Foreward

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 Guidebook for Safety Critical Software - Analysis and Development, was prepared by the NASA Lewis Research Center, Office of Safety and Mission Assurance, under a Research Topic (RTOP) task for the National Aeronautics and Space Administration.

The NASA Software Safety Standard NSS 1740.13 [1]prepared by NASA HQ addresses the “who, what, when and why” of Software Safety Analyses. This Software Safety Analysis Guidebook addresses the “how to”.

1.1 Scope

The focus of this document is on analysis and development of safety critical software. This guidebook can also be used for analysis and development of firmware which is software residing in non-volatile memory (e.g.. ROM or EPROM).

This Guidebook describes resource data required for each analysis task, the methodology and tools for performing the analysis, and the output products. It also describes how to use these products in the overall risk management activity.

The Guidebook goes on to describe techniques and procedures in more detail. To make the guidebook more practical, it contains analysis examples and possible pitfalls and problems that may be encountered during the analysis process. This Guidebook does not contain new analysis techniques. It is a synergistic collection of techniques already in use throughout NASA and industry. 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.

Numerous existing documents identify various analysis techniques, and those techniques which are well described elsewhere are referenced. 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 bottom-up 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.

The term "software components" is used in a generic sense to represent important software development products such as software requirements, software designs, software code or program sets, software tests, etc. can we remove this?

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 product assurance, system safety and software safety organizations).

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.

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.

We would like to thank the many NASA engineers and contractors who reviewed drafts of the guidebook, these include:

Reviewer	NASA Center	City	Contractor
Gilbert White	NASA HQ Code-QS	Washington D.C.	
Paul Senger	NASA HQ Code-QS	Washington D.C.	Vitro Corp
Ricky Butler	NASA Langley	Hampton, VA	
Michael Holloway	NASA Langley	Hampton, VA	
James Watson	NASA Langley	Hampton, VA	
Judith Gregory	NASA Marshall	Huntsville, AL	
Bonnie Pfitzer	NASA Marshall	Huntsville, AL	Madison Research Corp
Scott Seyl	NASA Johnson	Houston, TX	
David Tadlock	NASA Johnson	Houston, TX	
Dr. Robyn Lutz	JPL Caltech	Pasadena, CA	Univeristy of Iowa
George Abendroth	NASA Lewis	Cleveland, OH	

Michael Ciancone	NASA Lewis	Cleveland, OH	
Sell James	NASA Lewis	Cleveland, OH	
David Marchese	NASA Lewis	Cleveland, OH	
Susan Cole	NASA Lewis	Cleveland, OH	Raytheon Corp
James Roupp	NASA Lewis	Cleveland, OH	Raytheon Corp
Michele D. Smith	NASA Lewis	Cleveland, OH	Raytheon Corp
William Stuckey	NASA Lewis	Cleveland, OH	Raytheon Corp

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.

2. SYSTEM SAFETY PROGRAM

A system safety program is a prerequisite to performing analysis or development of safety critical software. This program maps out the types of analysis and the schedule for performing a series of systems and subsystem level analyses throughout the development cycle. It will also address how safety analyses results and the sign-off and approval process should be handled.

The first system safety program activity is a Preliminary Hazard Analysis (PHA), discussed in Section 2.1. The results of this step, a list of hazard causes and potential hazard controls, are flowed to the safety requirements development activity discussed in Section 2.2.

An aerospace system generally contains three elements: hardware, software and one or more human operators (e.g., ground controllers, mission specialists or vehicle pilots). Each of these 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 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. Therefore, throughout this guidebook, software is considered to be a system subset, (i.e., a Sub-system), within a larger System.

Before software can be analyzed or developed for use in a hazardous system or environment, a system PHA must be performed. Once initial system PHA results are available, safety requirements flowdown and subsystem and component hazard analyses can begin. System safety program analyses follow the lifecycle of the system and software development efforts, commencing with the Preliminary Hazard Analysis (PHA).

2.1 Preliminary Hazard Analysis (PHA)

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.

The PHA is the first of a series of system level hazard analyses, whose scope and methodology is described in the NASA NHB 1700 series documents [1] and NSTS 13830 Implementation Procedure for NASA Payload System Safety Requirements.[2] It is not the purpose of this document to explain or duplicate the process of systems hazard analyses described in the other NASA documents. However, for software developers, managers, and others unfamiliar with NASA system safety, the following section summarizes those processes and shows how the outputs of the PHA are fed into the software safety analysis process.

2.1.1 PHA Approach

The following is exerpted from NHB 1700.1 (V1-B) Appendix-H:

The Preliminary Hazard Analysis "documents which generic hazards (See Figure 2-1, Generic Hazards Checklist for a sample checklist of 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 may be used to identify safety-critical systems that will require the application of Failure Modes and Effects Analysis (FMEA) 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 from similar systems, other lessons learned, and hazards associated with the proposed design or function. Mishap and lessons learned information are available in the Mishap Reporting and Corrective Action System (MR/CAS) and the Lessons Learned Information System (LLIS). The risk assessment developed from the PHA will be used to ensure safety considerations are included in the 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."

Table 2-1 Generic Hazards Checklist

I	CONTAMINATION/CORROSION	A. CHEMICAL DISASSOCIATION B. CHEMICAL REPLACEMENT/COMBINATION C. MOISTURE D. OXIDATION E. ORGANIC (FUNGUS/BACTERIAL, ETC. F. PARTICULATE G. INORGANIC (INCLUDES ASBESTOS)
II	ELECTRICAL DISCHARGE/SHOCK	A. EXTERNAL SHOCK B. INTERNAL SHOCK C. STATIC DISCHARGE D. CORONA E. SHORT
III	ENVIRONMENTAL/WEATHER	A. FOG B. LIGHTNING C. PRECIPITATION (FOG/RAIN/SNOW/SLEET/HAIL) D. SAND/DUST E. VACUUM F. WIND G. TEMPERATURE EXTREMES
IV	FIRE/EXPLOSION	A. CHEMICAL CHANGE (EXOTHERMIC/ENDOTHERMIC) B. FUEL AND OXIDIZER IN PRESENCE OF PRESSURE AND IGNITION SOURCE C. PRESSURE RELEASE/IMPLOSION D. HIGH HEAT SOURCE
V	IMPACT/COLLISION	A. ACCELERATION (INCLUDING GRAVITY) B. DETACHED EQUIPMENT C. MECHANICAL SHOCK/VIBRATION/ACOUSTICAL D. METEOROIDS/METEORITES E. MOVING/ROTATING EQUIPMENT
VI	LOSS OF HABITABLE ENVIRONMENT*	A. CONTAMINATION B. HIGH PRESSURE C. LOW OXYGEN CONTENT D. LOW PRESSURE E. TOXICITY F. LOW TEMPERATURE G. HIGH TEMPERATURE
VII	PATHOLOGICAL/PHYSIOLOGICAL/ PSYCHOLOGICAL	A. ACCELERATION/SHOCK/IMPACT/VIBRATION B. ATMOSPHERIC PRESSURE (HIGH, LOW, RAPID CHANGE) C. HUMIDITY D. ILLNESS E. NOISE F. SHARP EDGES G. SLEEP, LACK OF H. VISIBILITY (GLARE, WINDOW/HELMET FOGGING) I. TEMPERATURE J. WORKLOAD, EXCESSIVE K. HIGH PLACES (POSSIBLE FALLING)
VIII	RADIATION	A. ELECTROMAGNETIC B. IONIZING RADIATION (INCLUDES RADON) C. NONIONIZING RADIATION
IX	TEMPERATURE EXTREMES	A. HIGH B. LOW C. VARIATIONS

*Health issues require coordination with Occupational Health Personnel

2.1.1.1 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 effectors 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 and safety assessment 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)
2) "Must not work" functions (MNWF)

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 MNWFs (Must-Not-Work-Functions) as well as the MWFs (Must Work Functions).

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. Obviously it becomes a MWF when it is time for it 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.

*2.1.1.2 **Risk Assessment***

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

The following definitions of hazard severity levels are from NASA NHB 1700.1.

Catastrophic	Loss of entire system; loss of human life or permanent disability
Critical	Major system damage; severe injury or temporary disability
Marginal	Minor system damage; minor injury
Negligible	No system damage; no injury

Table 2-2 Hazard Prioritization - System Risk Index

Severity Levels	Likelihood of Occurrence			
	Probable	Occasional	Remote	Improbable
Catastrophic	1	1	2	3
Critical	1	2	3	4
Marginal	2	3	4	5
Negligible	3	4	5	5

1 = Highest Priority (Highest System Risk), 5 = 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 level, 5, does not require 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 below in Section 3, where three levels of safety analysis are described, i.e. Minimum, Moderate and Full. These correspond to risk as follows:

Risk Level	Type of Safety Analysis Recommend
1	Not Applicable (Prohibited)
2	Full
3	Moderate
4	Minimum
5	None

The NASA policy towards hazards of Risk level 2, 3 or 4 is defined in NHB 1700.1 (V1-B) Section 102-f, as follows: “Eliminate or control hazards by corrective action with the following priorities:

(1) Eliminate hazards by design.”

> Hazards are eliminated where possible. This is accomplished through design, such as by eliminating an energy source. For example, a hazardous high voltage battery may be replaced with a non-hazardous low voltage battery.

“(2) Minimize or negate the effects of hazards by design.”

> Hazards which cannot be eliminated must be controlled. For those hazards, the PHA evaluates what can cause the hazards, and how to control the potential hazard causes. Control by design is preferred. For example, providing failure tolerance (e.g. by redundancy - series and/or parallel as appropriate), by providing substantial margins of safety, or by providing automatic safing.

“(3) Install safety devices.”

“(4) Install caution and warning devices.”

> Software used in these caution and warning devices is <u>safety critical</u>.

“(5) Develop administrative controls, including special procedures.”

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

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. 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.

Which Software is Hazardous ?

Either remote or embedded real-time software which controls hazardous or safety critical hardware are considered hazardous. As it is normally cost prohibitive to analyze all system software to the same depth, the system PHA, when integrated with the requirements placed on the software, identifies those programs, routines, or modules that are critical to system safety and must be examined in depth.

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 which controls real world processes, whose malfunction can result in a hazard. However, other types of software can also be the subject of software safety.

Software which performs off-line processes, results of which may result in a decision for automatic or manual operation that could be hazardous must be considered as safety critical. For example, test software (software used to test hardware or other software), and certain types of off-line computerized analysis could potentially lead to a hazardous condition. Faulty test software used to test safety critical hardware could cause damage or fail to report failures.   Offline computerized analyses such as thermal analysis or structural analysis are sometimes used to verify safety critical hardware designs (e.g., structural strength or fatigue life) and, if performed incorrectly, can indirectly result in hardware failures, especially if the hardware is not tested.

2.1.2 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, and interactor, time constraints, etc. (as per para 2.1.1.1 above)

Next, the hazard cause(s) are identified. 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.

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 Ser 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 2-1 Payload Hazard Report Form shows the NASA Shuttle Payload Hazard Report, which identifies hazards, their causes, controls, verification methods and verification status.   Detailed instructions for completing this form are given in NHB 1700.1 (V1-B) NASA Safety Policy and Requirements Document, Chapter-3, System Safety, and Appendix-H (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 process 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 a software requirement and is usually a system level requirement. It can result in further flowdown to software functionality by identifying software Hazard Controls as described below.

Hazard Cause: This is usually a fault or defect in hardware or software. A software cause might be an inadvertent command, improper execution of a critical process, loss of control, incorrect sequence, lack of necessary command, etc.

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 the NSTS Payload Hazard Report (Figure 3.1) 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.

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

Figure 2-1 Payload Hazard Report Form

PAYLOAD HAZARD REPORT		NO.	
PAYLOAD:		PHASE:	
SUBSYSTEM:	HAZ. GROUP:	DATE:	
HAZARD TITLE:			
APPLICABLE SAFETY REQUIREMENTS:		HAZARD CATEGORY	
			CATASTROPHIC
			CRITICAL
DESCRIPTION OF HAZARD:			
HAZARD CAUSES: See Continuation Sheet			
HAZARD CONTROLS: See Continuation Sheet			
SAFETY VERIFICATION METHODS: See Continuation Sheet			
STATUS OF VERIFICATION: See Continuation Sheet			
APPROVAL	PAYLOAD ORGANIZATION	STS	
PHASE I			
PHASE II			
PHASE III			

JSC Form 542B (rev Nov. 82) NASA JSC

Figure 2-2 Payload Hazard Report Continuation Sheet

PAYLOAD HAZARD REPORT CONTINUATION SHEET	NO.
PAYLOAD:	PHASE:
HAZARD TITLE:	DATE:
HAZARD CAUSES:	
HAZARD CONTROLS:	
SAFETY VERIFICATION METHODS:	
STATUS OF VERIFICATION:	

JSC Form 542B (rev Nov. 82) NASA JSC

2.1.3 Tools and Methods for PHA

Some tools and methods for performing a formal PHA are detailed in NASA Safety Manual, Chapter-3, System Safety [1]. Only items pertaining to software tools and software methods are discussed in this guide.

2.2 Safety Requirements Flowdown

The results of the first cycle of the system safety analysis, the PHA are a list of hazard causes and a set of candidate hazard controls. Any software hazard causes and software hazard controls and mitigators are taken forward as inputs to the software safety requirements flowdown process.

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

At least one software requirement is generated for each software hazard control. It 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 is subjected to rigorous software quality control, analysis, and testing. It is also to be traced through the software safety analysis process including verification.

2.2.1 Relating Software to Hazards

Generally a “software related hazard” is a result of loss of control of some safety critical system function. Remember, Software is a **SUBSET** of the complete system (a Sub-system) and as such must usually be considered in a systems context.

What is safety critical software ?

Software within a control system containing one or more hazardous or safety critical functions, 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 messages making safety critical decisions to operators, or
- used as a means of failure/fault detection.

Also, "contamination" of safety critical software by nearby uncontrolled software 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 of critical software when sharing a CPU or any routines with the safety critical software

A hazard cause is typically a fault, error or defect in either hardware, or software, or human operator. 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 2-3 Hazard Causes and Controls - Examples on page 16.

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.

Many hardware hazard causes can be addressed with software hazard controls. For example, software can detect hazardous hardware conditions (via instrumentation) and execute a corrective action and/or warn operators. Increasingly, hazard responses are delegated to software because of the quick reaction time needed and as personnel are replaced by computers.

However, in those designs, the software control must be isolated from the hazard cause which it is controlling. That is, the software hazard control must normally reside on a different computer processor from the one where the hazard/anomally might occur. Since a processor hardware failure would most likely affect its own software's ability to react to that CPU hardware failure it is best to monitor and control from a separate CPU. This is a challenging aspect of software safety, because multi-processor 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 not failure tolerant. Many also believe that computers fail safe, whereas NASA experience has shown that computers often 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 <u>can</u> cause a more serious hazardous condition.

NASA based on their 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.

In Brief:

Catastrophic Hazards: must be able to tolerate two hazard control failures.

Critical Hazards: must be able to tolerate a single hazard control failure.

NASA relies primarily on hardware controls to prevent hazards. Software can be used, but must be implemented with care. The number of latent errors that may exist in software is still unpredictable (non-random) and software failure tolerance cannot be verified in the same way as hardware. As the software development process matures, hopefully the software's reliability will improve. When possible NASA allows software to be only one of two hazard controls for catastrophic hazards.

Table 2-3 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 on hazardous commands
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
Operator	Operator	Two crew members, one commanding the other monitoring

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.

2.3 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. Remember, at this phase, only the highest level of definition is available for what functions the software will perform and these may change as development proceeds.

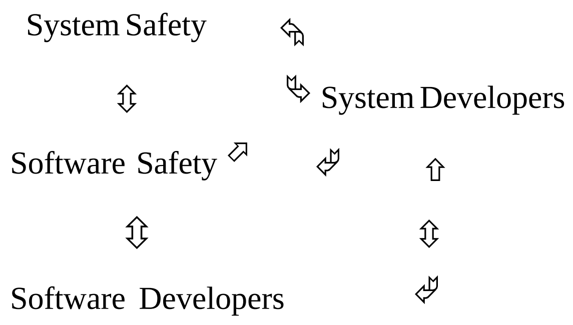
Many analysis and development techniques exist for safety critical software, and there is no "one size fits all" approach. The extent of software safety effort required must be planned based on the degree of safety risk of the system as discussed in Table 2-2 Hazard Prioritization - System Risk Index (page 8) which identifies three levels of software safety effort: full, moderate and minimum.

Section 3, SOFTWARE SAFETY PLANNING, describes these three levels of software safety effort, and presents guidelines on which analysis and development techniques may be selected for each. Section 4, SAFETY CRITICAL SOFTWARE DEVELOPMENT, then describes development tasks and techniques in detail. Section 5, SOFTWARE SAFETY ANALYSIS, describes the analysis tasks and techniques in detail.

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 2.1.1.2, determination of the level of safety risk inherent in the system was presented. Section 2.2.1 relates the system safety risk to the software. Then, Section 3.3, 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, flows those safety requirements to the software developers, and monitors 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.



The analysis and development tasks follow the software development lifecycle as follows.

3.1 Software Development Lifecycle Approach

Table 3-1 NASA Software Lifecycle - Reviews and Documents (page 20)shows the typical NASA software waterfall design lifecycle phases and lists the reviews and deliverable project documents required at each lifecycle 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 lifecycle 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 lifecycle phase in which they are conducted or produced.

In rapid prototyping environments, or spiral lifecycle 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 development must work closely together to coordinate the best time to perform each round of analysis.

Table 3-1 NASA Software Lifecycle - Reviews and Documents

LIFECYCLE PHASES	MILESTONE REVIEWS	SOFTWARE SAFETY TASKS	DOCUMENTS
Software Concept and Initiation (Project System and Subsystem Requirements and Design Development)	SCR - Software Concept Review Software Management Plan Review Phase-0 Safety Review	Scoping Safety Effort 2.1.2 PRELIMINARY HAZARD ANALYSIS 2.1.3 (PHA) Phase 0/1Safety Reviews Hazards Tracking and Problem Resolution	Software Management Plan Software Systems Safety Plan Software Configuration Management Plan Software Quality Assurance Plan Risk Management Plan
Software Requirements	SRR - Software Requirements Review Phase-1 Safety Review	2.2 SAFETY REQUIREMENTS FLOWDOWN 4.2.2 Generic Software Safety Requirements 4.2.1 Development of Software Safety Requirements 5.1 Software Safety Requirements Analysis 5.1.2 Requirements Criticality Analysis 5.1.3 Specification Analysis	

LIFECYCLE PHASES	MILESTONE REVIEWS	SOFTWARE SAFETY TASKS	DOCUMENTS
Software Architectural or Preliminary Design	Software Preliminary Design Review (PDR) Phase-1/2 Safety Review	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**	Preliminary Acceptance Test Plan Software Design Specification-(Preliminary)
Software Detailed Design	Software Critical Design Review (CDR) Phase-2 Safety Review	5.3.1 **Design Logic Analysis** 5.3.6 **Software Fault Tree Analysis (SFTA)** 5.3.7 **Petri-Nets** 5.3.8 **Dynamic Flowgraph Analysis** 5.3.2 **Design Data Analysis** 5.3.4 **Design Constraint Analysis** 5.3.13 **Formal Methods and Safety-Critical Considerations** 5.3.14 **Requirements State Machines**	Final Acceptance Test Plan Software Design Specification-(Final)

LIFECYCLE PHASES	MILESTONE REVIEWS	SOFTWARE SAFETY TASKS	DOCUMENTS
Software Implementation (Code and Unit Test)	Formal Inspections, Audits. Phase-2/3 Safety Review	5.4.1 **Code Logic Analysis** 5.4.4 **Code Data Analysis** 5.4.5 **Code Interface Analysis** 5.4.8 **Code Inspection Checklists (including coding** standards) 5.4.9 **Formal Methods** 5.4.10 **Unused Code Analysis**	Formal Inspection Reports
Software Integration and Test	Test Readiness Review Phase-3 Safety Review	4.6.1 **Testing Techniques** 4.6.2 **Software Safety Testing** 4.6.3 **Test Witnessing** 5.5.1 **Test Coverage** 5.5.2 **Test Results Analysis**	Test Reports, Problem/Failure Resolution Reports.
Software Acceptance and Delivery	Software Acceptance Review Phase-3 Safety Review		Software Delivery Documents; Acceptance Data Package
Software Sustaining Engineering and Operations		(Same activities as for development)	(Update of all relevant documents)

3.2 Tailoring the Effort - Value vs Cost

This section discusses tailoring the software safety effort based on the inherent characteristics of the software and the level of hazardous risk in a system and subsystems. Sections 4 and 5 of this Guidebook describe 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 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. 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.

The scoping of software safety activities for full or partial development efforts is discussed in this section.

3.2.1 Full Scale Development Effort vs Partial Scale Development

The software safety effort interfaces with the software development activity and the system safety activity. The system development phases are separated by system design reviews. Each system design review is conducted approximately in parallel with a corresponding system safety review as shown in Table 3-1 NASA Software Lifecycle - Reviews and Documents on page 20.

The software development effort may or may not be synchronized with the system development effort. There are normally no separate safety reviews for software. Even the phased system safety reviews are normally only done for manned spacecraft systems and payloads. For small (low value) unmanned or non-flight systems there is usually only one formal software safety review conducted at the end of the development effort. Sometimes this is simply an agenda item within an overall system acceptance review, for programs with low safety risk.

The results of the software safety analysis of each software lifecycle development phase should be reported at the respective system safety review for high risk programs, or summarized at the respective milestone design review for low risk programs.

3.2.2 Oversight Required

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

System Risk Index	Degree of Oversight
1	Not applicable (Prohibited)
2	Fully independent IV & V organization, as well as in-house SQA.
3	In house SQA organization.
4	Minimal to None required.
5	None required

This level of oversight is not for mission success purposes, where the oversight can be greater or less in each case.

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 Categorizing Safety-Critical Software Sub-systems

The effects of hazards may involve dangers to people, hardware, facilities, and/or the environment. Safety protection, either software or hardware, is mandatory for humans, any type of spacecraft, experiment, or instrument hardware. The level of safety effort required by NASA for protection of hardware varies according to the classification and hazard level. Classification of an unmanned spacecraft (A to D) is based on cost and level of acceptable risk as defined in NMI 8010.1 [3]. The classification determines the required level of effort for protection of hardware.

Most aerospace products are built up from small, simple hardware components to make large, complex systems. These components usually have a small number of states and can be tested exhaustively to determine their reliability based on failure rates. Reliability of a large hardware system is determined by the reliability values of its components. Reliability and safety goals of hardware systems can usually be reached through a combination of tested design and selection of components with appropriate reliability ratings.

Achieving safety goals through use of highly reliable software is more difficult. Reliability of software is much harder to determine. Software does not wear out or break down but may have a large number of states which 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 40-50K 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 can not be exhaustively tested, the level of effort required to meet safety goals must be determined using other inherent characteristics of the system. The following characteristics have a strong influence on the ability of software developers to create reliable, safe software:

1) 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 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 than software which is only required to recognize hazardous conditions and notify a human operator to take necessary safety actions.

   This 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. 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.).

   Software Control Categories: MIL-STD-882C [4] categories 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.

MIL-STD-882C then relates these software control categories to system hazard levels via a matrix, and uses this to decide what level of analysis and test should be applied. This approach is described below in Section 3.3.

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

The number of safety-related software requirements for hazards control increases with 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!

3) The **timing criticality** of hazardous control actions.

A system with software control of hazards that can get out of control within a very short period of time will require more software/safety assurance effort than slower changing systems. 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.

The above criteria define a system according to inherent properties and mission constraints. They do not attempt to classify a system based on the severity of any of its potential hazards as described in Section 2. Scoping the effort according to system hazard risk is described in the next section.

3.3 Scoping of Software Subsystem Safety Effort

The level of required software safety effort for a system (shown in Table 3-3) is determined by its System Category, derived from Table 2-2 Hazard Prioritization - System Risk Index (Page 8), and the hazard severity level from Section 2.1.1.2 Risk Assessment (Page 8).

All levels of software safety effort should include reviews by systems safety personnel of project specifications, requirements, design, etc., of all non-safety-critical components, to determine if these have been correctly developed and have not introduced any extra safety hazards. 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 Flowdown Analysis**). Sections 3.3.1 through 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.

Alternative Prioritization Method

Section 3.2.2 (1) described the MIL-STD-882C software control categories. That standard uses those categories to prioritize software safety tasks according to the matrix that follows.

The Software Hazard Criticality Matrix.is established using the hazard categories for the rows and the Software Control Categories for the columns. The matrix is completed by assigning Software Hazard Risk Index numbers to each element just as Hazard Risk Index numbers are assigned in the Hazard Risk Assessment Matrix. A Software Hazard Risk Index (SHRI) of `1' from the matrix implies that the risk may be unacceptable. A SHRI of `2' to `4' is undesirable or requires acceptance from the managing activity. Unlike the hardware related HRI, a low index number 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 3-2 MIL-STD-882C Software Hazard Criticality Matrix

	HAZARD CATEGORY			
CONTROL CATEGORY	CATASTRO-PHIC	CRITICAL	MODERATE	NEGLIGIBLE / MARGINAL
I	1	1	3	5
II	1	2	4	5
III	2	3	5	5
III	3	4	5	5

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

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 require a formal, rigorous program of quality and safety assurance to ensure complete coverage and analysis of all requirements, design, code, and tests.

Table 3-3 Software Sub-system categories

System Category	Descriptions
I (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.
II (System Risk Index 3)	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.
III (System Risk Index 4)	No software control of safety critical functions; no reporting of hazards to 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 2-2 Hazard Prioritization - System Risk Index (Page 8).

Table 3-4 Required Software Safety Effort

SYSTEM CATEGORY See Table 3-2	HAZARD SEVERITY LEVEL from Section 2.1.1.2			
	CATASTRO-PHIC	CRITICAL	MODERATE	NEGLIGIBLE / MARGINAL
I (System Risk Index 2)	Full	Full	Moderate	Minimum
II (System Risk Index 3)	Moderate	Moderate	Moderate	Minimum
III (System Risk Index 4)	Minimum	Minimum	Minimum	Minimum

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.

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 in 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 non features/functions added, which now make the software safety critical, but a formal program of software safety is not usually necessary.

3.4 Software Safety Assurance Techniques for Software Development Phases

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 is discussed later in Section 4 (Development Tasks) and Section 5 (Analysis Tasks).

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 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 3-5 Software Requirements Phase through Table 3-11 Software Module 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 (PESs). 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 lifecycle phases of safety-critical software development. For this guidebook, the Required Software Safety Efforts values displayed in Table 3-4 Required Software Safety Effort (page 30), will determine which development activities are required for each level of effort.

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:

M	-	Mandatory
HR	-	Highly Recommended
R	-	Recommended
NR	-	Not Recommended

Most of the "Not Recommended" entries result from consideration of time and cost in relation to the required level of effort. A mixture of entries marked as "Recommended" may be performed if extra assurance of safety 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. The final list of techniques to be used on any project should be developed jointly by negotiations between project management and safety assurance.

All the following tables, Table 3-5 Software Requirements Phase through Table 3-11 Software Module Testing , list software development, safety and assurance activities which should be implemented in the stated phases of development.

Life Cycle Phase	Tasks and Priorities	How To: Development Tasks	How To: Analysis Tasks
Concept Initiation	Table 3-5 Software Requirements Phase	Section 4.1	Section 5.1
Software Requirements	Table 3-5 Software Requirements Phase	Section 4.2	Section 5.1
Software Architectural Design	Table 3-5 Software Requirements Phase	Section 4.3	Section 5.1
Software Detailed Design	Table 3-7 Software Detailed Design Phase	Section 4.4	Section 5.3
Software Implementation	Table 3-8 Software Implementation Phase	Section 4.5	Section 5.1
Software Test	Table 3-9 Software Testing Phase Table 3-10 Dynamic Testing Table 3-11 Software Module Testing	Section 4.6	Section 5.1

Table 3-5 Software Requirements Phase

TECHNIQUE	SAFETY EFFORT LEVEL		
	MIN	**MOD**	**FULL**
2.1 Preliminary Hazard Analysis (PHA)	M	M	M
5.1.1 **Software Safety Requirements Flowdown Analysis**	R	HR	HR
5.1.1.1 **Checklists and cross references**	HR	HR	HR
5.1.2 **Requirements Criticality Analysis**	NR	R	HR
4.2.2 **Generic Software Safety Requirements**	R	HR	M
5.1.3 **Specification Analysis**	NR	R	HR
4.2.4 **Formal Inspections of Specifications**	R	M	M
5.1.5 **Timing, Throughput And Sizing Analysis**	HR	M	M

Key:	M	=	Mandatory
	HR	=	Highly Recommended
	R	=	Recommended
	NR	=	Not Recommended

Table 3-6 Software Architectural Design Phase

TECHNIQUE	SAFETY EFFORT LEVEL		
	MIN	**MOD**	**FULL**
5.2.1 **Update Criticality Analysis**	NR	R	HR
5.2.2 **Conduct Hazard Risk Assessment**	R	HR	M
5.2.3 **Analyze Architectural Design**	HR	M	M
5.2.4.1 **Interdependence Analysis**	R	HR	M
5.2.4.2 **Independence Analysis**	HR	M	M

Table 3-7 Software Detailed Design Phase

TECHNIQUE	SAFETY EFFORT LEVEL		
	MIN	**MOD**	**FULL**
5.3.6 **Software Fault Tree Analysis (SFTA)**	NR	R	M
5.3.7 **Petri-Nets**	R	HR	M
5.3.14 **Requirements State Machines**	R	HR	M
5.3.2 **Design Data Analysis**	R	HR	M
5.3.3 **Design Interface Analysis**	R	HR	M
5.3.10 **Measurement of Complexity**	R	HR	M
5.3.4 **Design Constraint Analysis**	R	HR	M
5.3.11 **Safe Subsets of Programming languages**	R	HR	M
5.3.13 **Formal Methods and Safety-Critical Considerations**	NR	R	HR
5.3.15 **Formal Inspections**	HR	M	M
Requirements translation to HOL	NR	R	HR
Emulation of HOL Statements	NR	R	HR
Static Code Analysis	NR	R	HR
Ada Emulation	NR	R	HR

Table 3-8 Software Implementation Phase

TECHNIQUE	SAFETY EFFORT LEVEL		
	MIN	MOD	FULL
5.4.1 **Code Logic Analysis**	R	HR	M
5.4.4 **Code Data Analysis**	R	HR	M
5.4.5 **Code Interface Analysis**	R	HR	M
5.4.10 **Unused Code Analysis**	R	HR	M
5.4.8 **Code Inspection Checklists (including coding** standards)	R	HR	M
5.4.9 **Formal Methods**	NR	HR	HR

Table 3-9 Software Testing Phase

TECHNIQUE	SAFETY EFFORT LEVEL		
	MIN	MOD	FULL
Testing Defensive Programming	NR	HR	M
Boundary Value Tests	R	HR	M
Error Guessing	NR	NR	R
Test Coverage Analysis	R	HR	M
Functional Testing	M	M	M
Fagan Formal Inspections (Test Plans)	HR	HR	M
Reliability Modeling	NR	HR	HR
Checklists of Tests	R	HR	HR

Table 3-10 Dynamic Testing

TECHNIQUE	SAFETY EFFORT LEVEL		
	MIN	MOD	FULL
Typical sets of sensor inputs	HR	M	M
Test specific functions	HR	M	M
Volumetric and statistical tests	R	HR	HR
Test extreme values of inputs	R	M	M
Test all modes of each sensor	R	M	M
Path testing	R	M	M
Every statement executed once	HR	M	M
Every branch tested at least once	HR	M	M
Every predicate term tested	R	HR	M
Every loop executed 0, 1, many times	R	M	M
Every path executed	R	HR	M
Every assignment to memory tested	NR	HR	HR
Every reference to memory tested	NR	HR	HR
All mappings from inputs checked	NR	HR	HR
All timing constraints verified	R	M	M
Test worst case interrupt sequences	R	R	NR

Test significant chains of interrupts	R	R	NR
Test Positioning of data in I/O space	HR	M	M
Check accuracy of arithmetic	NR	HR	M
All modules executed at least once	M	M	M
All invocations of modules tested	HR	M	M

Table 3-11 Software Module Testing

TECHNIQUE	SAFETY EFFORT LEVEL		
	MIN	MOD	FULL
Simulation (Test Environment)	R	HR	M
Load Testing (Stress Testing)	HR	M	M
Boundary Value Tests	R	HR	M
Test Coverage Analysis	R	HR	M
Functional Testing	M	M	M
Performance Monitoring	R	HR	M
Formal Progress Reviews	R	M	M
Reliability Modeling	NR	HR	HR
Checklists of Tests	R	HR	HR

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 are intended to carry out the cardinal safety rule and its corollary that no single event or action shall be allowed to initiate a potentially hazardous event 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 lifecycle phase involves system level requirements and design development.

Although most project work during this phase is concentrated on the subsystem level, software development has several tasks that must be initiated. These include the creation of important software documents and plans 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:

Document	**Software Safety Section**
System Safety Plan,	Include software as a subsystem, identify tasks.
Software Concepts Document,	Identify safety critical processes.
Software Management Plan, and Software Configuration Management Plan,	Coordination with systems safety tasks, flowdown incorporation of safety requirements. Applicability to safety critical software.
Software Security Plan, and the	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 2.1 PHA page 4).

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 flowdown to be consistent with the software safety requirements flowdown.

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

Software Requirements Document (SRD)

Safety-related requirements must be clearly identified in the SRD.

Software Interface Specification (SIS) or Interfaces Dontrol Document ICD)

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(page 41). Specific software safety requirements are system unique functional capabilities or constraints which are identified in three ways:

1) Through top down analysis of system design requirements (from specifications):

   The system requirements may identify system hazards up-front, and specify which system functions are safety critical. The (software) safety organization participates or leads the mapping of these requirements to software.

2) From the Preliminary Hazard Analysis (PHA):

   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.

3) Through bottom up analysis of design data, (e.g. flow diagrams, FMEAs, fault trees etc.):

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

4.2.1.1 Safety Requirements Flowdown

Generic safety requiredments 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 Flowdown Analysis, verifies that the safety requirements have been properly flowed into the specifications.

4.2.2 Generic Software Safety Requirements

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

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

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

NSTS 19943 Command Requirements and Guidelines for NSTS Customers

STANAG 4404 (Draft) NATO Standardization Agreement (STANAG) Safety Design Requirements and Guidelines for Munition Related Safety Critical Computing Systems

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.

AFISC SSH 1-1 System Safety Handbook - Software System Safety, Headquarters Air Force Inspection and Safety Center.

EIA Bulletin SEB6-A System Safety Engineering in Software Development (Electrical Industries Association)

Underwriters Laboratory - UL 1998 Standard for Safety - Safety-Related Software, January 4th, 1994

NUREG/CR-6263 MTR 94W0000114 High Integrity Software for Nuclear Power Plants, The MITRE Corporation, for the U.S. Nuclear Regulatory Commission.

GENERIC SOFTWARE SAFETY REQUIREMENTS FROM MSFC

1. The failure of safety critical software functions shall be detected, solated, and recovered from such that catastrophic and critical hazardous events are prevented from occurring.
2. Software shall perform automatic Failure Detection, Isolation, and Recovery (FDIR) for identified safety critical functions with a time to criticality under 24 hours.
3. Automatic recovery actions taken shall be reported to the crew, round, or controlling executive. There shall be no necessary response from crew or ground operators to proceed with the recovery action.
4. The FDIR switchover software shall be resident on an available, non-failed control platform which is different from the one with the function being monitored.
5. Override commands shall require multiple operator actions.

6. Software shall process the necessary commands within the time to criticality of a hazardous event.
7. Hazardous commands shall only be issued by the controlling application, or by the crew, ground, or controlling executive.
8. 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.
9. 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.
10. 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.
11. Software shall make available status of all software controllable inhibits to the crew, ground operators, or the controlling executive.
12. Software shall accept and process crew, ground operator, or controlling executive commands to activate/deactivate software controllable inhibits.
13. Software shall provide an independent and unique command to control each software controllable inhibit.
14. Software shall incorporate the capability to identify and status each software inhibit associated with hazardous commands.
15. Software shall make available current status on software inhibits associated with hazardous commands to the crew, ground operators, orcontrolling executive.
16. All software inhibits associated with a hazardous command shall have a unique identifier.
17. Each software inhibit command associated with a hazardous command shall be consistently identified using the rules and legal values.
18. 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.
19. Software shall have the ability to resume control of an inhibited operation after deactivation of a software inhibit associated with a hazardous command.
20. The state of software inhibits shall remain unchanged after the execution of an override.
21. Software shall provide error handling to support safety critical functions.

22 Software shall provide caution and warning status to the crew, ground operators, or the controlling executive.

23. Software shall provide for crew/ground forced execution of any automatic safing, isolation, or switchover functions.
24. Software shall provide for crew/ground forced termination of any automatic safing, isolation, or switchover functions.
25. Software shall provide procession for crew/ground commands in return to the previous mode or configuration of any automatic safing, isolation, or switchover function.
26. Software shall provide for crew/ground forced override of any automatic safing, isolation, or switchover functions.
27. Software shall provide fault containment mechanisms to prevent error propagation across replaceable unit interfaces.
28. 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.
29. 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.
30. Software (including firmware) POST utilized within any replaceable unit or component shall terminate in a safe state.
31. Software shall initialize, start, and restart replaceable units to a safe state.
32. 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.
33. Software shall require two independent operator actions to initiate or terminate a system function that could result in a critical hazard.
34. Software shall require three independent operator actions to initiate or terminate a system function that could result in a catastrophic hazard.
35. Operational software functions shall allow only authorized access.
36. Software shall provide proper sequencing (including timing) of safety critical commands.
37. Software termination shall result in a safe system state.
38. 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.

39. 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.
40. 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).
41. The software shall notify crew, ground, or the controlling executive during or immediately after execution of an automated hazardous or safing process.
42. Unused or undocumented codes shall be incapable of producing a critical or catastrophic hazard.
43. All safety critical elements (requirements, design elements, code modules, and interfaces) shall be identified as "safety critical."
44. An application software set shall ensure proper configuration of inhibits, interlocks, and safing logic, and exception limits at initialization.

4.2.2.1 Fault and Failure Tolerance/Independence

Most NASA space systems employ failure 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 effect 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, or 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 be considered ‘faults’, unless there are too many of them and the system should be ‘fault’ tolerant. 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 looming system failure within its area. 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.
This leads to 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 possibly

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, while if major failures can be detected, isolated, stopped or recovered from, it is presumed that this would require less work and be simpler.
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) 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 below.

“Must-Not-Work Functions” (MNWFs) 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 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.

4.2.2.2 Hazardous Commands

Appendix-A.2 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 (Para 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.2 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. The *USS Thresher* submarine was lost with all hands due to a hardware interlock preventing engine start-up without preheat. During the Apollo-13 mission many unplanned command sequences and procedures were developed in-flight to overcome the emergency situation. For example, the nominal Lunar Module power up sequence timeline could not be completed before the Command Module battery power expired, so a different (shorter) sequence was used.

4.2.2.3 Coding Standards

Coding Standards, a class of generic software requirements, are, in practice, “safe” subsets of programming languages. These are needed because most compilers can be unpredictable in how they work. For example, dynamic memory allocation is predictable. In applications where some portions of memory are safety critical, it is important to control which memory elements are assigned in a particular compilation process; the defaults chosen by the compiler might be unsafe. Some attempts have been made at developing coding safety standards (safe subsets). These are further discussed later in Section 4.5 Software Implementation (page 62).

4.2.2.4 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.

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

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

Control system design can define timing requirements. Based on the established body of classical and modern dynamic control theory, such as dynamic control system design, and multivariable design in the s-domain (Laplace transforms) for analog continuous processes. Systems engineers are responsible for overall control system design. Computerized control systems use sampled data (versus continuous data). Sampled analog processes should make use of Z-transforms to develop difference equations to implement the control laws. This will also make most efficient use of real-time computing resources.[1]

Sampling rates should be selected with consideration for noise levels and expected variations of control system and physical parameters. For measuring signals which are not critical, the sample rate should be at least twice the maximum expected signal frequency to avoid aliasing. For critical signals, and parameters used for closed loop control, it is generally accepted that the sampling rate must be much higher, at least a factor of ten above the system characteristic frequency is customary.[1]

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

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

Quantization: Digitized systems should select **wordlengths** long enough to reduce the effects of quantization noise to ensure stability of the system [10]. Selection of wordlengths and floating point coefficients should be appropriate with regard to the parameters being processed in the context of the overall control system. Too short wordlengths can result in system instability and misleading readouts. Too long wordlengths 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 accomodate this. Also, check timing clock reference datum, synchronization and accuracy (jitter). Analyze task scheduling (e.g., with Rate Monotonic Analysis (RMA)).

4.2.3 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 & Design (OOA & OOD) and Formal Methods. To date, the most popular analysis methods have been Functional Decomposition, Data Flow (or Structured Analysis), and Information Modeling. OOA actually incorporates some of the techniques of all of these within its method, at lower levels, once the system is cast into objects with attributes and services. In the 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 lifecycle 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 DFD organization into an implementable format.

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."

4.2.3.1 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 lifecycle 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 effecting the rest of the system.

OOA incorporates the principles of abstraction, information hiding, inheritance, and a method of organizing the problem space by using 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 verses 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 thier 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.

4.2.3.2 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.”

While Formal Methods (FM) are not widely used in NASA or US industry, FM has gained some acceptance in Europe. 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 lifecycle cost by eliminating many costly defects prior to coding.

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].

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

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:

1. Testing (Lots of it)
2. Design Diversity (i.e. software fault-tolerance: N-version programming, recovery blocks, etc.)

3. Fault/Failure Avoidance (i.e. formal specification/verification, automatic program synthesis, reusable modules)

The problem with life testing is that in order to measure ultrareliability 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 correlations. This is infeasible as well---it requires as much testing time as life-testing the system because the correlations 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.

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 predicte 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 Formal Inspections of Specifications

Formal inspections and formal analysis are different. Formal inspections are otherwise known as Fagan Inspections, named after John Fagan of IBM who devised the method. 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]. FIs should be performed within every major step of the software development process.

Formal Inspections, while valuable within each design phase or cycle, have the most impact when applied early in the life of a project, especially the requirements specification and definition stages of a project. 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 FIs.

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 finding 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 4-1 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

4.3 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
In addition, 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 4.2.2.1 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 MWFs and MNWFs, certain Safety Critical Computer Software Components (SCCSCs) must be independent of each other, this usually means 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

N-Version programming

Recovery blocks

Resourcefulness

Abbott-Neuman Components

Self-Checks.

These are discussed in more detail below.

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, SSCSCs 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.

One approach is to establish **"Fault Containment Regions"** (FCRs) to prevent propagation of software faults. This attempts to prevent fault propagation such as: from non-critical software to SCCSCs; from one redundant software unit to another, or from one SCCSC to another. Techniques known as "firewalling" or partitioning should be used to provide sufficient isolation of FCRs to prevent hazardous fault propagation.

FCRs 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 FCRs is to host them on different/ independent hardware processors. Sometimes it is acceptable to have independent FCRs hosted on the same processor depending on the specific hardware configuration. For example, if the FCRs are stored in separate memory chips and they are not simultaneously or concurrently multi-tasked 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. FCRs are defined in reference [2], SSP 50038 Computer Based Control System Safety Requirements - International Space Station Alpha.

N-Version Programming:

Reference [26], NSTS 1700.7B 201.1e(1) stipulates the following:

"A computer system shall be considered zero failure tolerant in controlling a hazardous system (i.e., a single failure will cause loss of control), unless the system utilizes independent computers, each executing uniquely developed instruction sequences to provide the remaining two hazard controls".

This stipulation effectively mandates the use of N-Version programming in any attempt to achieve failure tolerance, at least for payloads on the NSTS. Reference [2], Section 7.3.1, 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.

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

Recovery blocks
Resourcefulness
Abbott-Neuman Components
Self-Checks

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”. However, it has worked very effectively on some occassions. Like any tool or method it must be implemented properly to produce the desired results

4.3.2 Selection of COTS and Reuse

During the architectural design phase, and even earlier, decisions are made to select COTS (Commercial Off The Shelf Software) or to reuse applications developed from other similar projects.

COTS is commonly used for operating systems, and processor microcode and devices because it is becoming prohibitively expensive to custom develop software for these. There is also a strong trend in government to use commercial products, instead of custom developing similar but much more expensive products.

Very little has been done to verify the safety of COTS, and most COTS items are “unknown” in terms of their safety integrity (i.e. there is typically no development or analysis documentation). Good prior history can be a positive indication of COTS dependability, but is not definitive. Another indication of quality is if the COTS vendor is registered under ISO 9000. However, ISO 9000 certified companies sometimes produce poor quality products; ISO 9000 certifies processes, not products.

The Software Egnineering Institute (SEI) at Carnegie Mellon University (CMU) has established a Capability Maturity Model (CMM) which defines 5 levels of maturity for software development organizations. Level-5 being the highest maturity and Level-1 the lowest. This method of classifying organizations is becoming widely accepted, and can be useful when assessing a COTS provider.

An independent COTS certification authority would be highly desirable, but so far does not exist.

There have recently been some well publicized defects in COTS on the open market. Use of COTS in safety critical systems is an area of particular concern.

Non-COTS software products (i.e. government developed), are sometimes reused and adapted for new projects. There is a need for certification and verification of libraries intended for reuse.

<u>Features to consider when selecting a real-time embedded operating system.</u>

Almost every application is unique, and there is no simple universal set of criteria. The following suggestions are fairly commonly encountered issues.

a) Memory management: operating systems which support a dedicated hardware MMU (Memory Management Unit) are superior. This guarantees protection of designated memory space. In a multi-tasking application it ensures the integrity of memory blocks dedicated to individual tasks, ensuring that tasks do not write into each others' memory space. It protects each task from errors such as bad pointers in other tasks.

b) Determinism: this 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.

c) 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.

d) Speed - more specifically, the context switching time. This control can be "full" or "lightweight".

e) Interrupt latency - how fast an interrupt can be serviced.

f) Method of scheduling:

   this can be predetermined logical sequences (verified by Rate Monotonic Analysis), or

   it can be priority based preemptive scheduling in a multi-tasking environment, or

   "Round Robin" time slice scheduling at the same priority for all tasks.

g) POSIX compliance (1003.1b/c). This is a standard used by many operating systems to permit transportability of applications between different operating systems.

h) Support for synchronization and communication between tasks.

I) Support for tools such as debuggers, ICE (In Circuit Emulation) and multiprocessor debugging. Ease of use, cost and availability of tools.

j) Support for multi-processor configuration (multiple CPUs) if required.

k) Address the language in which the operating system kernel is written, using the same criteria as selecting application programming languages. Unfortunately, most operating systems (and other COTS) are written in C or C++ which are inherently unsafe languages.

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.

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 this software test plans and procedures. Results of preliminary tests of prototype code should be evaluated and documented in the Software Development Folders (SDFs). 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

Coding checklists should be used by software developers in software coding and implementation.

The coding checklists should be generated by an overall ongoing Specification Analysis activity which began in the requirements phase, (see Section 5.1.3). 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.

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.

4.5.2 Coding Standards

Software implementation should comply with coding standards. A formal inspection should verify that the standards and checklists were followed. The software developers should use proper, safe subsets of programming languages as described in Section 5.3.11.

Defensive Programming
Hazards can be mitigated (but not controlled) using defensive programing 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.

4.5.3 Unit Level Testing

Software Integration and Test is discussed below in Section 4.6. Unit level testing begins during the software implementation phase. This is because units and modules often cannot be thoroughly tested during integration because individual module lever inputs and outputs are no longer accessible. Unit level testing can identify implementation problems which require changes to the software. For these reasons, unit level testing must be mostly completed prior to software integration.

4.6 Software Integration and Test

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. It is prudent to test first in a simulated environment, before connecting the computer to hazardous actuators such as rocket engines or ordnance. This testing, conducted in accordance with the safety test plan and procedures, will verify that the software meets safety requirements. Normally, the software developer's testing ensures that the software performs all required functions correctly. 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. 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 in 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. This contains a description of the problems encountered and recommended solutions.

4.6.1 Testing Techniques

Testing should be performed either in a controlled environment in which execution is controlled and 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. Fidelity of the simulators should be carefully assessed.

Demonstration testing executes the software on a computer and in an environment identical to the operational computer and environment.

Safety testing exercises program functions under both nominal and extreme conditions. Safety testing includes nominal, stress, performance and negative testing.

Configuration Management should act as the sole distributor of media and documentation for all acceptance tests and for delivery to [sub]system integration and testing.

Software Test Plan

The Software Test Plan, should incorporate software safety information in the following test activities:

Unit and Integration Testing: Identify tests for safety-critical interfaces. Demonstrate how they cover the safety test requirements (see Section 4.6.2 below). Specify pass/fail criteria for each safety test. Specify safety constraints and dependencies on other test sets. Describe review and reporting process for safety-critical components. List test results, problems, and liens.

Acceptance Testing: Specify acceptance test for safety-critical components. Demonstrate how tests cover the safety requirements in the SRD. Specify pass/fail criteria. Specify any special procedures, constraints, and dependencies for implementing and running safety tests. Describe review and reporting process for safety-critical components. List test results, problems, and liens.

Software Test Reports

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

Unit Testing Results: Report test results in Software Development Folders.

Integration Test Reports: Report results of testing safety-critical interfaces versus the requirements outlined in the Software Test Plan.

Acceptance Testing: Report results of testing safety-critical components versus the requirements outlined in the Software Test Plan.

Any Safety Critical findings should be used to update the hazard reports.

4.6.2 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.

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.

4.6.3 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 cannot 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 3.2 TAILORING THE EFFORT - VALUE vs COST 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.

4.6.4 COTS and GFE Testing

Safety critical Commercial Off The Shelf software (COTS) included in the system should be analyzed and tested. Commercial software includes commercially developed, commercially acquired, proprietary, reused, inherited and other software not specifically developed for the system. These analyses and tests should be performed whether this software is modified or not.

Note that most computer operating systems are COTS.

Subject any safety-critical Government Furnished (GFE) software, whether modified or not, to the same software safety analysis and testing requirements as the software that was developed under the contract which invokes this task. Existing hazards analyses and test results of unmodified GFE 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.

Correct the software to eliminate or reduce to an acceptable level of risk any safety hazards discovered during system integration testing or acceptance testing. The corrected software must be retested under identical conditions to ensure that these hazards have been eliminated, and that other hazards do not occur.

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: List any waivers of safety requirements.

  Program Set Acceptance Test Results: List results of safety tests.
- New or Changed Capabilities: List any approved change requests for safety-critical items.
- Problem Disposition: List 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 offline analog (or simulator) of the operational software system, prior to placing it into service.

5. SOFTWARE SAFETY ANALYSIS

During the software lifecycle, the software safety organization performs various analysis tasks, employing a variety of techniques. This section describes techniques which have been useful in NASA activities and some from elsewhere. The methodology for each technique is described, together with the data to be used and products generated in each case. Some discussion on the relative cost and value of each technique is provided. Absolute cost and value metrics are not available in most cases.

Techniques fall into one of two categories:

1) Top down system hazards and failure analyses, defining environment for software failure tolerance and hazards analyses, (e.g., hardware/software interactions).

2) Bottom up review of design products to identify and disposition failure modes not predicted by top down analysis. This will ensure validity of assumptions of top down analysis, and verify conformance to requirements.

Note that a typical software safety analysis activity will require both types of analyses, although not every technique described needs to be applied. Guidance is provided on the relative costs and merits of each technique to assist in the planning of software safety activities.

Results of software safety analysis are reported back to the system safety organization. (For example, new hazards or problems in complying with safety requirements which might require changes to hardware configuration).

As software controls become more defined, software hazard analyses will identify individual program sets, modules, units, etc. which are safety-critical. These analyses include:

Software Safety Requirements Analysis

Architectural Design Analysis

Code Analysis

Test Analysis

Operations & Maintenance

5.1 *Software Safety Requirements Analysis*

The requirements analysis activity verifies that safety requirements for the software were properly flowed down, and that they are correct, consistent and complete. It also looks for new hazards, and unexpected software functions.

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 which may generate new software requirements. Specification analysis is also performed to ensure consistency of requirements.

5.1.1 Software Safety Requirements Flowdown Analysis

Safety requirements are flowed down into the system design specifications as described in Section 4.2.1.1.

An area of concern in the flowdown process is incomplete analysis, and/or inconsistent analysis of highly complex systems, or use of ad hoc techniques by biased or inexperienced analysts. The most rigorous (and most expensive) method of addressing this concern is adoption of formal methods for requirements analysis and flowdown, described previously in Section 4.2.3.2 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.

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

5.1.1.1 Checklists and cross references

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

Develop a systematic checklist of software safety requirements and any hazard controls, ensuring they correctly and completely include (and cross reference) the appropriate specifications, hazard analyses test and design documents. This should include both generic and specific safety requirements as discussed in Section 4.2.1. Section 5.1.4 on form inspections lists some sources for starting a safety checklist. Develop a hazard requirements flowdown matrix which maps safety requirements and hazard controls to system/software functions and to software modules and components. Where components are not yet defined, flow to the lowest level possible and tag for future flowdown.

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. At this point the systems engineers may elect to make design changes to reduce the criticality levels or consolidate modules reducing the number of critical modules.

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

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

The methodology for Requirements Criticality Analysis is as follows:

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

At the system level: identify hardware or software items that receive/pass/initiate critical signals or hazardous commands (see 4.2.2.2).

At the software requirements level: identify software functions or objects that receive/pass/initiate critical signals or hazardous commands.

This safety activity examines the system/software requirements and design to identify unsafe conditions for resolution such as out-of-sequence, wrong event, inappropriate magnitude, incorrect polarity, inadvertent command, adverse environment, deadlocking, and failure-to-command modes.

The software safety requirements analysis considers such specific requirements as the characteristics discussed below in 5.1.2.1,Critical Software Characteristics.

The following resources are available for the Requirements Criticality Analysis:

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

1) Software Development Activities Plan [Software Development Plan] Software Assurance Plan [None], Software Configuration Management Plan [Same] and Risk Management Plan [Software Development Plan].

2) System and Subsystem Requirements [System/Segment Specification (SSS), System/Segment Design Document].

3) Requirements Document [Software Requirements Specifications].

4) External Interface Requirements Document [Interface Requirements Specifications] and other interface documents.

5) Functional Flow Diagrams and related data.

6) Program structure documents.

7) Storage and timing analyses and allocations.

8) Background information relating to safety requirements associated with the contemplated testing, manufacturing, storage, repair, installation, use, and final disposition of the system.

9) 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.

10) Historical data such as lessons learned from other systems and problem reports.

Output products are the following:

1. Table 5-1 Subsystem Criticality Matrix
2. UpdatedSafety Requirements Checklist
3. Definition of Safety Critical Requirements.

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

The results of the criticality analysis may be used to develop Formal Inspection (FI) checklists for performing the FI process described later in Formal Inspections of Specifications

5.1.2.1 Critical Software Characteristics

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

All characteristics of safety critical software must be evaluated to determine if they are safety critical. Safety critical characteristics should be controlled by requirements which 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:

1) specific limit ranges

2) out of sequence event protection requirements (e.g., if-then statements)

3) timing

4) 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.

5) voting logic

6) hazardous command processing requirements (see 4.2.2.2 Hazardous Commands)

7) fault response

8) fault detection, isolation, and recovery

9) 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	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	

Key:	IMI	=	Inadvertent Motor Ignition
	CA	=	Collision Avoidance
	ICD	=	Inadvertent Component Deployment

This matrix is an example output of a software requirements criticality analysis as described in Section 5.1.2. 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 2.1.1.2 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:

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

For the latter three techniques a large, well established body of literature exists describing in detail these methods, and many others, and background for each. Instead of reproducing those lengthy texts the reader is directed to the excellent references listed below. A brief description of each technique will be given so that the analyst can determine if further study is warranted.

Beizer, Boris, “Software Testing Techniques”, Van Nostrand Reinhold, 1990. - (Note: Despite its title, the book mostly addresses analysis techniques).

Beizer, Boris, “Software System Testing and Quality Assurance”, Van Nostrand Reinhold, 1987. (Also includes many analysis techniques).

Yourdon Inc., “Yourdon Systems Method - model driven systems development”, Yourdon Press, N.J., 1993.

DeMarco, Tom, “Software State of the Art: selected papers’, Dorset House, NY, 1990.

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:

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

5.1.3.1 Hierarchy analysis

Hierarchy analysis establishes the dependency of requirements on each other

5.1.3.2 Control-flow analysis

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

5.1.3.3 Information-flow analysis

Examines the relationship between functions and data. Identifies incorrect, missing, and inconsistent input/output specifications. Data flow diagrams are commonly used to report the results of this activity, so this tchnique 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.

5.1.3.4 Functional simulation models

Simulators are useful development tools for evaluating system performance and human interactions. One 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

5.1.4 Formal Inspections

Software Formal Inspections are an important activity for safety to participate in, especially during the requirements phase. Not only is it a very good level at which to help effect change, the safety representative can more thoroughly learn about the system and how it is supposed to work as well as be in a position to find areas of safety concern, or non-concern, early-on in the life of a project.

As part of the safety activity, a safety checklist should be created to follow when reviewing the requirements. This checklist should be based on the safety requirements discussed in Section 4.2.

The generic requirements portion of the checklist should be tailored to emphasize those 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 4-1 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.

5.1.5 Timing, Throughput And Sizing Analysis

Timing and sizing analysis for safety critical functions evaluates software requirements that relate to execution time and memory allocation. Timing and sizing analysis focuses on program constraints. Typical constraint requirements are maximum execution time and maximum memory usage. The safety organization should evaluate the adequacy and feasibility of safety critical timing 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.

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

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.

In many cases it is difficult to predict the amount of computing resources required. Hence, making use of past experience is important.

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 scratchpad space for storing interim and final results of computations. Memory estimates in early program phases can be inaccurate, and the estimates should be updated and based on prototype codes and simulations before they become realistic. Dynamic Memory Allocation can be viewed as either a practical memory run time solution or as a nightmare for assuring proper timing and usage of critical data. Any suggestion of Dynamic Memory Allocation, common in OOD, CH environments, should be examined very carefully; even in “non-critical” functional modules.

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

Address I/O for science data collection, housekeeping and control. Evaluate 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 time variations of CPU load, determine circumstances of peak load and whether it is acceptable. Consider multi-tasking effects. Note that excessive multi-tasking can result in system instability leading to “crashes”.

Sampling rates versus rates of change of physical parameters

Design criteria for this is discussed in Section 4.2.2.4 Timing, Sizing and Throughput Considerations. Analysis should address the validity of the system performance models used, together with simulation and test data, if available.

5.1.6 Conclusion

Use of 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 analysis will need to be repeated during design and code phases as the requirements may not have the depth of detail required to determine how the memory I/O will function.

The output of the Software Safety Requirements Analyses (SSRA) are used as input to follow-on software safety analyses. The SSRA shall be 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 safety requirements set, an architectural design is developed as per Section 4.3. Section 5.2 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 above 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 (SCCSCs) are identified.

Bottom-up safety analysis is performed on the architectural design to identify potential hazards, to define and analyze SCCSCs (Safety Critical Computer Software Components) and the early test plans are reviewed to verify incorporation of safety related testing.

Analyses included in the Architectural Design Phase are as follows:

Update Criticality Analysis
Conduct Hazard Risk Assessment

Analyze Architectural Design
Interdependence Analysis
Independence Analysis
Update Timing/Sizing Analysis

5.2.1 Update Criticality Analysis

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

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

Some of these modules will be safety critical, and some will not. The criticality analysis (See section 5.1.2) provides the appropriate initial criticality designation for each software function. The safety activity relates identified hazards from the following analyses previously described to the Computer Software Components (CSCs) that may affect or control the hazards.

Analysis	Guidebook Section
Preliminary Hazard Analysis (PHA)	2.1
Software Subsystem Hazard Analysis	2.3
Software Safety Requirements Analysis	5.1

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

5.2.2 Conduct Hazard Risk Assessment

The safety activity performs a system hazard risk assessment to identify and prioritize those SCCSCs that warrant further analysis beyond the architectural design level. System risk assessment of hazards as described in the NHB 1700 series of documents, consists of ranking 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 SCCSCs 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 (page 8).

5.2.3 Analyze Architectural Design

The safety activity analyzes the Architectural Design of those SCCSCs identified in the preceding paragraphs to ensure all safety requirements are specified correctly and completely in the Architectural Design. In addition, the safety activity determines where in the Architectural Design, and under what conditions unacceptable hazards occur. This is done by postulating credible faults/failures and evaluating their effects on the system. Input/output timing, multiple event, out-of-sequence event, failure of event, wrong event, inappropriate magnitude, incorrect polarity, adverse environment, deadlocking, and hardware failure sensitivities are included in the analysis.

Methods used for FMEA (Failure Modes and Effects Analysis) can be used substituting software components for hardware components in each case. A widely used FMEA procedure is MIL-STD-1629, which is based on the following steps:

a. define the system to be analyzed...,

b. construct functional block diagrams...,

c. identify all potential item and interface failure modes...,

d. evaluate each failure mode in terms of the worst potential consequences...,

e. identify failure detection methods and compensating provisions...,

f. identify corrective design or other actions to eliminate / control failure...,

g. identify impacts of the corrective change....,

h. document the analysis and summarize the problems which could not be corrected.....

Formal Inspections (described earlier), design reviews and animation/simulation augment this process.

5.2.3.1 Design Reviews

Design data is reviewed to ensure it properly reflects applicable software safety requirements. Design changes are generated where necessary.

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

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

5.2.3.2 Animation/Simulation

Simulators, prototypes (or other dynamic representations of the required functionality as specified by the design), and test cases to exercise crucial functions can be developed. Run the tests and observe the system response. Requirements can be modified as appropriate.

Documented test results can confirm expected behavior or reveal unexpected behavior. The status of critical verifications are captured by hazard reports.

5.2.4 Interface Analysis

5.2.4.1 Interdependence Analysis

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

5.2.4.2 Independence Analysis

The safety activity evaluates available design documentation to determine the independence/dependence and interdependence of SCCSCs to both safety-critical and non-safety-critical CSCs. Those CSCs that are found to affect the output SCCSCs are designated as SCCSCs. Areas where FCR (Fault Containment Region) integrity is compromised are identified.

Methodology: Map the safety critical functions to the software modules and map the software modules to the hardware hosts and FCRs. Each input and output of each SCCSC should be inspected.

Resources are definition of safety critical functions (MWF and MNWF) needing to independent (from Section 4.2.2.1), design descriptions, and data diagrams.

Design changes to achieve valid FCRs and corrections to SCCSC designations may be necessary

5.3 Detailed Design Analysis

During the Detailed Design phase, more detailed software artifacts are available, permitting rigorous analyses to be performed. Detailed Design Analyses can make use of artifacts such as 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

Software Fault Tree Analysis (SFTA)

Petri-Nets

Dynamic Flowgraph Analysis

Markov Modeling

Measurement of Complexity

Safe Subsets of Programming languages

Formal Methods and Safety-Critical Considerations

Requirements State Machines

Formal Inspections

Description of each technique is provided below. Choice of techniques in terms of cost versus effectiveness was discussed earlier in Section 3.2, TAILORING THE EFFORT - VALUE vs COST.

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.

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 affecting memory should be reviewed in conjunction with system safety.

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

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

The analysis should consider the validity and effectiveness of checksums and CRCs. The sophistication of error checking implemented should be appropriate for the predicted bit error rate of the interface. An overall system error rate should be defined, and budgeted to each interface.

Examples of interface problems:

- sender sends eight bit word with bit 7 as parity, but recipient believes bit 0 is parity.
- sender transmits updates at 10 Hz, but receiver only updates at 1 Hz.
- sender encodes word with leading bit start, but receiver decodes with trailing bit start.
- interface deadlock prevents data transfer (e.g., receiver ignores or cannot recognize "Ready To Send").
- user reads data from wrong address.
- sender addresses data to wrong address.
- In a language such as C, or C++ where data typing is not strict, sender may use different data types than reviewer expects. (Where there is strong data typing, the compilers will catch this).

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 and sizing constraints as per Section 5.1.5,

- 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 wordlength (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.

5.3.5 Rate Monotonic Analysis

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

For further details on this technique, refer to publications by Sha and Goodenough, References [22] and [25].

5.3.6 Software Fault Tree Analysis (SFTA)

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

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.

FTA is a complex subject, and is described further in Appendix-B attached to this Guidebook.

5.3.7 Petri-Nets

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

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

Some of the potential advantages of Petri-nets over other safety analysis techniques include the following:

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

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

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

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

Petri net analysis is a complex subject, and is treated in more detail in Appendix-C. A good reference on the use of Petri-Nets in safety-critical applications can be found in [19]. Most of this section, Appendix-C, and the figures used are taken from [19].

5.3.8 Dynamic Flowgraph Analysis

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

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

As an example, Figure 5-1 DFM Application to a Simple System illustrates a simple embedded system, and Figure 5-2 DFM System Model is a DFM model of that system. Models such as these are analyzed to determine how a certain state (desirable or undesirable) can be reached. This is done by developing timed fault trees which take the form of logical combinations of static trees relating the system parameters at different points in time. Figure 5-3 Example of DFM Analysis shows a set of timed fault trees for the example DFM model. The resulting information concerning the hardware and software states that can lead to certain events of interest can then be used to increase confidence in the system, eliminate unsafe execution paths, and identify testing criteria for safety critical software functions.

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

Figure 5-1 DFM Application to a Simple System

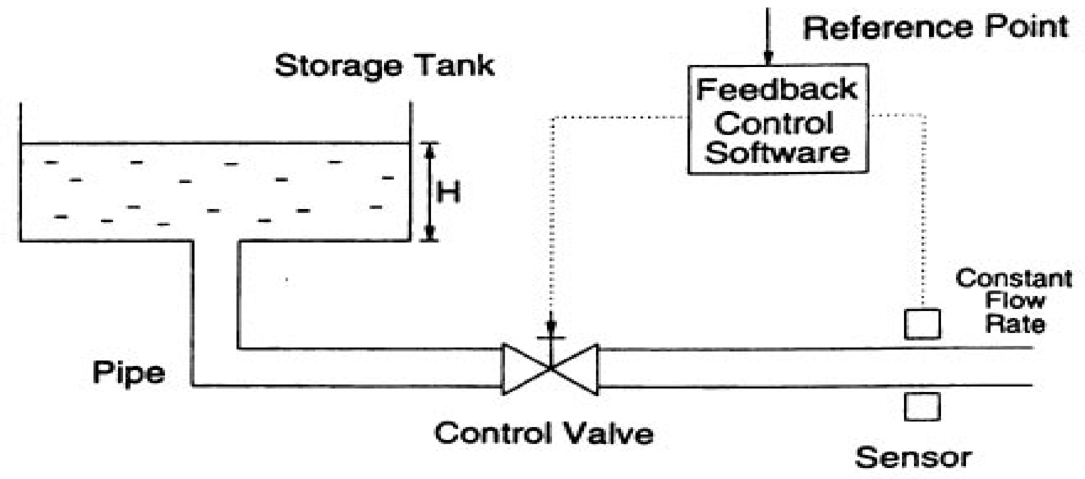
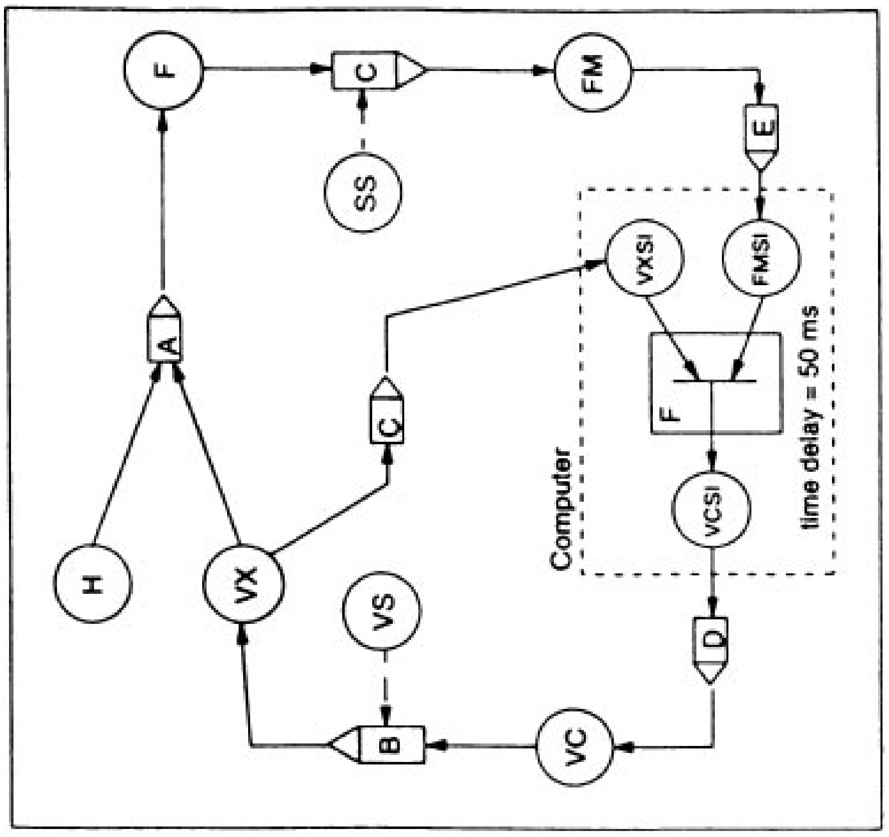
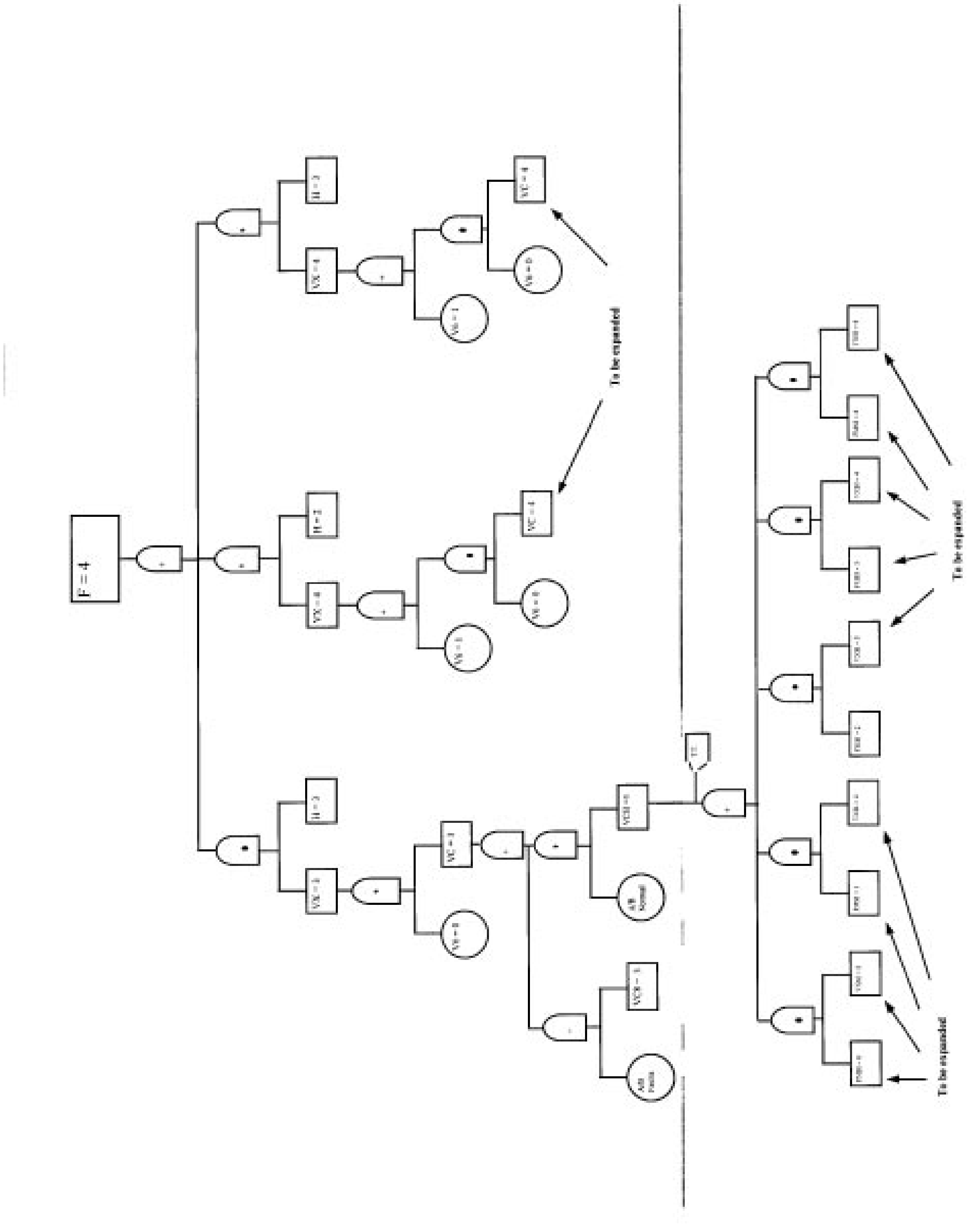


Figure 5-2 DFM System Model



H - Height of Water in the Storage Tank
F - Downstream Flowrate
FM - Measured Flowrate
VX - Position of the Valve
VC - Commanded Valve Position
VS - State of the Valve
SS - State of the Flowrate Sensor
VCSI - Software Image of the Valve Position Command
VXSI - Software Image of the Valve Position
FMSI - Software Image of the Flowrate

Figure 5-3 Example of DFM Analysis



5.3.9 Markov Modeling

Markov Modeling techniques were developed for complex systems and some analysts use 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 modelling 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.

5.3.10 Measurement of Complexity

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

Linguistic, structural, and combined metrics exist for measuring the complexity of software and while discussed below briefly, it is suggested that the developer utilize the following references from Section 5.1.3 for further guidance for these techniques.

> Beizer, Boris, “Software Testing Techniques”, Van Nostrand Reinhold, 1990. - (Note: Despite its title, the book mostly addresses analysis techniques).
>
> Beizer, Boris, “Software System Testing and Quality Assurance”, Van Nostrand Reinhold, 1987. (Also includes many analysis techniques).
>
> Yourdon Inc., “Yourdon Systems Method - model driven systems development”, Yourdon Press, N.J., 1993.
>
> DeMarco, Tom, “Software State of the Art: selected papers’, Dorset House, NY, 1990.

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

Resources are the detailed design, high level language description, source code, and automated complexity measurement tool(s).

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

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

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

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

5.3.11 Safe Subsets of Programming languages

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

The use of software to control safety-critical processes is placing software development environments (i.e. languages, compilers, utilities, etc.) under increased scrutiny. When computer languages are taught, students are seldom warned of the limitations and insecurities that the environment possesses. An insecurity is a feature of a programming language whose implementation makes it impossible or extremely difficult to detect some violation of the language rules, by mechanical analysis of a program's text. The computer science profession has only recently focused on the issues of the inherent reliability of programming environments for safety-critical applications.

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

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

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

5.3.11.1 Insecurities Common to All Languages

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

- Probably the most common misuse in practically all programming languages is that of uninitialized variables. This mistake is very hard to catch because unit testing will not flag it unless explicitly designed to do so. The typical manifestation of this error is when a program that has been working successfully is run under different environmental conditions and the results are not as expected.
- Calls to 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.
- The order of evaluation of operands when side effects from function calls modify the operands is generally dismissed as poor programming practice but in reality is an issue that is poorly defined (no standard of any type has been defined) and arbitrarily resolved by implementors of language compilers.

5.3.11.2 Method of Assessment

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

When evaluating a language, the following questions should be asked of the language as a minimum:

1) Can it be shown that the program cannot jump to an arbitrary location?
2) Are there language features that prevent an arbitrary memory location from being overwritten?

3) Are the semantics of the language defined sufficiently for static code analysis to be feasible?

4) Is there a rigorous model of both integer and floating point arithmetic within the standard?

5) Are there procedures for checking that the operational program obeys the model of the arithmetic when running on the target processor?

6) Are the means of typing strong enough* to prevent misuse of variables?

7) Are there facilities in the language to guard against running out of memory at runtime?

8) Does the language provide facilities for separate compilation of modules with type checking across module boundaries?

9) Is the language well understood so designers and programmers can write safety-critical software?

10) 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

5.3.11.3 C Language

The C language is extremely popular because of its flexibility and support environment. There is a wide variety of mature and inexpensive development and verification tools available. Also, the pool of experienced vendors and personnel is quite large. However, its 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. Many safety-critical applications have been coded in C and function without serious flaws. The implications of choosing a language with characteristics that do not provide the checks present in Pascal, Ada or Modula 2 places the burden of a thorough verification of code and input data sequences on the developers.

The characteristic of C that decreases its reliability is that C is not a strongly typed language. Data typing in C 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. The other 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 medium for expression of programs. 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.

Floating Point Arithmetic in C:

The ANSI C standard does not mandate any particular implementation for floating point arithemtic, 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 wordlength round-off.

Miscellaneous problems:

The following quotations were taken from Imperial College, London, UK, world wide web home page Dictionary of Computer Terminology, compiled by Denis Howe. It contains graphic descriptions of common problems with C.

“**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). Lisp fans enjoy pointing out that this can't happen in *their* favorite language, which eschews precedence entirely, requiring one to use explicit parentheses everywhere.

“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.

“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.

“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 marvellous 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.

http://wombat.doc.ic.ac.uk/

The Free On-line Dictionary of Computing

The above quotation was 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).

Additional guidelines on C programming practices are described in the book “Safer C:” (Reference [25], and also in [27] and [28]).

5.3.11.4 Pascal Language

The Pascal language is well defined and is considered a suitable medium for safety-critical applications provided restrictions are made. SPADE Pascal (SPADE PASCAL is a commercially available product and is used here only as an example to illustrate a technical point) 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 to be considered a serious candidate for safety-critical applications.

5.3.11.5 Ada Language

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 Ada 83. 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 Ada 95. Ada 95 corrects many of the safety deficiencies of Ada 83.

5.3.11.5.1 Ada 83

The Ada language still has many ambiguities in its definition and this is probably the largest single reliability issue. The semantics of Ada 83 were not well defined and, as of 1989, about 800 queries had been raised to DoD over the definition. A subset of Ada 83 called SPADE Ada addresses most of the requirements for a language used for safety-critical applications. The Ada language has strong typing that is more restrictive than C's or Pascal's. The GOTO statement was only provided for porting of Fortran programs to Ada and not for regular use. Exception handling is a feature of the language and users can write extremely detailed handlers. Exception handlers have their drawbacks for safety-critical systems. Writing an exception handler may result in an overall lower reliability of the program if it is not written properly. Carre, in reference [16], makes the point that proving an exception handler may be more difficult than writing an exception free program and proving it to be so.

5.3.11.5.2 Insecurities in the Ada 83 Language

When evaluating a programming language, a thorough analysis of the known insecurities is necessary to determine if the language is adequate for safety-critical programming. Problems caused by an incorrect implementation of the Ada standard by a compiler vendor can only be mentioned in this entry in a generic fashion. An obvious safety-related concern is if developing organizations are aware of problems generic to the Ada language and have a vendor's problem report list. A user who is considering selecting a compiler for a safety-critical application must be aware that all implementations, even though they have passed the Ada ACVC (Ada Compiler Validation Capability) will contain defects. To evaluate how one compiler stands up against another a user should request the history of defects, both those found by customers and by the vendor, along with typical and worst case delays in fielding fixes. This history, along with a rough customer base figure, will give an approximate idea of how extensive and rigorous a test bed the compiler has been subjected to. If a vendor cannot, or will not, supply this kind of information then even if the compiler has been used on a similar project before hand, that vendor should not be considered a viable choice.

The insecurities of the Ada language are classified and briefly explained in the Ada Language Reference Manual (LRM) 1.6 [17].

5.3.11.6 Subset Ada

SPADE Ada is based upon the "Pascal core of Ada" and is supplemented by packages, private types, functions with structured values and with some restrictions to the library system. SPADE Ada requires annotations to convey certain semantic information necessary for the static code analysis tool. Annotations are preceded by the Ada comment character '--' (so compilers don't interpret what follows) but the SPADE tool uses the annotated declarations to extract information required for analysis and proofs. Imposing the discipline of a subset of Ada is still valuable, even if the SPADE Ada code checker is not available, because the behavior of the restricted language constructs is predictable, many insecurities of the language have been eliminated because the features they were associated with are not used, and manual verification of the code (proofs by hand) can be performed if required.

5.3.11.7 C++ Language

The C++ programming language is an extension (superset) of the C programming language discussed above (Section 5.3.11.3). The main difference between C and C++ is that C++ provides an Object Oriented environment, allowing structured design to be applied. This is a positive development, since structured design, as discussed earlier, reduces design errors prior to coding. C++ is also more strongly typed than C. However, C++ suffers from many of the same drawbacks as C.

A "safe subset" of C++ does not presently exist.

Some good practices suggested to mitigate the inherent insecurities of C++ are as follows:

a) 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.);

b) only rely on fully abstract classes, passing interface but not implementation (suggestion by Szyperski at 1995 Safety through Quality Conference - NASA-KSC [30]).

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.

5.3.11.8 Programming Languages: Conclusions

A technique for evaluating safety-critical languages has been described along with other considerations for implementers and for choosing a vendor. The Ada subset we have described is suitable for safety-critical systems. Although a Pascal subset is suitable it was not described because Ada is rapidly becoming the most common language on new projects. The choice of "C" is to be avoided for our domain of interest because the language lacks the features that permit robust, reliable programming. The Ada subset minimizes the insecurities that were discussed in this entry by restricting the misuse of certain features. The objective of the subset is to define an unambiguous syntax so that the semantics can be used for proofs if necessary. All Ada implementations have defects and implementers of safety-critical systems should review the historical defect list to verify that the problems have been addressed and fixed in a timely fashion

5.3.12 Miscellaneous Problems Present in Languages Other than C

The following quotations were also 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, or by use of a higher-level language, such as Lisp, which employs a garbage collector.

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 ounters 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; see grind crank). 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."

5.3.13 Formal Methods and Safety-Critical Considerations

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

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

On a qualitative level the following list identifies different levels of application of assurance methods in software development [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) <u>Formal development down to object code</u> requires that formal mathematical proofs be carried out on the executable code.

2) <u>Formal development down to source code</u> requires that the formal specification of the system undergo proofs of properties of the system.

3) <u>Rigorous development down to source code</u> 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) <u>Structured development to requirements analysis then rigorous development down to source code</u> 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) <u>Structured development down to source code</u> 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) <u>Ad hoc</u> techniques encompass all of the non-structured and informal techniques (i.e. hacking, code a little then test a little).

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

5.3.14 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, (eg, Object Oriented Design see Section 4.2.1 and Formal Methods (see Section 4.2.3.2)).

Whether or not Formal Methods are used to develop a system, a high level RSM can be used to provide a view into the architecture of an implementation without being engulfed by all the accompanying detail. Semantic analysis criteria can be applied to this representation and to lower level models to verify the behavior of the RSM and determine that its behavior is acceptable.

The analysis criteria will be listed in a section below and in subsequent sections because they are applicable at practically every stage of the development life cycle.

5.3.14.1 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].

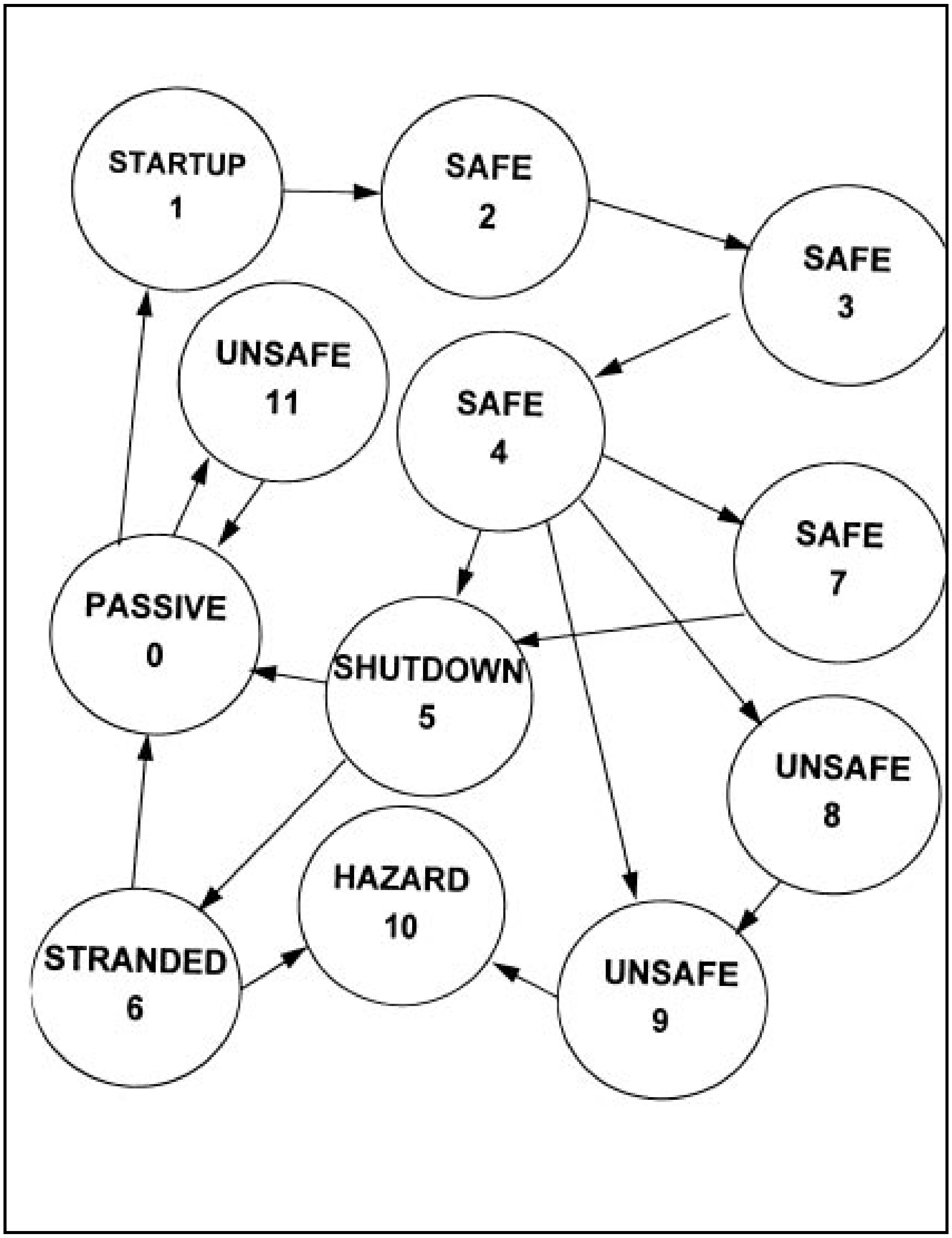
The state machine models should be built to abstract different levels of hierarchy. The models are partitioned in a manner that is based on considerations of size and logical cohesiveness. An uppermost level model should contain at most 15 to 20 states; this limit is based on the practical consideration 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 timeout may be specified and a state transition to an unsafe or passive state may be defined.

Figure 5-4 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 timeout 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.

5.3.14.2 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.

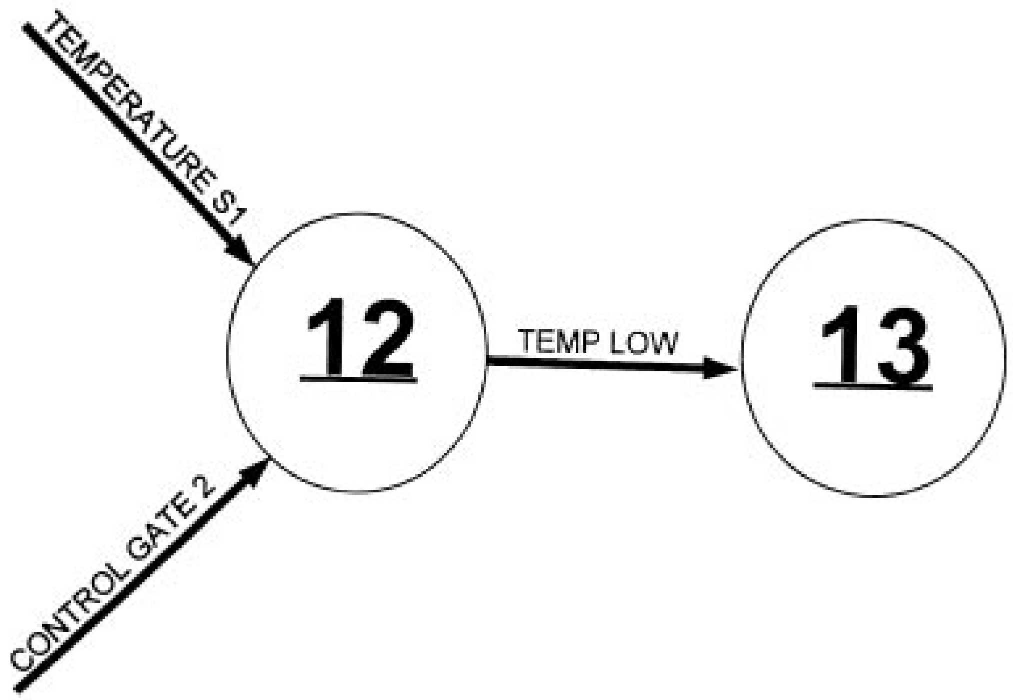
5.3.14.3 Input/Output Variables

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].

5.3.14.4 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 5-5 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 $\leq$ Threshold when Control Gate 2 is high.

5.3.14.5 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 timeout, 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 (see Figure 5-4).

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.

5.3.14.6 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].

5.3.14.7 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:

1) Error handling.
2) Logging and generation of warning messages.
3) Reduction of processing load (execute only core functionality).
4) Masking of nonessential interrupts.
5) Reduction of accuracy and/or response time.
6) Signals to external world to slow down inputs.
7) 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.

5.3.14.8 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.

5.3.14.9 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.

5.3.14.10 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 lifecycle 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.

5.3.15 Formal Inspections

The process of Formal Inspection begun in previous design phases (e.g., Section 4.2.4) should continue.

New software artifacts are now available, and new inspections checklists should be developed appropriate for the new artifacts, and building on lessons learned earlier in the design lifecycle. The artifacts of this design phase (e.g., PDL, prototype code) are more detailed than previous phases.

5.4 Code Analysis

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

Code permits real measurements of size, complexity and resource usage.

Code Analyses include the following:

1) Code Logic Analysis

2) Software Fault Tree Analysis (SFTA)

3) Petri-Nets

4) Code Data Analysis

5) Code Interface Analysis

6) Measurement of Complexity

7) Code Constraint Analysis

8) Safe Subsets of Programming languages

9) Formal Methods and Safety-Critical Considerations

10) Requirements State Machines

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

Each of these analyses, contained in this section, should be undergoing their second iteration, since they should have all been applied previously to the code-like products (PDL) of the detailed design.

There are some commercial tools available which perform one or more of these analyses in a single package. These tools can be evaluated for their validity in performing these tasks, such as logic analyzers, and path analyzers. However, unvalidated COTS tools, in themselves, cannot generally be considered valid methods for formal safety analysis. COTS tools are often useful to reveal previously unknown defects.

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

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.

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.

Continue the following analysis as described previously in Section 5.3 DETAILED DESIGN ANALYSIS:

5.4.2 Fault trees

See 5.3.6.

5.4.3 Petri nets.

See 5.3.7.

5.4.4 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 interferring with safety critical data. Also, check the "typing" of safety critical declared variables.

5.4.5 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 5.4.7 Design Constraint Analysis below.

5.4.6 Measurement of Complexity

As a goal, software complexity should be minimized to reduce likelihood of errors. Complex software also is more likely to be unstable, or suffer from unpredictable behavior.

Modularity is a useful technique to reduce complexity. Complexity can be measured via McCabe's metrics and similar techniques.

5.4.7 Update Design Constraint Analysis

The criteria for design constraint analysis applied to the detailed design in section 5.3.4 can be updated using the final code. At the code phase, real testing can be performed to characterize the actual software behavior and performance in addition to analysis.

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

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

5.4.8 Code Inspection Checklists (including coding standards)

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

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

5.4.9 Formal Methods

Generation of code is the ultimate output of Formal Methods. In a "pure" Formal Methods system, analysis of code is not required. In practice, however, attempts are often made to "apply" Formal Methods to existing code after the fact. In this case the analysis techniques of the previous sections (5.4.1 through 5.4.5) may be used to "extract" the logic of the code, and then compare the logic to the formal requirements expressions from the Formal Methods.

5.4.10 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 technique for identifying unused code; however, unused code is often identified during the course of performing other types of code analysis. Unused code can be found during unit testing with COTS coverage analyzer tools.

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

5.5 Test Analysis

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

1) analyses before the fact to ensure validity of tests
2) analyses of the test results

During the test planning discussed in Section 4.6, tests are devised to verify all safety requirements where testing has been selected as appropriate verification method. This is not considered here as analysis. Analysis before the fact should, as a minimum, consider test coverage for safety critical Must-Work-Functions.

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].

Techniques known as “white box” testing can be performed, usually at the modular level.

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

5.5.2 Test Results Analysis

Test results are analyzed to verify that all safety requirements have been satisfied. The analysis also verifies that all identified hazards have been eliminated or controlled to an acceptable level of risk. The results of the test safety analysis are provided to the ongoing system safety analysis activity.

All test discrepancies of safety critical software should be evaluated and corrected in an appropriate manner.

5.5.3 Independent Verification and Validation

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

5.6 Operations & Maintenance

Maintenance of software is describe above in section 4.8.

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 offline analog (or simulator) of the operational software system, prior to placing it into service.

6. REFERENCES

References are listed by section, numbered from [1] for each section.

1. INTRODUCTION
1.1 Scope

[1] NSS 1740.13 NASA Software Safety Standard, Interim Release, June 1994

[2] NSTS 13830B Implementation Procedure for NASA Payload System Safety Requirements

[3] SMAP-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. SYTEM SAFETY PROGRAM

2.1 Preliminary Harzard Analysis (PHA)

[1] NHB 1700.1 (V1-B) NASA Safety Policy and Requirements Document, Chapter-3, System Safety, and Appendix-H (Analysis Techniques)

[2] NSTS 13830B Implementation Procedure for NASA Payload System Safety Requirements

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, 1995. Appendix-A Medical Devices - The Therac-25 Story.

3.2 Tailoring the Effort - Value vs Cost

[3] NMI 8010.1, "Classification of NASA Space Transportation (STS) Payloads".

[4] MIL-STD-882C Military Standard - System Safety Program Requirements insertion for 3.4 Software Safety Assurance Techniques for Software Development Phases

3.4 Software Safety Assurance Techniques for Software Development Phases

[5] International Electrotechnical Committee (IEC), draft standard IEC 1508, “Software for Computers in the Application of Industrial Safety-Related Systemns”.

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) SafetyDesign Requirements and Guidelines for Munition 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

5 SOFTWARE SAFETY ANALYSIS

5.1 Software Safety Requirements 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 is being replaced by DOD-STD-498)

[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

5.1.3 Specification Analysis

[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.

5.3 Detailed Design Analysis

[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

5.3.12 FORMAL METHODS

[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.

5.3.13 REQUIREMENTS STATE MACINES

[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.

5.5 TEST ANALYSIS

[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.

6 REFERENCES

[1] DO-178B Software Considerations in Airborne Systems and Equipment Certification (Federal Aviation Administration).

[2] Malan, Ruth; Coleman, Derek; Letsinger, Reed. “Lessons Learned from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard.” OOPSLA ‘95, Austin, TX. ACM 0-89791-703-0/95/0010.