringing out more information as it is being used. Include more reviewers who have diverse knowledge of the systems involved. In addition to looking at the project from different angles, the diversity of background will result in a more keen awareness of the impact of changes to all organizations.

Documentation is also very important to using this analysis technique. So, when reviewing documents, use many and different types of resources (systems and software engineers, hardware engineers, system operations personnel, etc.), so that differing perspectives have been utilized in the review process. The obvious benefit is a better product as a result of critique from numerous angles.

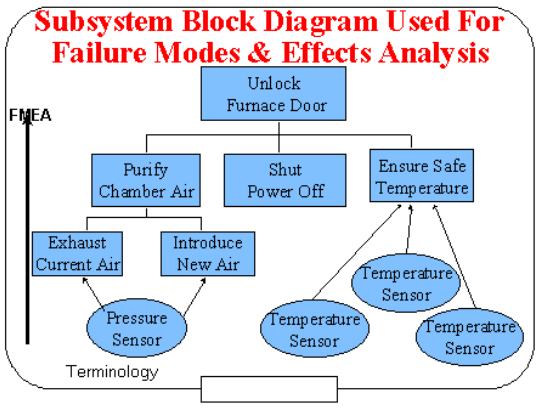
Again, don't work in a vacuum! Communication is paramount to success.

Where should you use the SFMEA technique? All of the following areas, though you should focus on the safety-critical aspects.

- Single Failure Analysis
- Multiple Failure Analysis
- Hardware/Software Interfaces
- Requirements
- Design
- Detailed Design

## C.4 The SFMEA Process

**Figure C-1**



FMEA analysis begins at the bottom (the "end" items). Figure C-1 shows a subsystem, indicating how each piece interacts with the others. Logic (and's and or's) is not included on this introductory diagram. The end items are the pressure sensor and temperature sensor. The diagram shows how the failures propagate up through the system, leading to a hazardous event.

Software FMEA's follow the same procedure used for hardware FMEA's, substituting software components for the hardware. Alternately, software could be included in the system FMEA, if the systems/reliability engineer is familiar with software or if a software engineer is included in the FMEA team. MIL-STD-1629 is a widely used FMEA procedure, and this appendix is based on it.

To perform a Software Failure Modes and Effects Analysis (SFMEA), you identify:

- Project/system components
- Ground rules, guidelines, and assumptions
- Potential functional and interface failure modes
- Each failure mode in terms of potential consequences
- Failure/fault detection methods and compensating provisions
- Corrective design or actions to eliminate or mitigate failure/fault
- Impacts of corrective changes

### C.4.1 Identify Project/system Components

Engineers must know the project, system, and purpose and keep the "big picture" in mind as they perform the analysis. A narrow perspective can prevent you from seeing interactions between components, particularly between software and hardware. Communicate with those of differing backgrounds and expertise.

In performing a FMEA, defining whatever is being worked on is the first order of business. The "whatever" can be a project, system, subsystem, "unit", or some other piece of the puzzle. Depending on where the project is in the development life-cycle (requirements, design, implementation), you will hopefully have some documents to work with. If the documentation is lacking, you will have to do some detective work. Often there is a collection of semi-formal paperwork on the requirements or design produced by the software team but not written into a formal requirements or design document. Look for a "Software Development Folder", talk with the developers, and accumulate whatever information you can. If little is on paper, you will have to interview the developers (and project management, hardware engineers, systems people, etc.) to create your own documentation.

Once you know what the system is and what it is supposed to do, it's time to start breaking down the system into bite size chunks. Break a project down into its subsystems. Break a subsystem down into its components. This process begins with a high level project diagram which consists of blocks of systems, functions, or objects. Each block in the system will then have its own diagram, showing the components that make up the block (subsystem). This is a lot of work, but you don't usually have to do the whole project! Not every subsystem will need to be detailed to its lowest level. Deciding what subsystems need further breakdown comes with experience. If in doubt, speak with the project members most familiar with the subsystem or component.

During the requirements phase, the lowest-level components may be functions or problem domains. At the preliminary (architectural) design phase, functions, Computer Software Configuration Items (CSCIs), or objects/classes may be the components. CSCIs, units, objects, instances, etc. may be used for the detailed design phase.

Take the "blocks" you've created and put them together in a diagram, using logic symbols to show interactions and relationships between components. You need to understand the system, how it works, and how the pieces relate to each other. It's important to lay out how one component may affect others, rippling up through the system to the highest level. Producing this diagram helps you, the analyst, put the information together. It also provides a "common ground" when you are discussing the system with other members of the team. They can provide feedback on the validity of your understanding of the system.

### C.4.2 Ground Rules

Before you begin the SFMEA, you need to decide what the ground rules are. There are no right or wrong rules, but you need to know ahead of time what will be considered a failure, what kinds of failures will be included, levels of fault-tolerance, and other information. Some sample ground rules are:

1. All failure modes are to be identified at the appropriate level of detail: component, subsystem, and system.
2. Each experiment mission shall be evaluated to determine the appropriate level of analysis required.
3. The propagation of failure modes across interfaces will be considered to the extent possible based on available documentation.
4. Failures or faults resulting from defective software (code) shall be analyzed to the function & object level during detailed design.

5. Failure modes induced by human error shall not be included in this FMEA.
6. The criticality categorization of a hardware item failure mode shall be made on the basis of the worst case potential failure effect.
7. Identical Items which perform the same function, in the same environment (where the only difference is location) will be documented on a worksheet only once provided that the failure mode effects are identical.
8. Containment structures such as combustion chambers and gas cylinders will be analyzed.
9. For catastrophic hazards, **dual** component failures ( items which are one-fault tolerant) are credible.
10. For **catastrophic** hazards, triple component failures (items with two-fault tolerance) are not credible.
11. For critical hazards, single component failures are credible.
12. For critical hazards, **dual** component failures are not credible
13. Release of the contents in a single containment gas bottle does not constitute a hazard of any kind provided that the gases released are pre-combustion gases.(e.g., flammability, toxicity, 02 depletion)
14. Items exempt from failure modes and effects analysis are: tubing, mounting brackets, secondary structures, electrical wiring, and electronic enclosures.

Besides the ground rules, you need to identify and document the assumptions you've made. You may not have sufficient information in some areas, such as the speed at which data is expected at an interface port of the system. If the assumption is incorrect, when it is examined it will be found to be false and the correct information will be supplied (sometimes loudly). This examination will occur when you describe what you believe to be the normal operation of the system or how the system handles faults to the other project members.

Don't let assumptions go unwritten. Each one is important. In other words, "ASSUME NOTHING" unless you write it down. Once written, it serves as a focus to be further explored and exploded.

Try to think "outside the box" – beyond the obvious. Look at the project as a whole, and then at the pieces/parts. Look at the interactions between components, look for assumptions, limitations, and inconsistencies.

**Figure C-2**

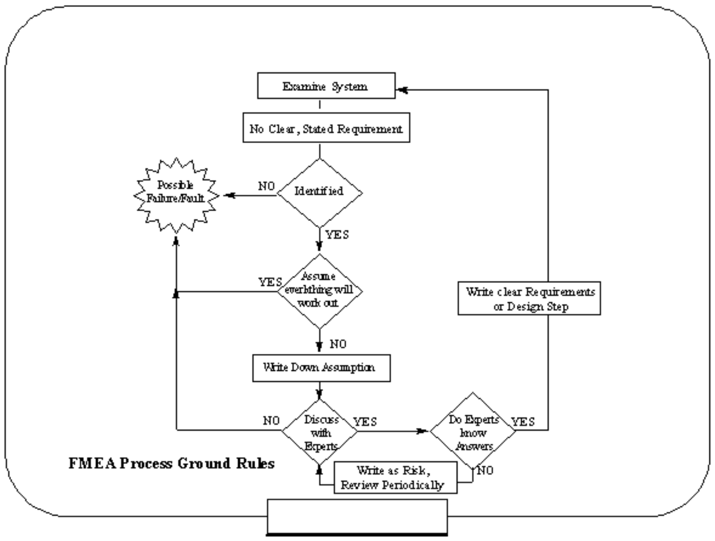

Examine System
No Clear, Stated Requirement
Identified
NO
Possible Failure/Fault
YES
Assume everything will work out
YES
NO
Write Down Assumption
Discuss with Experts
NO
YES
Do Experts know Answers
YES
NO
Write clear Requirements or Design Step
Write as Risk, Review Periodically
FMEA Process Ground Rules

Figure C-2 shows the process of recognizing your assumptions, documenting them, finding out what the reality is, and clarifying them for future reference.

### C.4.3 Identify Failures

Once you understand the system, have broken it into components, created ground rules, and documented your assumptions, it’s time to get to the fun part: identifying the possible failures. Failures can be functional (it doesn’t do what it was supposed to do), undesirable responses to bad data or failed hardware, or interface related.

Functional failures will be derived from the Preliminary Hazard Analysis (PHA) and subsequent Hazard Analyses, including subsystem HA’s. There will probably be hardware items on this list. This analysis looks at software’s relationship to hardware.

It is important to identify functions that need protecting. These functions are “must work functions” and “must not work functions”. A failure may be the compromise of one of these functions by a lower-level software unit.

There are also interfaces to be dealt with. There are more problems identified with interfaces, according to some researchers, than any other aspect of software development. Interfaces are software-to-software (function calls, interprocess communication, etc.), software-to-hardware (e.g. setting a Digital-to-Analog port to a specified voltage), hardware-to-software (e.g. software reads a temperature sensor), or hardware-to-hardware. SFMEA’s deal with all of these except the hardware-to-hardware interfaced. These are included in the system FMEA. Interfaces also (loosely) include transitions between states or modes of operation.

As you look at the system, you will find that you need to make more assumptions. Write them down. When all else fails, and there is no place to get useful information, sometimes a guess is in order. Again, write it down and go discuss it with others. The “others” should include people outside of your area of expertise. If you are a software person, go talk with safety and systems. If you are a safety specialist, talk with systems, software, and reliability experts.

### C.4.3.1 Examination of Normal Operations as Part of the System

The normal operations of the system include it performing as designed, being able to handle known problem areas, and its fault tolerance and failure response (if designed into the system). Hopefully, the system was designed to correctly and safely handle all anticipated problems. The SFMEA will find those areas where *unanticipated* problems create failures.

This step identifies how the software responds to the failures. This step validates the sufficiency, or lack thereof, of the product "to do what its supposed to do". This has the side affect of confirming the product developers' understanding of the problem. In order to understand the operation of a system it may be necessary to work and communicate with systems engineering if you are a software engineer. Systems engineering must also communicate with software engineering, and both must talk with safety and Software Assurance (SA).

The normal operation of the software as part of the system or function is described in this part of the SFMEA.

### C.4.3.2 Identify Possible Areas for Faults

Areas to examine for possible faults include:

- ❖ **Data Sampling Rate.** Data may be changing more quickly than the sampling rate allows for, or the sampling rate may be too high for the actual rate of change, clogging the system with unneeded data.
- ❖ **Data Collisions.** Examples of data collisions are: transmission by many processors at the same time across a LAN, modification of a record when it shouldn't be because of similarities, and modification of data in a table by multiple users in an unorganized manner.
- ❖ **Command Failure to Occur.** The command was not issued or not received.
- ❖ **Command out of sequence.** There may be an order to the way equipment is commanded on (to an operational state). For instance, it is wise to open dampers to the duct work going to the floors, as well as the dampers to bring in outside air before turning on the air handling units of a high rise office building.
- ❖ **Illegal Command.** Transmission problems or other causes may lead to the reception of an unrecognized command. Also, a command may be received that is illegal for the current program state.
- ❖ **Timing.** Dampers take a long time to open (especially the big ones) so, timing is critical. A time delay would be necessary keep from imploding (sucking in) the outside air dampers or possibly exploding the supply air dampers, by turning on the air handler prematurely.
- ❖ **Safe Modes.** It is sometimes necessary to put a system which may or may not have software in a mode in where everything is safe (i.e. nothing melts down or blows up). Or the software maintains itself and other systems in a hazard free mode.

- **Multiple Events or Data.** What happens when you get the data for the same element twice, within a short period of time? Do you use the first or second value?
- **The Improbable.** The engineers or software developers will tell you that something "can't happen". Try to distinguish between truly impossible or highly improbable failures, and those that are unlikely but possible. The improbable **will** happen if you don't plan for it.

These are all sorts of things that software can do to cause system or subsystem failures. Not every software fault will lead to a system failure or shutdown, and even those failures that occur may not be safety critical. There are lots more types of faults than these, but these are good start when looking for things that can go wrong.

### *C.4.3.3 Possible Failure Modes*

Identify the possible failure modes and effects in an Events Table and Data Table, included in *Section C.4.8*.

Examples of failure modes are:

**Hardware Failures/Design Flaws**

- Broken sensors lead S/W down wrong path
- No sensors or not enough sensors - don't know what H/W is doing
- Stuck valves or other actuators

**Software**

- Memory over written (insufficient buffer or processing times).
- Missing input parameters, incorrect command, incorrect outputs, out of range values, etc.
- Unexpected path taken under previously unthought of conditions.

**Operator**

- Accidental input of unknown command, or proper command at wrong time.
- Failure to issue a command at required time.
- Failure to respond to error condition within a specified time period.

**Environment**

- Gamma Radiation
- EMI
- Cat hair in hard drive
- Power fluctuations

### *C.4.3.4 Start at the Bottom*

Go back to the block diagrams you created earlier. Starting at the lowest level, look at a component and determine the effect of that component failing, in one of its failure modes, on the components in the level above it.

You may need to consider the effect of this component and all the effected components at the next higher level as well. This must be worked all of the way up the chain.

This is a long process. However, if the safety critical portions are fairly isolated in a system, then the analyst will be looking at only those parts of the system that can lead to a critical failure. This is true for the detailed design and implementation phases/versions of this analysis. For the requirements and preliminary design phases, the system is more abstract (and therefore smaller and more manageable).

### C.4.4 Identify Consequences of each Failure

The next thing to look at is the effect (consequences) of the defined faults/failures. It is also important to consider the criticality or severity of the failure/fault.

So far in the FMEA process, we've concentrated on the safety perspective. However, it's time to look at reliability as well. Like safety , reliability, looks at:

- **Severity** may be catastrophic, critical, marginal, or negligible.
- **Likelihood of occurrence** may be probable, occasional, remote or improbable.

Risk levels are defined as 1 through 5, with 1 being prohibitive (i.e. not allowed-must make requirements or design change). The critically categories include the added information of whether the component or function has redundancy or would be a single point of failure.

For each project and center there may be some variation in the ranking of severity level and risk level. This is, after all, not an exact science so much as a professional best guess (best engineering judgment).

The relationship between reliability's criticality categories and the safety risk level is shown in the following table:

| Criticality Category | Relative Safety Risk Level |
|---|---|
| 1 – A single failure point that could result in a hazardous condition, such as the loss of life or vehicle. | Levels 1 to 2 |
| 1R – Redundant components/items for which, if all fail, could result in a hazardous condition. | Levels 1 to 2 |
| 2 – A single failure point that could result in a degree of mission failure (the loss of experiment data) | Levels 2 to 3 |
| 2R – Redundant items, all of which if failed could result in a degree of mission failure (the loss of experiment data). | Levels 2 to 3 |
| 3 – All others. | Levels 4 and 5 |

### C.4.5 Detection and Compensation

At this step, you need to identify the methods used by the system to detect a hazardous condition, and provisions in the system that can compensate for the condition.

For each failure mode, a fault/failure detection method should be identified. A failure detection mechanism is a method by which a failure can be discovered by an operator under normal system operation or by some diagnostic. Failure detection in hardware is via sensing devices or instruments. In software this could be done by error detection software on transmitted signals, data or messages, memory checks, initial conditions, etc.

For each failure mode, a compensating provision should be identified, or the risk accepted if it is not a hazardous failure. Compensating provisions are either design provisions or operator actions which circumvent or mitigate. This step is required to record the true behavior of the item in the presence of an internal malfunction of failure. A design provision could be a redundant item or a reduced function that allows continued safe operation. An operator action could be the notification at an operator console to shut down the system in an orderly manner.

An example: The failure is the loss of data because of a power loss (hardware fault), or because other data overwrote it (a software fault) .

Detection: A critical source and CPU may be backed up by a UPS (uninterruptible power supply) or maybe not. Detect that power was lost and the system is now on this backup source. Mark data at time x as not reliable. This would be one detection scheme.

Compensation for the occurrence of this failure: Is there another source for that data.? Can it be re-read? Or just marked as suspect or thrown out and wait for next normal data overwrite it? What of having a UPS, battery backup, redundant power supply? Of course these are all hardware answers. Can software detect if the data is possibly suspect and tag it or toss it, wait for new input, request for new input, get data from alternate sources, calculate from previous data (trend) etc.?

What if input data comes in faster than expected and was overwriting pervious data before it was processed. How would this system know? What could be done about it? For example, a software system normally receives data input cyclically from 40 sources, then due to partial failures or maintenance mode, now only 20 sources are in cycles and the token is passed 2 times faster. Can buffers handle the increased data rate?

### C.4.6 Design Changes

After a critical hazard has been identified, the project needs to

™ Identify corrective actions

™ Identify changes to the design

™ Verify the changes

™ Track all changes to closure

After a critical hazard has been identified it is usually eliminated or mitigated. The result of either of these two actions is a corrective action. This corrective action may be via documented new requirements, design, process, procedure, etc.  Once implemented, it must be analyzed and verified to correct the failure or hazard.

It is important to look at the new design, once the change is made, to verify that no new hazards have been created.

### C.4.7 Impacts of Corrective Changes

A corrective action will have impact.  Impacts can be to the schedule, design, functionality, performances, process, etc. If the corrective action results in a change to the design of the software, then some segment of that software will be impacted. Even if the corrective action is to modify the way an operator uses the system there is impact.

You need to go back and analyze the impact of the changes to the system or operating procedures to be sure that they (singularly or jointly) don’t have an adverse effect and do not create a new failure mode for a safety critical function or component.

Often fixes introduce more errors and there must be a set process to insure this does not occur in safety critical systems. Ensure that verification procedures cover the effected areas.

### C.4.8 Example forms

This worksheet is used to gather relevant information on the system. It is also a great place to put the data developed during the analysis. The ID number can be a drawing number, work break down structure number, CSCI identification, or other identification value.

**FMEA Worksheet**

| **ITEM Description** | **ID #** | **SUBSYSTEM COMPONENT** | **LOCAL FAILURE MODE/EFFECT** | **SYSTEM EFFECT** | **CRIT** |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

Once elements of the system are identified, list them in this worksheet and identify their functions.

**COMPONENTS**

| ITEM DESCRIPTION | ITEM ID | FUNCTION | FAILURE MODE | LOCAL EFFECT | SYSTEM EFFECT | DETECTABILITY | CRIT |
|---|---|---|---|---|---|---|---|
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

For a Software FMEA, the **Data Table** is used to list the effects of **bad data** on the performance of the system or process being analyzed. A **Data Item** can be an input, output, or information stored, acted on, passed, received, or manipulated by the software. The **Data Fault Type** is the manner in which a flaw is manifested (**bad data**), including data that is out of range, missing, out of sequence, overwritten, or wrong.

**SFMEA DATA TABLE**

| Mode | Data Item | Data Fault Type | Description | Effect (local and system) | Crit |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

The **Events Table** is used to list the effects of an event being performed. The **Event Item** is the occurrence of some action, either within the software or performed on hardware or other software. An event can be an expected and correct, expected but incorrect, unexpected and incorrect, or unexpected but correct action. **Event Fault Types** can occur locally (with a module) or on the system as a whole. Types can include halt (abnormal termination), omission (failure of the event to occur), incorrect logic/event, or timing/order (wrong time or out of sequence).

**SFMEA EVENTS TABLE**

| Mode | Event Item | Event Fault Type | Description | Effect (local and system) | Crit |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

# APPENDIX D Requirements State Machines

## *D.1 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. The transition from one state to another is induced by a trigger or input that is labeled on the transition. Generally an output is produced by the state machine [38].

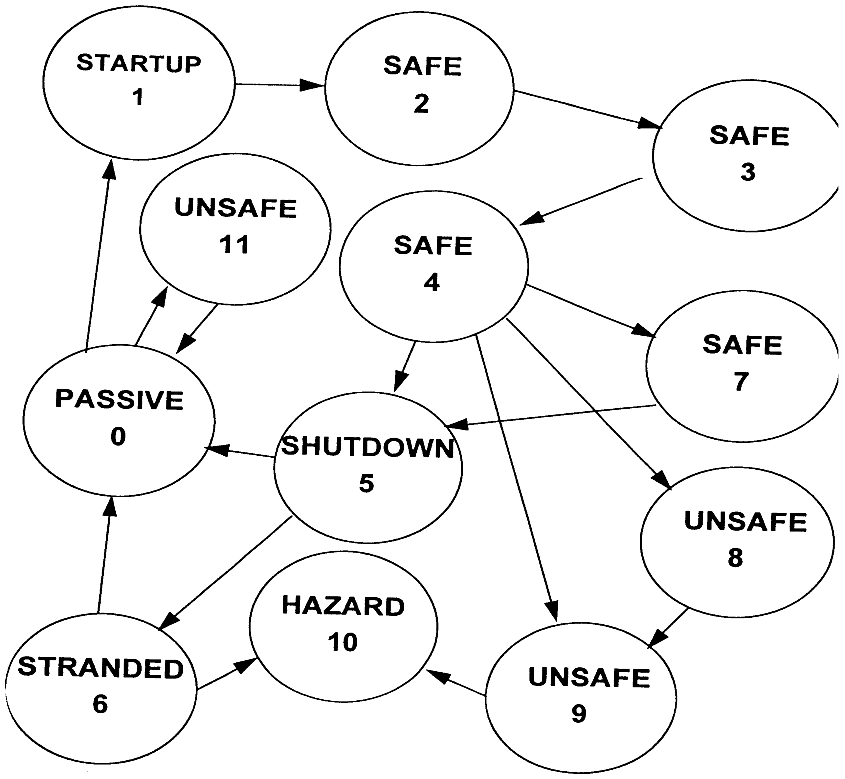
The state machine models should be built to abstract different levels of hierarchy. The models are partitioned in a manner that is based on considerations of size and logical cohesiveness. An uppermost level model should contain at most 15 to 20 states; this limit is based on the practical consideration 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 [39]. 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 (see *Figure 5-4 Example of State Transition Diagram*). 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 time-out may be specified and a state transition to an unsafe or passive state may be defined.

**Figure D-1 Example of State Transition Diagram**



The shutdown state represents the final state of the system. This state is the only path to the passive state once the state machine has begun operation. Every system must have at least one shutdown state. A time-out may be specified if the system fails to close down. If a timeout occurs, a transition to an unsafe or stranded state would be the outcome. Transition to the shutdown state does not guarantee the safety of the system. Requirements that stipulate safety properties for the shutdown state are necessary to insure that hazards do not occur while the system is being shutdown.

A safe state represents the normal operation of the system. A safe state may loop on itself for many cycles. Transitions to other safe states is a common occurrence. When the system is to be shutdown, it is expected to transition from a safe state to the shutdown state without passing through an unsafe state. A system may have zero or more safe states by definition. A safe state also has the property that the risk of an accident associated with that state is acceptable (i.e., very low).

Unsafe states are the precursors to accidents. As such, they represent either a malfunction of the system, as when a component has failed, or the system displays unexpected and undesired behavior. An unsafe state has an unacceptable, quantified level of risk associated with it from a system viewpoint. The system is still in a controllable state but the risk of transition to the hazard state has increased. Recovery may be achieved through an appropriate control action that leads either to an unsafe state of lesser risk or, ideally, to a safe state. A vital consideration when analyzing a path back to a safe state is the time required for the transitions to occur before an accident occurs. A system may have zero or more unsafe states.

The hazard state signals that control of the system has been lost. In this situation the loss of the system is highly probable and there is no path to recovery. The hazard state should take action where possible to contain the extent of damage.

The stranded state represents the situation, where during the course of a shutdown operation, the system has lost track of state information and cannot determine a course of action. This state has a high potential to transition to an unsafe state after a specified time depending upon what system is modeled or possibly upon environmental conditions. The only recovery from this state is a power-on restart.

## D.2 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.

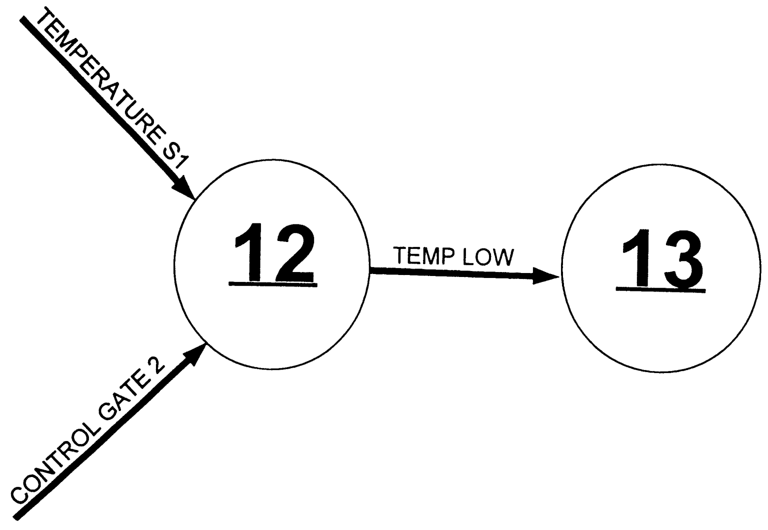
## D.3 Input/Output Variables

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 [40].

## D.4 State Attributes

The state attributes of the RSM are to be labeled according to the scheme in *Figure 5-5 Example RSM and Signals*.

**Figure D-2 Example RSM and Signals**



State 12 outputs a temperature low signal to State 13. Control Gate 2 is a trigger that controls the flow of temperature S1 into State 12.

This diagram can be expressed as:

Temp Low := Temperature S1 ≤ Threshold when Control Gate 2 is high.

## D.5 Trigger Predicates

A necessary, but not a sufficient condition for a system to be called robust, is that there must always be a way for the RSM to leave every state of the system. This leads us to define two statements about RSMs:

1) Every state in the RSM has a defined behavior (transition) for every possible input.
2) One or more input predicates, out of all possible input predicates, must be able to trigger a transition out of any state.

In case there is no input within a specified time, every state must have a defined transition, such as a time-out, that triggers an action. The state machine may also express what actions are taken if the input data is out of range. Low level functions, such as exception handling, may be features that are required for an implementation.

A relatively simple method that provides an elementary correctness check is for range verification of input data. The computational cost in most cases will probably not be significant. While range checking does not provide a guarantee of correctness, it is the first line of defense against processing bad data. Obviously, if the input data is out of range, we have identified either a bad sensor or a bad data communication medium.*

The RSM technique has limitations when analyzing fault tolerant systems that contain two or more independent lanes. In redundant systems the use of threshold logic may generate another class of safety problems. The area where problems may arise is threshold logic used to validate inputs coming from different sensors. Typically the value read from the different sensors will differ by a certain percentage. Sensors are calibrated to minimize this difference, but a check must be made to verify that neither of the following situations occur: 1) a threshold may trigger one lane of the system and not the other if a value below the threshold is contrasted with a value above the threshold from the other lane; and 2) the input as processed by the control law will generate quantitatively and qualitatively different control actions. This effect can be avoided if a vote is taken at the source of the data before transmitting potentially confusing data. In the case of fully redundant, dual lane systems, each system may determine that the other is in error when in reality there is no hardware or software error. A high level RSM will not show this explicitly but it is an issue that needs to be considered in the design before any prototyping, or worse yet, coding takes place [41].

Timing problems are common causes of failures of real-time systems. Timing problems usually happen because either timing is poorly specified or race conditions that were not thought possible occur and cause an unwanted event to interrupt a desired sequence. All real-time data should have upper and lower bounds in time. Race conditions occur when the logic of a system has not taken into account the generation of an event ahead of the intended time. This type of error occurs when events that should be synchronized or sequenced are allowed to proceed in parallel. This discussion will not address the obvious case of an error in the sequence logic.

*A third possibility may also exist: the data may truly be valid, but the understanding of the system or environment state is incomplete and data having values outside of the expected range is regarded as invalid (e.g. data on ozone loss in the atmosphere above Antarctica was regarded as invalid until ground based observations confirmed the situation).

The ability to handle inputs will be called capacity and the ability to handle diverse types of input will be called load. A real-time system must have specifications of minimum and maximum capacity and load. Robustness requires the ability of the system to detect a malfunction when the capacity limits have been violated. Capacity limits are often tied to interrupts where hardware and software analyses are necessary to determine if the system can handle the workload (e.g., CPU execution time, memory availability, etc.). Load involves multiple input types and is a more comprehensive measure than capacity. Criteria for the system or process load limits must be specified. For a system to be robust, a minimum load specification needs to be specified, as well as a maximum (assuming that a real-time process control system has inputs of a certain period). The capacity and load constraints as developed for the RSM will help serve as a guide for designing the architecture of the system and subsequently in the final system implementation. These performance requirements have safety implications. The ability of the system to handle periodic capacity and load requirements is a fundamental safety property. If a system cannot handle the work load then the safety of the system is at risk because process control is not performed in a timely fashion.

## D.6 Output Predicates

The details of when an output is valid may not be known at the time the RSM is generated but these constraints should be documented somewhere in the RSM to serve as a guideline for the implementer. In a similar fashion to inputs, outputs must have their value, and upper and lower timing bounds specified. Output capacity is limited by the ability of the actuator to respond. Compatibility must exist between the frequency of reaction to input and the capacity of the output mechanism to respond. This requires that a timing analysis be performed to be certain that potential worst case input and output rate speeds can be adequately handled by both software and hardware. For output data to be valid the input data must be from a valid time range. Control decisions must be based on data from the current state of the system, not on stale data. In the computation of the output, the delay in producing the output must not exceed the permissible latency. An example of an incorrect output timing problem occurred on the F-18 fighter plane. A wing mounted missile failed to separate from the launcher after ignition because a computer program signaled the missile retaining mechanism to close before the rocket had built up enough thrust to clear the missile from the wing. The aircraft went violently out of control, but the missile fuel was eventually expended and the pilot was able to bring the plane under control before a crash occurred [42].

## D.7 Degraded Mode Operation

When a system cannot meet its work load requirements in the allotted time or unanticipated error processing has consumed processor resources and insufficient time is available for normal processing, the system must degrade in a graceful manner. Responses to graceful degradation include:

- Masking of nonessential interrupts
- Logging and generation of warning messages
- Reduction of processing load (execute only core functionality)
- Error handling
- Reduction of accuracy and/or response time
- Signals to external world to slow down inputs
- Trace of machine state to facilitate post event analysis

Which of the above measures get implemented depends on the application and its specific requirements.

Where there is load shedding, a degraded mode RSM will exist that exhibits properties that in all likelihood are different from the original RSM. The same analysis that is performed for the RSM of the fully operational system should be done on the degraded mode RSM.

Special care must be taken in the implementation of performance degradation that reduces functionality and/or accuracy. A situation can arise where, because of the transition to a state machine with different properties (and therefore, the control laws of the original RSM will be affected by a reduction in accuracy or frequency of the inputs), the outputs may not transition smoothly. In systems where operator intervention is an integral part of operation, this jolt may confuse the operator and contribute to further degradation because of operator inability to predict behavior. In principle, where response time limits can be met, predictability is preferable to abrupt change.

In order to recover from degraded mode operation there needs to be a specification of the conditions required to return to normal operations. These conditions must be specific enough to avoid having the system continuously oscillate back and forth between normal and degraded mode. In practice, a minimum delay and a check of the cause of the anomaly can achieve this.

## *D.8 Feedback Loop Analysis*

Process control models provide feedback to the controller to notify changes in state caused by manipulated variables or internal disturbances. In this manner the system can adjust its behavior to the environment. An RSM can be used to verify if feedback information is used and what signals are employed. If feedback is absent then either the design is incorrect or the requirements are faulty. The design of the system needs to incorporate a mechanism to detect the situation where a change in the input should trigger a response from the system and the response is either too slow, too fast or unexpected. For example, when a command is given to turn on a heater, a resulting temperature rise curve would be expected to follow a theoretical model within certain tolerances. If the process does not respond within a certain period of time then it can be assumed that something is wrong and the software must take an appropriate action. At a minimum, this action should be the logging of the abnormality for future analysis. The simplest, most inexpensive check for a servo loop is to verify if the reference position is different from the actual position. If the difference is non-negligible, some form of control action must be taken. If the actual position does not vary in the presence of a command to act, then it can be concluded that there is a fault in the system. RSMs can be used to help design the control process and to verify that all feedback loops are closed and that they generate the appropriate control action.

## *D.9 Transition Characteristics*

Requirements may also involve specifications regarding transitions between states. A system may or may not possess certain properties, while some other properties are mandatory. All safe states must be reachable from the initial state. Violation of this principle leads to a contradiction of requirements or a superfluous state. No safe state should ever transition, as a result of a computer control action, to an unsafe state. In principle, an automated (i.e., computer controlled) system should never transition to a hazardous state unless a failure has occurred. In general, if operator action is considered (such as the issuing of a command), the previously stated requirement may be impossible to accomplish given the requirements of certain systems. In this latter situation, the transition into and out of the unsafe state should be done in a manner that takes the least amount of time and the system eventually reverts back to a safe state.

Once the system is in an unsafe state, either because of error conditions or unexpected input, the system may transition to another unsafe state that represents a lower risk than the previous state. If it is not possible to redesign the system so that all transitions from a hazardous state eventually end in a safe state, then the approach must be to design the transitions to the lowest possible risk, given the environment. Not all RSM diagrams will be able to achieve an intrinsically safe machine, that is, one that does not have a hazardous state. The modeling process's main virtue lies in the fact that, through analysis of the RSM, faults may be uncovered early in the life cycle. The objective and challenge is to design a system that poses a tolerable level of risk.

The design of a robust system requires that, for all unsafe states, all soft and hard failure modes be eliminated. A soft failure mode occurs when an input is required in at least one state through a chain of states to produce an output and that the loss of the ability to receive that input could potentially inhibit the software. A hard failure mode is analogous to a soft failure except that the input is required for all states in the chain and the loss of the input will inhibit the output.

If a system allows for reversible commands, then it must check that, for every transition into a state caused by the command, it can transition back to the previous state via a reverse command. While in that state, an input sequence must be able to trigger the deactivation of the command. In a similar fashion, if an alarm indicates a warning and the trigger conditions are no longer true, then the alert should also cease (if appropriate operator acknowledgment action was performed when required). State transitions do not always have to be between different states. Self loops are permissible, but eventually every real-time system must initiate a different function and exit from the self loop. Watchdog timers may be used to catch timeouts for self loops. The RSM technique helps a designer by graphically representing these constraints and assisting in specifying implementation level detail.

## *D.10 Conclusions*

The RSM techniques described above can be used to provide analysis procedures to help find errors and omissions. Incorporating the RSM analysis into the development cycle is an important step towards a design that meets or exceeds safety requirements. Practically all the current safety oriented methodologies rely on the quality of the analyst(s) for results and the techniques mentioned above are a first attempt at formalizing a system's safety properties.

The RSM technique does not claim to guarantee the design of a 100% safe system. Inevitably some faults (primarily faults of omission) will not be caught, but the value of this methodology is in the fact that many faults can be made evident at an early stage, if the right mix of experienced people are involved in the analysis. Complexity of current software and hardware has caused a nonlinear increase in design faults due to human error. For this reason and because testing does not prove the absence of faults, it is recommended that the RSM modeling techniques be employed as early as possible in the system life cycle. The RSM methodology, if applied with system safety considerations, is a valuable step towards a partial proof to show the effects and consequences of faults on the system. If the RSM model is robust and the design can be shown to have followed the criteria in the previous sections, then a significant milestone will have been completed that demonstrates that the system is ready to proceed to the next phase in the life-cycle and developers will have a high level model that satisfies a core set of requirements.

From an overall systems perspective, the RSM model is used to provide a high level view of the actual system, and further refinements of the states can give insight into implementation detail. This model is then checked against the rules formulated in the previous sections. Deviation from the rules involves additional risk and, as such, this additional risk should be evaluated and documented. This process of documentation is necessary for a post project analysis to confirm the success of the system or to analyze the cause of any failures.

The technique of using RSMs to explore properties of safety critical systems is a highly recommended practice that development teams should follow. Verification of the safety properties of the RSM should be performed as a team effort between software developers, systems safety and software quality assurance. If the RSM analysis or any equivalent technique has not been performed for the design of a complex system, then that project is running the risk that major design constraints will be put aside until late in the development cycle and will cause a significant cost impact.

## APPENDIX E

### E.1 Checklists for Off-the-Shelf (OTS) Items

| Item to consider | Answer or Comment |
|---|---|
| **Does the OTS software fill the need *in this system*?** Is its operational context compatible with the system under development? Consider not only the similarities between the system(s) the OTS was designed for and the current system, but also the differences. Look carefully at how those differences affect operation of the OTS software. | |
| **How stable is the OTS product?** Are bug-fixes or upgrades released so often that the product is in a constant state of flux? | |
| **How responsive is the vendor to bug-fixes?** Does the vendor inform you when a bug-fix patch or new version is available? | |
| **How compatible are upgrades to the software?** Has the API changed significantly between upgrades in the past? Will your interface to the OTS software still work, even after an upgrade? | |
| **How "cutting edge" is the software technology?** OTS software is often market driven, and may be released with bugs (known and unknown) in order to meet an imposed deadline or to beat the competition to market. | |
| Conversely, **is the software so well known that it is assumed to be error free and correct?** Think about operating systems and language libraries. In a safety critical system, you do not want to *assume* there are no errors in the software. | |

| Item to consider | Answer or Comment |
|---|---|
| **What is the user base of the software?** If it is a general use library, with thousands of users, you can expect that most bugs and errors will be found and reported to the vendor. Make sure the vendor keeps this information, and provides it to the users! Small software programs will have less of a “shake down” and *may* have more errors remaining. | |
| **What level of documentation is provided with the software?** Is there more information than just a user’s manual? Can more information be obtained from the vendor (free or for a reasonable price)? | |
| **Is source code included**, or available for purchase at a reasonable price? Will support still be provided if the source code is purchased or if the software is slightly modified? | |
| **Can you communicate with those who developed the software**, if serious questions arise? Is the technical support available, adequate, and reachable? Will the vendor talk with you if you modify the product? | |
| **Will the vendor support older versions of the software**, if you choose not to upgrade? Many vendors will only support the newest version, or perhaps one or two previous versions. | |
| **Is there a well-defined API (Application Program Interface)**, ICD (interface control document), or similar documentation that details how the user interacts with the software? Are there “undocumented” API functions? | |
| **What are the error codes returned by the software?** How can it fail (return error code, throw an exception, etc.)? Do the functions check input variables for proper range, or it is the responsibility of the user to implement? | |

| Item to consider | Answer or Comment |
|---|---|
| **Can you obtain information on the internals of the software**, such as the complexity of the various software modules or the interfaces between the modules? This information may be needed, depending on what analyses need to be performed on the OTS software. | |
| **Can you get information about the software development process** used to create the software? Was it developed using an accepted standard (IEEE 12207, for example)? What was the size of the developer team? | |
| **What types of testing was the software subjected to?** How thorough was the testing? Can you get copies of any test reports? | |
| **Are there any known defects in the software?** Are there any unresolved problems with the software, especially if the problems were in systems similar to yours? Look at product support groups, newsgroups, and web sites for problems unreported by the vendor. However, also keep in mind the source of the information found on the web – some is excellent and documented, other information is spurious and incorrect. | |
| **Were there any analyses performed on the software**, in particular any of the analyses described in section 5? Formal inspections or reviews of the code? | |
| **How compatible is the software with your system?** Will you have to write extensive glueware to interface it with your code? Are there any issues with integrating the software, such as linker incompatibility, protocol inconsistencies, or timing issues? | |

| Item to consider | Answer or Comment |
|---|---|
| **Does the software provide all the functionality required?** How easy is it to add any new functionality to the system, when the OTS software is integrated? Will the OTS software provide enough functionality to make it cost-effective? | |
| **Does the OTS-to-system interface require any modification?** For example, does the OTS produce output in the protocol used by the system, or will glueware need to be written to convert from the OTS to the system protocol? | |
| **Does the software provide extra functionality?** Can you "turn off" any of the functionality? If you have the source code, can you recompile with defined switches or stubs to remove the extra functionality? How much code space (disk, memory, etc.) does the extra software take up? What happens to the system if an unneeded function is accidentally invoked? | |

Lessons learned from earlier projects using OTS software are useful. The following checklist can be used to reduce the risk of using OTS software:

| No. | Items To Be Considered | Does It Apply? (yes/no) | Planned Action |
|---|---|---|---|
| **1*** | **Has the vendor's facilities and processes been audited?** Allow an audit of their facility and processes. If for any reason an audit cannot be conducted then the OTS software is considered an unmitigated significant hazard, and as such, the OTS software may be inappropriate for the intended device. | | |
| **2*** | **Are the verification and validation activities for the OTS appropriate?** Demonstrate that the verification and validation activities performed for the OTS software are appropriate and sufficient to fulfill the safety and effectiveness requirements for the device. | | |
| **3*** | **Can the project maintain the OTS independent of vendor support?** Ensure that the project can maintain the OTS software even if the original developer ceases support. | | |
| 4 | **Does software contain interfaces, firewalls, wrappers, etc.?** Consider interfaces, firewalls, wrappers and glue early in the process. When creating wrappers avoid dependency on internal product interfaces and functionality or isolate the dependencies. | | |
| 4 | **Does software provide diagnostics?** Look for built-in diagnostics and error handling. | | |
| 5 | **Any key products influencing choices?** Identify key products (or strategies or standards) that can influence other choices before product evaluation. | | |
| 6 | **Has the software vendor been used before?** Employ any past experience with vendor/product. Ask for information from other projects. Use databases of information, keeping in mind that the behavior of a product can change depending on how it is used. | | |
| 7 | **Is this the initial version?** Do not buy a version 1.0. | | |
| 8 | **Have competitors been researched?** Ask competitors of the products about the other products. | | |
| 9 | **Is the source code available?** Consider buying the source code so you can perform your own testing. Note that this is expensive and will usually require waiving technical support and/or the warranty. | | |
| 10 | **Are industry standard interfaces available?** Ensure the product uses industry standard interfaces. | | |
| 11 | **Has product research been thorough?** Base product selection on analysis of the facts. | | |
| 12 | **Is the validation for the OTS driver software package available?** Include the validation process for the OTS driver software package as part of the system interface validation process. | | |

| No. | Items To Be Considered | Does It Apply? (yes/no) | Planned Action |
|---|---|---|---|
| | This includes the verification of the data values in both directions for the data signals; various mode settings for control signals in both directions (if applicable); and the input/output interrupt and timing functions of the driver with the CPU and operating system. | | |
| 13 | **Are there features that will not be used?** Determine how to handle unused features. | | |
| 14 | **Have tools for automatic code generation been independently validated?** Determine whether tools for automatic code generation have been independently validated.  OTS tool selection should follow the same process as component selection. | | |
| 15 | **Can previous configurations be recovered?** Reevaluate each version and ensure that the previous configuration can be restored. | | |
| 16 | **Will a processor require a recompile?** Perform a complete and comprehensive retest of the system replacing a processor that requires a recompile. | | |
| 17 | **Has a safety impact assessment been performed?** Perform a safety impact assessment when new or modified OTS components are placed in a baselined system. Document hazards in a Failure Modes and Effects Analysis (FMEA) table.  Ensure there is traceability between the hazard reports, the design requirements, and the test reports. Analysis should include the review of known problem reports, user manuals, specifications, patches, literature and internet searches for other user's experience with this OTS Software. | | |
| 18 | **Will the OTS tools affect safety?** Keep in mind the tool's purpose when selecting OTS tools. Determine whether the results are easy to verify and whether the results of the tool's use will influence decisions that affect safety. | | |
| 19 | **Is the OTS being used for the proper application?** Use OTS products for the purpose for which they were created. | | |
| 20 | **Is there compatibility between the OTS hardware and software?** Realize that not all OTS hardware can run all OTS software. | | |
| 21 | **Does the vendor have ISO certification?** Determine whether the vendor is ISO certified or has been awarded a SEI rating of 3 or higher.  This provides confidence that their development process is adequate. | | |
| 22 | **Does the vendor receive quality products from their suppliers?** Ensure that vendors are aware that they are responsible for the product quality from their contractors and subcontractors. | | |

*** A PROJECT WITH LIFE THREATENING HAZARDS MUST DO THESE ITEMS**

## E.2 Generic Software Safety Requirements From MSFC

| REQUIREMENT TO BE MET | APPLICABILITY Yes/No/Partial | ACTION Accept/Work |
|---|---|---|
| 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 to the crew, ground, or controlling executive. There shall be no necessary response from crew or ground operators to proceed with the recovery action. | | |
| The FDIR switch over 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 the crew, ground, or controlling executive. | | |
| Software that executes hazardous commands shall notify the initiating crew, ground operator, or controlling executive 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 crew, ground operators, or the controlling executive. | | |
| Software shall make available status of all software controllable inhibits to the crew, ground operators, or the controlling executive. | | |
| Software shall accept and process crew, ground operator, or controlling executive 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 inhibit associated with hazardous commands. | | |
| Software shall make available current status on software inhibits associated with hazardous commands to the crew, ground operators, or controlling executive. | | |

| REQUIREMENT TO BE MET | APPLICABILITY<br>Yes/No/Partial | ACTION<br>Accept/Work |
|---|---|---|
| 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 crew, ground operators, or the controlling executive. | | |
| Software shall provide for crew/ground forced execution of any automatic safing, isolation, or switch over functions. | | |
| Software shall provide for crew/ground forced termination of any automatic safing, isolation, or switch over functions. | | |
| Software shall provide procession for crew/ground commands in return to the previous mode or configuration of any automatic safing, isolation, or switch over function. | | |
| Software shall provide for crew/ground forced override of any automatic safing, isolation, or switch over functions. | | |
| Software shall provide fault containment mechanisms to prevent error propagation across replaceable unit interfaces. | | |
| Hazardous payloads shall provide failure status and data to core software systems. Core software systems shall process hazardous payload status and data to provide status monitoring and failure annunciation. | | |
| 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. | | |

| REQUIREMENT TO BE MET | APPLICABILITY<br>Yes/No/Partial | ACTION<br>Accept/Work |
|---|---|---|
| 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 crew intervention to begin or complete). | | |
| The software shall notify crew, ground, or the controlling executive 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. | | |

## E.3 Design for Safety Checklist

From a  paper given at a talk to the Forth Interest Group (UK) in London during May 1992.  Paul E. Bennet

1. Keep the design simple and highly modular. Modularity aids in the isolation of systematic failure modes.
2. Minimize common failure modes. The calculation time for failure probabilities can be extended as by the cube of common mode entries in a fault tree.
3. Identify safe states early in the design. Have these fully checked and verified for completeness and correctness.
4. Ensure that failures of dynamic system activities result in the system achieving a known and clearly identified safe state within a specified time limit.
5. Specify system interfaces clearly and thoroughly. Include, as part of the documentation, the required action or actions should the interface fail.
6. Diagrams convey the most meaning. They can often achieve more than words alone and should be used when presenting design ideas to the customer.
7. Design all systems using the same methodologies framework wherever possible. A well practiced craft helps minimize errors.

## E.4 Checklist of generic (language independent) programming practices

Taken from nuclear standard, appendix B (see http://www.sohar.com/J1030/appb.htm), "Review Guidelines on Software languages for Use in Nuclear Power Plant Safety Systems" Final Report-NUREG/CR-6463

- ✓ Minimize use of dynamic memory. Using dynamic memory can lead to memory leaks. To mitigate the problem, release allocated memory as soon as possible. Also track the allocations and deallocations closely.
- ✓ Minimize memory paging and swapping. In a real-time system, this can cause significant delays in response time.
- ✓ Avoid *goto*'s. *Goto*'s make execution time behavior difficult to fully predict as well as introducing uncertainty into the control flow. When used, clearly document the control flow, the justification for using *goto*'s, and thoroughly test them.
- ✓ Minimize control flow complexity. Excessive complexity makes it difficult to predict the program flow and impedes review and maintenance. Project guidelines or coding standards should set specific limits on nesting levels.
- ✓ Initialize variables before use! Using uninitialized variables can cause anomalous behavior. Using uninitialized pointers can lead to exceptions or core dumps.
- ✓ In larger routines, use single entry and exit points in subprograms. Multiple entry or exit points introduce control flow uncertainties. In small subprograms, multiple exit points may actually make the routine more readable, and should be allowed. Document any secondary entry and exit points.
- ✓ Minimize interface ambiguities. Interface errors account for many design and coding errors. Look at the interfaces to hardware, other software, and to human operators.
- ✓ Use data typing. If the language does not enforce it, include it in the coding standards and look for it during formal inspections.
- ✓ Provide adequate precision and accuracy in calculations, especially within safety critical modules.
- ✓ Use parentheses to specify precedence order, rather than relying on the order inherent in the language. Assumptions about precedence often lead to errors, and the source code can be misinterpreted when reviewing it.
- ✓ Avoid functions or procedures with side effects. Side effects can lead to unplanned dependencies, and ultimately to bugs.
- ✓ Separate assignments from evaluation statements. Mixing them can cause unanticipated side effects. An example of a mixed assignment/evaluation statement is:

```
        y = toupper(x=getchar());        // x=getchar() should be on separate line
```

- ✓ Instrumentation (debugging statements, etc) should be highly visible. If left in the run-time system, it should be minimized to avoid timing perturbations. Visibility allows the "real code" to be obvious when the source code is reviewed, and it makes it easier to be sure all instrumentation is removed for the run-time system.
- ✓ Minimize dynamic binding. Dynamic binding is a necessary part of polymorphism. When used, it should be justified. Keep in mind that it causes unpredictability in name/class association and reduces run-time predictability.

- ✓ Be careful when using operator overloading. While it can help achieve uniformity across different data types (which is good), it can also confuse the reader (and programmers) if used in a non-intuitive way.
- ✓ Use tasking with care. While it adds many good features to programs (e.g. splitting the work into logical units, each of which can be tested independently), it can also lead to timing uncertainties, sequence of execution uncertainties, vulnerability to race conditions, and deadlocks.
- ✓ Minimize the use of interrupt driven processing. Interrupts lead to non-deterministic response times, which is very important in real-time systems. The best way to handle this is to have the interrupt processing do the bare minimum, and return to primary program control as soon as possible. Check how the operating system does time slicing (usually using clock interrupts), and what overhead or problems may be inherent in their implementation.
- ✓ Handle exceptions locally, when possible. Local exception handling helps isolate problems more easily and more accurately. If it is not possible to do this, then thorough testing and analysis to verify the software's behavior during exception testing is recommended.
- ✓ Check input data validity. Checking reduces the probability of incorrect results, which could lead to further errors or even system crashes. If the input can be "trusted", then checking is not necessary.
- ✓ Check the output data validity, if downstream input checking is not performed. This reduces incorrect results, which can have mild to major effects on the software.
- ✓ Control the use of built-in functions through project specific guidelines. Built-in functions (usually in the language library) have unknown internal structure, limitations, precision, and exception handling. Thoroughly test the functions that will be used, use a certified compiler, or review formal testing done on the compiler.
- ✓ Create coding standards for naming, indentation, commenting, subprogram size, etc. These factors affect the readability of the source code, and influence how well reviews and inspections can find errors.
- ✓ When doing mixed-language programming, separate out the "foreign" code, to enhance readability. Also document it well, including the justification. Mixed-language programming should be used only when necessary (such as accessing hardware with C, from a Java program).
- ✓ Use single purpose functions and procedures. This facilitates review and maintenance of the code.
- ✓ Use each variable for a single purpose only. "Reusing" a variable (usually a local) makes the source code confusing to read and maintain. If the variable is named properly for its original purpose, it will be misnamed for the new purpose.
- ✓ If the hardware configuration may change, for this project or in the future, isolate hardware-dependent code.
- ✓ Check for dead code. Unreachable code may indicate an error. It also causes confusion when reading the code.
- ✓ Use version control tools (configuration management).
- ✓ Utilize a bug tracking tool or database. Once a bug is found, it should be tracked until eliminated. Bug databases are also good sources to use when creating checklists for code inspections.
- ✓ Avoid large if-then-else and case statements. Such statements are extremely difficult to debug, because code ends up having so many different paths. The difference between best-case and worst-case execution time becomes significant. Also, the difficulty of structured code coverage testing grows exponentially with the number of branches.
- ✓ Avoid implementing delays as no-ops or empty loops. If this code is used on a different processor, or even the same processor running at a different rate (for example, a 25MHz vs. 33MHz CPU), the code may stop working or work incorrectly on the faster processor.

## E.5 Checklist of assembly programming practices for safety

- ✓ Use the macro facility of the assembler, if it exists, to simplify the code and make it more readable. Use if/else and loop control of the macro facility.
- ✓ If using labels, make the names meaningful. Label1 is not meaningful.
- ✓ Be careful ~~of~~ to check the ~~???~~ base of numbers (decimal, octal, hexadecimal)
- ✓ Use comments to describe WHAT the procedure or section is meant to do. It is not always clear from the assembly code.
- ✓ Update comments when the code changes, if the intent of the procedure or section changes as well.
- ✓ Use named code segments if possible. Consider separate segments for reset, non-volatile memory initialization, timer interrupts, and other special-purpose code

## E.6 Checklist of C programming practices for safety

Taken from nuclear standard, appendix B (see http://www.sohar.com/J1030/appb.htm)
Refer to generic list as well

- ✓ Limit the number and size of parameters passed to routines. Too many parameters affect readability and testability of the routine. Large structures or arrays, if passed by value, can overflow the stack, causing unpredictable results. Always pass large elements via pointers.
- ✓ Use recursive functions with great care. Stack overflows are common. Verify that there is a finite recursion!
- ✓ Utilize functions for boundary checking. Since C does not do this automatically, create routines that perform the same function. Accessing arrays or strings out-of-bounds is a common problem with unpredictable, and often major, results.
- ✓ Do not use the *gets* function, or related functions. These do not have adequate limit checks. Writing your own routine allows better error handling to be included.
- ✓ Use *memmove*, not *memcpy*. *Memcpy* has problems if the memory overlaps.
- ✓ Create wrappers for built-in functions to include error checking.
- ✓ If "*if…else if…else if…*" gets beyond two levels, use a *switch…case* instead. This increases readability.
- ✓ When using *switch…case*, always explicitly define *default*. If a *break* is omitted, to allow flow from one case to another, explicitly document it.
- ✓ Initialize local (automatic) variable. They contain garbage before explicit initialization. Pay special attention to pointers, since they can have the most dangerous effects.
- ✓ Initialize global variables in a separate routine. This ensures that variables are properly set at warm reboot.
- ✓ Check pointers to make sure they don't reference variables outside of scope. Once a variable goes out of scope, what it contains is undefined.
- ✓ Only use *setjmp* and *longjmp* for exception handling. These commands jump outside function boundaries and deviate from normal control flow.

- ✓ Avoid pointers to functions. These pointers cannot be initialized and may point to non-executable code. If they must be used, document the justification.
- ✓ Prototype all functions and procedures! This allows the compiler to catch errors, rather than having to debug them at run-time. Also, when possible, use a tool or other method to verify that the prototype matches the function.
- ✓ Minimize interface ambiguities, such as using expressions as parameters to subroutines, or changing the order of arguments between similar functions. Also justify (and document) any use of functions with an indefinite number of arguments. These functions cannot be checked by the compiler, and are difficult to verify.
- ✓ Do no use ++ or – operators on parameters being passed to subroutines or macros. These can create unexpected side effects.
- ✓ Use bit masks instead of bit fields, which are implementation dependent.
- ✓ Always explicitly cast variables. This enforces stronger typing. Casting pointers from one type to another should be justified and documented.
- ✓ Avoid the use of typedef's for unsized arrays. This feature is badly supported and error-prone.
- ✓ Avoid mixing signed and unsigned variables. Use explicit casts when necessary.
- ✓ Don't compare floating point numbers to 0, or expect exact equality. Allow some small differences due to the precision of floating point calculations.
- ✓ Enable and ~~h~~read compiler warnings. If an option, have warnings issued as errors. Warnings indicate deviation that may be fine, but may also indicate a subtle error.
- ✓ Be cautious if using standard library functions in a multitasking environment. Library functions may not be re-entrant, and could lead to unspecified results.
- ✓ Do not call functions within interrupt service routines. If it is necessary to do so, make sure the functions are small and re-entrant.
- ✓ Avoid the use of the ?: operator. The operator makes the code more difficult to read. Add comments explaining it, if it is used.
- ✓ Place #include directives at the beginning of a file. This makes it easier to know what files are actually included. When tracing dependencies, this information is needed.
- ✓ Use #define instead of numeric literals. This allows the reader or maintainer to know what the number actually represents (RADIUS_OF_EARTH_IN_KM, instead of 6356.91). It also allows the number to be changed in **one** place, if a change is necessitated later.
- ✓ Do not make assumptions about the sizes of dependent types, such as *int*. The size is often platform and compiler dependent.
- ✓ Avoid using reserved words or library function names as variable names. This could lead to serious errors. Also, avoid using names that are close to standard names, to improve the readability of the source code.

## E.7 Checklist of C++ programming practices for safety

Taken from nuclear standard, appendix B (see http://www.sohar.com/J1030/appb.htm)
Include all C programming practices and generic practices as well

- ✓ Always pass large parameters (structures, arrays, etc.) via reference. If it will not be changed, pass it as const.
- ✓ Group related parameters within a class, to minimize the number of parameters to be passed to a routine.
- ✓ Avoid multiple inheritance, which can cause ambiguities and maintenance problems.
- ✓ When overloading an operator, make sure that its usage is natural, not clever. Obscure overloading can reduce readability and induce errors. Using + to add two structures is natural. Using + with structures to perform a square of each element is not natural.
- ✓ Explicitly define class operators (assignment, etc.). Declare them private if they are not to be used.
- ✓ For all classes, define the following: Default constructor, copy constructor, destructor, operator=.
- ✓ Declare the destructor virtual. This is necessary to avoid problems if the class is inherited.
- ✓ Use *throw* and *catch* for exception handling, not C's *setjmp* and *longjmp*, which are difficult to recover from.
- ✓ Avoid pointers to members. These unnecessarily complicate the code.
- ✓ Use *const* variables and functions whenever possible. When something should not change, or a function should not change anything outside of itself, use *const*.

## E.8 Checklist of Fortran programming practices for safety

The following is extracted from Appendix A of Hatton, L. (1992) "Fortran, C or C++ for geophysical software development", Journal of Seismic Exploration, 1, p77-92.

- ✓ Unreachable code. This reduces the readability and therefore maintainability.
- ✓ Unreferenced labels. Confuses readability.
- ✓ The EQUIVALENCE statement except with the project manager's permission. This statement is responsible for many questionable practices in Fortran giving both reliability and readability problems. Permission should not be given lightly. A really brave manager will unequivocally forbid its use. Some programming standards do precisely this.
- ✓ Implicit reliance on SAVE. (This prejudices re-usability). A particular nasty problem to debug. Some compilers give you SAVE whether you specify it or not. Moving to any machine which implements the ANSI definition from one which SAVE's by default may lead to particularly nasty run and environment sensitive problems. This is an example of a statically detectable error which is almost impossible to find in a source debugger at run-time.
- ✓ The computed GOTO except with the project manager's permission. Often used for efficiency reasons when not justified. Efficiency should never precede clarity as a programming goal. The motto is "tune it when you can read it".
- ✓ Any Hollerith. This is non-ANSI, error-prone and difficult to manipulate.
- ✓ Non-generic intrinsics. Use generic intrinsics only on safety grounds. For example, use REAL() instead of FLOAT().
- ✓ Use of the ENTRY statement. This statement is responsible for unpredictable behavior in a number of compilers. For example, the relationship between dummy arguments specified in the SUBROUTINE or FUNCTION statement and in the ENTRY statements leads to a number of dangerous practices which often defeat even symbolic debuggers.
- ✓ BN and BZ descriptors in FORMAT statements. These reduce the reliability of user input.
- ✓ Mixing the number of array dimensions in calling sequences. Although commonly done, it is poor practice to mix array dimensions and can easily lead to improper access of n-dimensional arrays. It also inhibits any possibility of array-bound checking which may be part of the machine's environment. Unfortunately this practice is very widespread in Fortran code.
- ✓ Use of BLANK='ZERO' in I/O. This degrades the reliability of user input.
- ✓ Putting DO loop variables in COMMON. Forbidden because they can be inadvertently changed or even lead to bugs in some optimizing compilers.
- ✓ Declarations like REAL R(1). An old-fashioned practice which is frequently abused and leads almost immediately to array-bound violations whether planned or not. Array-bound violations are responsible for a significant number of bugs in Fortran.
- ✓ Passing an actual argument more than once in a calling sequence. Causes reliability problems in some compilers especially if one of the arguments is an output argument.
- ✓ A main program without a PROGRAM statement. Use of the program statement allows a programmer to give a module a name avoiding system defaults such as main and potential link clashes.

- ✓ Undeclared variables. Variables must be explicitly declared and their function described with suitable comment. Not declaring variables is actually forbidden in C and C++.
- ✓ The IMPLICIT statement. Implicit declaration is too sweeping unless it is one of the non-standard versions such as IMPLICIT NONE or IMPLICIT UNDEFINED.
- ✓ Labeling any other statement but FORMAT or CONTINUE. Stylistically it is poor practice to label executable statements as inserting code may change the logic, for example, if the target of a DO loop is an executable statement. This latter practice is also obsolescent in Fortran 90.
- ✓ The DIMENSION statement. It is redundant and on some machines improperly implemented. Use REAL etc. instead.
- ✓ READ or WRITE statements without an IOSTAT clause. All READ and WRITE statements should have an error status requested and tested for error-occurrence.
- ✓ SAVE in a main program. It merely clutters and achieves nothing.
- ✓ All referenced subroutines or functions must be declared as EXTERNAL. All EXTERNALS must be used. Unless EXTERNAL is used, names can collide surprisingly often with compiler supplied non-standard intrinsics with strange results which are difficult to detect. Unused EXTERNALS cause link problems with some machines, leading to spurious unresolved external references.
- ✓ Blank COMMON. Use of blank COMMON can conflict with 3rd. party packages which also use it in many strange ways. Also the rules are different for blank COMMON than for named COMMON.
- ✓ Named COMMON except with the project manager's permission. COMMON is a dangerous statement. It is contrary to modern information hiding techniques and used freely, can rapidly destroy the maintainability of a package. The author has bitter, personal experience of this ! Some company's safety-critical standards for Fortran explicitly forbid its use.
- ✓ Use of BACKSPACE, ENDFILE, REWIND, OPEN, INQUIRE and CLOSE. Existing routines for each of these actions should be designed and must always be used. Many portability problems arise from their explicit use, for example, the position of the file after an OPEN is not defined. It could be at the beginning or the end of the file. The OPEN should always therefore be followed by a REWIND, which has no effect if the file is already positioned at the beginning. OPEN and INQUIRE cause many portability problems, especially with sequential files.
- ✓ DO loops using non-INTEGER variables. The loop may execute a different number of times on some machines due to round-off differences. This practice is obsolescent in Fortran 90.
- ✓ Logical comparison of non-INTEGERS. Existing routines for this should be designed which understand the granularity of the floating point arithmetic on each machine to which they are ported and must always be used. Many portability problems arise from its explicit use. The author has personal experience whereby a single comparison of two reals for inequality executed occasionally in a 70,000 line program caused a very expensive portability problem.
- ✓ Any initialization of COMMON variables or dummy arguments is forbidden inside a FUNCTION, (possibility of side-effects). Expression evaluation order is not defined for Fortran. If an expression contains a function which affects other variables in the expression, the answer may be different on different machines. Such problems are exceedingly difficult to debug.
- ✓ Use of explicit unit numbers in I/O statements. Existing routines to manipulate these should be designed and must always be used. Many portability problems arise from their explicit use. The ANSI standard only requires them to be non-negative. What they are connected to differs wildly from machine to machine. Don't be surprised if your output comes out on a FAX machine !

- ✓ CHARACTER*(N) where N>255. A number of compilers do not support character elements longer than 255 characters.
- ✓ FORMAT repeat counts > 255. A number of compilers do not support FORMAT repeat counts of more than 255.
- ✓ COMMON blocks called EXIT. On one or two machines, this can cause a program to halt unexpectedly.
- ✓ Comparison of strings by other than the LLE functions. Only a restricted collating sequence is defined by the ANSI standard. The above functions guarantee portability of comparison.
- ✓ Using the same character variable on both sides of an assignment. If character positions overlap, this is actually forbidden by the standard but some compilers allow it and others don't. It should simply be avoided. The restriction has been removed in Fortran 90.
- ✓ Tab to a continuation line. Tabs are not part of the ANSI Fortran definition. They are however easily removable if used only to code lines and for indentation. If they are also used for continuation (like the VAX for example), it means they become syntactic and if your compiler does not support them, removing them is non-trivial.
- ✓ Use of PAUSE. An obsolescent feature with essentially undefined behavior.
- ✓ Use of '/' or '!' in a string initialized by DATA. Some compilers have actually complained at this !
- ✓ Using variables in PARAMETER, COMMON or array dimensions without typing them explicitly before such use. e.g. PARAMETER (R=3) INTEGER R Some compilers get it wrong.
- ✓ Use of CHAR or ICHAR. These depend on the character set of the host. Best to map onto ASCII using wrapper functions, but almost always safe today.
- ✓ Use of ASSIGN or assigned GOTO. An obsolescent feature legendary for producing unreadable code.
- ✓ Use of Arithmetic IF. An obsolescent feature legendary for producing unreadable code.
- ✓ Non-CONTINUE DO termination. An obsolescent feature which makes enhancement more difficult.
- ✓ Shared DO termination for nested DO loops. An obsolescent feature which makes enhancement more difficult.
- ✓ Alternate RETURN. An obsolescent feature which can easily produce unreadable code.
- ✓ Use of Fortran keywords or intrinsic names as identifier names. Keywords may be reserved in future Fortran standards. The practice also confuses readability, for example, IF (IF(CALL)) STOP=2 Some people delight in this sort of thing. Such people do not take programming seriously.
- ✓ Use of the INTRINSIC statement. The ANSI standard is particularly complex for this statement with many exceptions. Avoid.
- ✓ Use of END= or ERR= in I/O statements. (IOSTAT should be used instead). Using END= and ERR= with associated jumps leads to unstructured and therefore less readable code.
- ✓ Declaring and not using variables. This just confuses readability and therefore maintainability.

- ✓ Using COMMON block names as general identifiers, where use of COMMON has been approved. This practice confuses readability and unfortunately, compilers from time to time.
- ✓ Using variables without initializing them. Reliance on the machine to zero memory for you before running is not portable. It also produces unreliable effects if character strings are initialized to zero, (rather than blank). Always initialize variables explicitly.
- ✓ Use of manufacturer specific utilities unless specifically approved by the project manager. This simply reduces portability, in some cases pathologically.
- ✓ Use of non-significant blanks or continuation lines within user-supplied identifiers. This leads both to poor readability and to a certain class of error when lists are parsed, (it may have been a missing comma).
- ✓ Use of continuation lines in strings. It is not clear if blank-padding to the end of each partial line is required or not.
- ✓ Passing COMMON block variables through COMMON and through a calling sequence. This practice is both illegal and unsafe as it may confuse optimizing compilers and in some compilers simply not work. It is a very common error.
- ✓ An IF..ELSEIF block IF with no ELSE. This produces a logically incomplete structure whose behavior may change if the external environment changes. A frequent source of "unexpected software functionality".
- ✓ DATA statements within subroutines or functions. These can lead to non-reusability and therefore higher maintenance and development costs. If constants are to be initialized, use PARAMETER.
- ✓ DO loop variables passed as dummy arguments.
- ✓ Equivalencing any arrays other than at their base, even if use of EQUIVALENCE has been approved. Some machines still have alignment problems and also modern RISC platforms rely on good alignment for efficiency. So at best, it will be slow and at worst, it will be wrong.
- ✓ Equivalencing any variable with COMMON, even if use of EQUIVALENCE and COMMON has been approved. This rapidly leads to unreadable code.
- ✓ Type conversions using the default rules, either in DATA or assignment statements. Type conversions should be performed by the programmer - state what you mean. For example: R = I Wrong R = REAL(I) Right
- ✓ Use of mixed-type arithmetic in expressions.
- ✓ Use of precedence in any kind of expression. Parenthesize to show what you mean. Although Fortran precedence is relatively simple compared with C which has 15 levels of precedence, it is still easy to get it wrong.
- ✓ Concatenated exponentiation without parenthesizing, e.g. a**b**c. People too often forget what this means. Exponentiation associates from the right.
- ✓ Calling sequence matching. Make sure that calling sequence arguments match in type number and direction. Inconsistencies here are responsible for many unreliability problems in Fortran.

## E.9 Checklist of Pascal programming practices for safety

Taken from nuclear standard, appendix B (see http://www.sohar.com/J1030/appb.htm)
Refer to generic list as well

- ✓ If using pointers, use handles whenever possible. Handles allow the memory management to recapture and compact free memory.
- ✓ Use care with multiple-condition flow statements. The order of evaluation cannot be guaranteed.
- ✓ Isolate interrupt receiving tasks into implementation dependent packages. Pass the interrupt to the main tasks via a normal entry. Interrupt entries are implementation dependent features that may not be supported.
- ✓ Use symbolic constants instead of numeric literals. This increases readability and maintainability.
- ✓ Avoid the use of the *mod* operator. Not all compilers follow the Standard, and this will create portability problems.

## E.10 Checklist for Visual Basic

From “An Evaluation of Object-Based Programming with Visual Basic”, James M. Dukovic and Daniel T. Joyce, 0-7803-2492-7/95 IEEE

1. If you want to add a public function or subroutine to a form, place it in a code module having the same name as the form. Place private functions and subroutines inside the general procedure section of the form. This is required because general procedures inside forms are not visible to other objects.
2. Do not use global variables. Use procedure-level variables for temporary variables and module-level variables for object data. This will require you to pass parameters to all methods ensuring a cleaner interface.
3. Do not use the *static* statement to declare variables or procedures. Use module-level variables for all object data. Static variables can become lost in your code.
4. Create handles to access properties declared in code modules. You may access properties in forms and controls directly. This will help hide the implementation of the object.
5. Code modules should be objects. They should have data and methods (subroutines and functions) to access and manipulated the data. Use object-oriented design techniques to define your objects.
6. Forms should only contain code that is unique to the form. Most code should be placed in modules. Forms are likely to change. Modules are much more stable.
7. Set the Visual Basic Environment Option called *Require Variable Declaration* to *Yes*. This will force you to declare variables explicitly. It does not, however, force you to specify a type.
8. Explicitly declare data types for all variables and functions. This will improve readability and reduce the risk of data type errors.
9. Hide the implementation of an object as much as possible. The object’s interface should reveal as little about the implementation and underlying data structure as possible. This will allow you to make changes to an object without impacting other objects.
10. Avoid using environment specific parameters in the object’s interface. For example, small, medium or large is preferable to passing pixels or twips.
11. Use the variant data type sparingly. Although it allows a form of parametric polymorphism, the resultant code can be difficult to understand and its use will increase the risk of data type errors.
12. Declare subroutines and functions as private whenever possible. Subroutines and functions should only be public if they are used by other objects. This will make the code more readable and it will prevent other objects from accessing private methods directly.
13. Document your interfaces at the top of each method. Include variable types, sizes, and allowable values.
14. Use standard objects whenever possible. For example, use message boxes and dialogue boxes instead of creating specific forms.
15. Do not use the OptionBase statement to alter the lower bound of array subscripts. Altering the lower bound may make reuse more difficult.
16. Design your objects with weak coupling. That is, create your object so its dependency on other objects is minimal. This will make it easier to understand your objects.

## *E.11 Checklist for selecting an RTOS*

From "Selecting a Real-Time Operating System", Greg Hawley, Embedded Systems Programming, March, 1999

| Criteria | Considerations |
|---|---|
| Language/Microprocessor Support | The first step in finding an RTOS for your project is to look at those vendors supporting the language and microprocessor you'll be using. |
| Tool Compatibility | Make sure your RTOS works with your ICE, compiler, assembler, linker, and source code debuggers. |
| Services | Operating systems provide a variety of services. Make sure your OS supports the services (queues, times, semaphores) you expect to use in your design. |
| Footprint | RTOSes are often scalable, including only those services you end up needing in your applications. Based on what services you'll need, and the number of tasks, semaphores, and everything else you expect to use, make sure your RTOS will work in the RAM and ROM you have |
| Performance | Can your RTOS meet your performance requirements? Make sure you understand benchmarks vendors give you and how they apply to the hardware you will really be using. |
| Software Components | Are required components (protocol stacks, communications services, real-time databases, Web services, virtual machines, graphics libraries, and so on) available for your RTOS? How much effort will it be to integrate them? |
| Device Drivers | If you're using common hardware, are device drivers available for your RTOS? |
| Debugging Tools | RTOS vendors may have debugging tools that help find defects that are harder to find with source-level debuggers (such as deadlocks, forgotten semaphore puts, and so on). |
| Standards Compatibility | Are there safety or compatibility standards your application demands? Make sure your RTOS complies. |
| Technical Support | Phone support is typically covered for a limited time after your purchase or on a year-to-year basis through support. Sometimes applications engineers are available. Additionally, some vendors provide training and consulting. |
| Source vs. Object Code | With some RTOSes you get the source code to the operating system when you buy a license. In other cases, you get only object code or linkable libraries. |
| Licensing | Make sure you understand how the RTOS vendor licenses their RTOS. With some vendors, run-time licenses are required for each board shipped and development tool licenses are required for each developer. |
| Reputation | Make sure you're dealing with someone you'll be happy with. |
| Services | Real-time operating systems provide developers a full complement of features: several types of semaphores (counting, mutual exclusion), times, mailboxes, queues, buffer managers, memory system managers, events, and more. |
| Priority Inheritance | Must support, or priority inversion can result |

## *E.12 Good Programming Practices Checklist*

These items should be considered when creating a Coding Standard or when beginning a software project.

| **Programming Practice** | **Yes/No/NA** | **Comment or Justification** |
|---|---|---|
| **General Suggestions** | | |
| **CPU self test.** Test the CPU on boot up. | | |
| **Fill ROM/RAM/flash with a known pattern** (halt, illegal instruction, return) to guard against illegal jumps. | | |
| **ROM tests**. Verify integrity of ROM (EEPROM, Flash disk, etc.) prior to executing the software stored in it. | | |
| **Watchdog Timers.** Implement a watchdog timer to reboot software if it gets “stuck”. | | |
| **Guard against Variable Corruption.** Store multiple copies of critical variables, especially on different storage media or physically separate memory. | | |
| **Stack Checks.** Checking the stack guards against stack overflow or corruption. By initializing the stack to a known pattern, a stack monitor function can be used to watch the amount of available stack space. | | |
| ~~after the code is written.”~~**Write what you need, and use what you write!** Don’t make unnecessarily verbose or lengthy documentation, unless contractually required. It is better to have short documents that the developers will actually read and use. | | |
| Initialize all **unused memory** locations to a pattern that, if executed as an instruction, will cause the system to revert to a known safe state. | | |
| **Don’t use a stop or halt instruction**. The CPU should be always executing, whether idling or actively processing | | |
| **When possible, put safety-critical operational software instructions in nonvolatile read-only memory.** | | |
| **Don’t use scratch files** for storing or transferring safety-critical information between computers or tasks within a computer. | | |

| Programming Practice | Yes/No/NA | Comment or Justification |
|---|---|---|
| **Keep Interface Control Documents up to date.** Out-of-date information usually leads to one programmer creating a module or unit that will not interface correctly with another unit. The problem isn't found until late in the testing phase, when it is expensive to fix. | | |
| **Prohibit program patches.** During development, patching a program is a bad idea. During operations, patching may be a necessity, but should still be carefully considered. | | |
| **Follow the two person rule**. At least two people should be thoroughly familiar with the design, code, testing and operation of each software module of the system. If one person leaves the project, someone else understands what is going on. | | |
| **Design Issues** | | |
| **Program Calculation Checks.** Simple checks can be used to give confidence in the results from calculations. | | |
| **Verify all reused code was designed for reuse.**~~.~~ | | |
| **Do not implement program as "One big loop".** ~~"When real-time software is designed as a single big loop, we have no flexibility to modify the execution time of various parts of the code independently. Few real-time systems need to operate everything at the same rate."~~A single large loop forces all parts of the software to operate at the same rate. | | |
| **Analyze hardware peculiarities before starting software design.** | | |
| **Avoid inter-module dependencies when possible.** This maximizes software reusability. | | |
| **Create more than a single design diagram.** Getting the entire design on paper is essential. | | |
| **Design in error detection and handling!** Tailor the effort to the level of the code – don't put it everywhere! | | |
| **Perform a memory analysis of the design.** Estimate how much memory your system uses and adjust the design if the system is bumping up against its limits. | | |

| Programming Practice | Yes/No/N A | Comment or Justification |
|---|---|---|
| **Avoid indiscriminate use of interrupts.** Use of interrupts can cause priority inversion in real-time systems if not implemented~~carefully. "Interrupts are perhaps the biggest cause of priority inversion in real-time systems, causing the system to not meet all of its timing requirements. The reason for this delay is that interrupts preempt everything else and aren't scheduled."This~~carefully. | | |
| **Use come-from checks.** For safety critical modules, make sure that the correct module called it, and that it was not called accidentally by a malfunctioning module. | | |
| Provide **separate authorization and separate control functions** to initiate a critical or hazardous function. This includes separate "arm" and "fire" commands for critical capabilities. | | |
| **Do not use input/output ports for both critical and non-critical functions**. | | |
| Provide **sufficient difference in addresses** between critical I/O ports and non-critical I/O ports, such that a single address bit failure does not allow access to critical functions or ports. | | |
| Make sure all **interrupt priorities** and responses are defined. All interrupts should be initialized to a return, if not used by the software. | | |
| Provide for an **orderly shutdown** (or other acceptable response) upon the detection of **unsafe conditions**. | | |
| Provide for an **orderly system shutdown** as the result of a **command shutdown**, power interruptions, or other failures. | | |
| Protect against **out-of-sequence transmission** of safety-critical function messages by detecting and deviation from the normal sequence of transmission. Revert to a known safe state when out-of-sequence messages are detected. | | |
| **Hazardous sequences should not be initiated by a single keyboard entry**. | | |
| Prevent **inadvertent entry into a critical routine**. Detect such entry if it occurs, and revert to a known safe state. | | |
| When **safety interlocks** are removed/bypassed for a test, the software should verify the reinstatement of the interlocks at the completion of the testing. | | |

| Programming Practice | Yes/No/NA | Comment or Justification |
|---|---|---|
| **Critical data** communicated from one CPU to another should be verified prior to operational use. | | |
| Set a **dedicated status flag** that is updated between each step of a hazardous operation. | | |
| **Verify critical commands** prior to transmission, and upon reception. It never hurts to check twice! | | |
| **Make sure all flags used are unique and single purpose.** | | |
| Put the majority of **safety-critical decisions and algorithms** in a single (or few) software development module(s). | | |
| **Decision logic** using data from hardware or other software modules should not be based on values of all ones or all zeros. Use specific binary patterns to reduce the likelihood of malfunctioning hardware/software satisfying the decision logic. | | |
| **Perform reasonableness checks on all safety-critical inputs**. | | |
| Perform a **status check** of critical system elements prior to executing a potentially hazardous sequence. | | |
| Always **initialize the software into a known safe state.** This implies making sure all variables are set to an initial value, and not the previous value prior to reset. | | |
| **Don't allow the operator to change safety-critical time limits in decision logic**. | | |
| When the system is safed, usually in response to an anomalous condition or problem, provide the **current system configuration** to the operator. | | |
| **Create a list of possible hardware failures that may impact the software.** The list will be invaluable when testing the error handling capabilities of the software, as well as making sure hardware failures have been considered in the design. | | |
| **Be careful if using multi-threaded programs.** Subtle program errors can result from unforeseen interactions among multiple threads. | | |
| **Consider the stability of the requirements.** If the requirements are likely to change, design as much flexibility as possible into the system. | | |

| Programming Practice | Yes/No/N A | Comment or Justification |
|---|---|---|
| **Design for weak coupling** between modules (classes, etc.). The more independent the modules are, the less you can screw them up later in the process. | | |
| **Reduce complexity.** Calculate a complexity metric. Look at modules that are very complex and reduce them if possible. | | |
| **Implementation (Coding) Issues** | | |
| **So not implement delays as empty loops.** This can create problems (and timing difficulties) if the code is run on faster or slower machines, or even if recompiled with a newer, optimizing compiler. | | |
| **Avoid fine-grain optimizing during first implementation.** | | |
| **Check variables for reasonableness before use.** If the value is out of range, there is a problem – memory corruption, incorrect calculation, hardware problems (if sensor), etc. | | |
| **Use readbacks to check values.** When a value is written to memory, the display, or hardware, another function should read it back and verify that the correct value was written. | | |
| **Safety-critical modules should have only one entry and one exit point.** | | |
| **Create a dependency graph.** Given such a diagram, it's easy to identify what parts of the software can be reused, create a strategy for incremental testing of modules, and develop a method to limit error propagation through the entire system. | | |
| **Consider compiler optimization carefully.** Debuggers may not work well with optimized code. | | |
| **Testing Issues** | | |
| **Plan and script all tests.** Do not rely on interactive and incomplete test programs. | | |
| **Measure the execution time of your code.** Determine if there are any bottlenecks, or any modules that should be considered for optimization. | | |

| Programming Practice | Yes/No/NA | Comment or Justification |
|---|---|---|
| **Use execution logging,** with independent checking, to find software runaway, illegal functions, or out-of-sequence execution. | | |
| **Test for memory leakage.** Instrument the code and run it under load and stress tests. | | |
| **Use a simulator or ICE (In-circuit Emulator)** system for debugging in embedded systems. | | |

## *E.13 Software Requirements Phase Checklist*

| | |
|---|---|
| Project: | **Safety Effort Level:** |
| **Phase:** ***Software Requirements*** | **Software Safety Analyst:** |

| **Technique** | **Perform?** | **Justification for not performing** |
|---|---|---|
| Preliminary Hazard Analysis (PHA)<br>*Section 2.4* | | |
| Software Safety Requirements Flow-down Analysis<br>*Section 5.1.1* | | |
| Checklists and cross references<br>*Section 5.1.1.1* | | |
| Requirements Criticality Analysis<br>*Section 5.1.2* | | |
| Generic Software Safety Requirements<br>*Section 4.2.2* | | |
| Specification Analysis<br>*Section 5.1.3* | | |
| Formal Methods - Specification Development<br>*Section 4.2.3* | | |
| Formal Inspections of Specifications<br>*Section 4.2.5* | | |
| Timing, Throughput And Sizing Analysis<br>*Section 5.1.5* | | |
| Software Fault Tree Analysis<br>*Section 5.1.6, Appendix B* | | |

## *E.14 Architectural Design Phase Checklist*

| | |
|---|---|
| **Project:** | **Safety Effort Level:** |
| **Phase:** *Architectural Design Phase* | **Software Safety Analyst:** |

| **Technique** | **Perform?** | **Justification for not performing** |
|---|---|---|
| COTS & Software Reuse Considerations<br>*Sections 4.3.3 and 7.1* | | |
| Selection of Programming Language, Environment, Tools, and Operating System<br>*Section 4.3.4* | | |
| Coding Standards<br>*Section 4.3.5* | | |
| Update Criticality Analysis<br>*Section 5.2.1* | | |
| Conduct Hazard Risk Assessment<br>*Section 5.2.2* | | |
| Analyze Architectural Design<br>*Section 5.2.3* | | |
| Interdependence Analysis<br>*Section 5.2.4.1* | | |
| Independence Analysis<br>*Section 5.2.4.2* | | |
| Update Timing/Throughput/Sizing Analysis<br>*Section 5.2.5* | | |
| Update Software Fault Tree Analysis<br>*Section 5.2.6* | | |
| Formal Inspections of Architectural Design Products<br>*Section 5.2.7* | | |
| Formal Methods and Model Checking<br>*Section 5.2.8* | | |

## *E.15 Detailed Design Phase Checklist*

| | |
|---|---|
| **Project:** | **Safety Effort Level:** |
| **Phase:** ***Detailed Design Phase*** | **Software Safety Analyst:** |

| **Technique** | **Perform?** | **Justification for not performing** |
|---|---|---|
| Model Checking<br>*Section 4.2.4* | | |
| Data Logic Analysis<br>*Section 5.3.1* | | |
| Design Data Analysis<br>*Section 5.3.2* | | |
| Design Interface Analysis<br>*Section 5.3.3* | | |
| Design Constraint Analysis<br>*Section 5.3.4* | | |
| Rate Monotonic Analysis<br>*Section 5.3.5* | | |
| Dynamic Flowgraph Analysis<br>*Section 5.3.6* | | |
| Markov Modeling<br>*Section 5.3.7* | | |
| Measurement of Complexity<br>*Section 5.3.8* | | |
| Selection of Programming languages<br>*Section 5.3.9* | | |
| Formal Methods and Model Checking<br>*Section 5.3.10* | | |
| Requirements State Machines<br>*Section 5.3.11, Appendix D* | | |
| Formal Inspections of Detailed Design Products<br>*Section 5.3.12* | | |
| Software Failure Modes and Effects Analysis<br>*Section 5.3.13, Appendix C* | | |
| Updates to Previous Analyses (SFTA, Timing, Criticality, etc.)<br>*Section 5.3.14* | | |

## E.16 Implementation Phase Checklist

| | | |
|---|---|---|
| **Project:** | | **Safety Effort Level:** |
| **Phase:** ***Software Implementation Phase*** | | **Software Safety Analyst:** |
| **Technique** | **Perform?** | **Justification for not performing** |
| Coding Checklists<br>*Section 4.5.1* | | |
| Defensive Programming<br>*Section 4.5.2* | | |
| Refactoring<br>*Section 4.5.3* | | |
| Code Logic Analysis<br>*Section 5.4.1* | | |
| Code Data Analysis<br>*Section 5.4.2* | | |
| Code Interface Analysis<br>*Section 5.4.3* | | |
| Update Measurement of Complexity<br>*Section 5.4.4* | | |
| Update Design Constraint Analysis<br>*Section 5.4.5* | | |
| Formal Code Inspections, Checklists, and Coding Standards<br>*Section 5.4.6* | | |
| Formal Methods<br>*Section 5.4.7* | | |
| Unused Code Analysis<br>*Section 5.4.8* | | |
| Interrupt Analysis<br>*Section 5.4.9* | | |
| Final Timing, Throughput, and Sizing Analysis<br>*Section 5.4.10* | | |
| Program Slicing<br>*Section 5.4.11* | | |
| Update Software Failure Modes and Effects Analysis<br>*Section 5.4.12* | | |

## *E.17 Software Testing Phase Checklist*

| | |
|---|---|
| **Project:** | **Safety Effort Level:** |
| **Phase:** ***Software Testing Phase*** | **Software Safety Analyst:** |

| **Technique** | **Perform?** | **Justification for not performing** |
|---|---|---|
| Unit Level Testing<br>*Section 4.5.4* | | |
| Integration Testing<br>*Section 4.6.3* | | |
| System and Functional Testing<br>*Section 4.6.5* | | |
| Software Safety Testing<br>*Section 4.6.6* | | |
| Test Coverage Analysis<br>*Section 5.5.1* | | |
| Formal Inspections of Test Plan and Procedures<br>*Section 5.5.2* | | |
| Reliability Modeling<br>*Section 5.5.3* | | |
| Checklists of Tests<br>*Section 5.5.4* | | |
| Test Results Analysis<br>*Section 5.5.5* | | |
| Independent Verification and Validation<br>*Section 5.5.6* | | |

## *E.18 Dynamic Testing Checklist*

| | |
|---|---|
| **Project:** | **Safety Effort Level:** |
| **Phase:** ***Dynamic Testing (Unit or Integration Level)*** | **Software Safety Analyst:** |

| **Technique** | **Perform?** | **Justification for not performing** |
|---|---|---|
| Typical sets of sensor inputs | | |
| Test specific functions | | |
| Volumetric and statistical tests | | |
| Test extreme values of inputs | | |
| Test all modes of each sensor | | |
| Every statement executed once | | |
| Every branch tested at least once | | |
| Every predicate term tested | | |
| Every loop executed 0, 1, many, max-1, max, max+1 times | | |
| Every path executed | | |
| Every assignment to memory tested | | |
| Every reference to memory tested | | |
| All mappings from inputs checked | | |
| All timing constraints verified | | |
| Test worst case interrupt sequences | | |
| Test significant chains of interrupts | | |
| Test Positioning of data in I/O space | | |
| Check accuracy of arithmetic | | |
| All modules executed at least once | | |
| All invocations of modules tested | | |

## E.19 Software System Testing Checklist

**Project:** | **Safety Effort Level:**

**Phase:** *Software System Testing* | **Software Safety Analyst:**

| **Technique** | **Perform?** | **Justification for not performing** |
|---|---|---|
| Simulation (Test Environment) | | |
| Load Testing | | |
| Stress Testing | | |
| Boundary Value Tests | | |
| Test Coverage Analysis | | |
| Functional Testing | | |
| Performance Monitoring | | |
| Disaster Testing | | |
| Resistance to Failure Testing | | |
| “Red Team” Testing | | |
| Formal Progress Reviews | | |
| Reliability Modeling | | |
| Checklists of Tests | | |