NATO STANDARD

AOP-52

GUIDANCE ON SOFTWARE SAFETY DESIGN AND ASSESSMENT OF MUNITION-RELATED COMPUTING SYSTEMS

Edition B Version 1

NOVEMBER 2016

NORTH ATLANTIC TREATY ORGANIZATION

ALLIED ORDNANCE PUBLICATION

Published by the
NATO STANDARDIZATION OFFICE (NSO)

INTENTIONALLY BLANK

NORTH ATLANTIC TREATY ORGANIZATION (NATO)

NATO STANDARDIZATION OFFICE (NSO)

NATO LETTER OF PROMULGATION

29 November 2016

1. The enclosed Allied Ordnance Publication AOP-52, Edition B, Version 1, GUIDANCE ON SOFTWARE SAFETY DESIGN AND ASSESSMENT OF MUNITION-RELATED COMPUTING SYSTEMS, which has been approved by the nations in the CNAD Ammunition Safety Group (AC/326), is promulgated herewith. The agreement of nations to use this publication is recorded in STANAG 4452.

3. AOP-52, Edition B, Version 1 is effective upon receipt and supersedes AOP-52, Edition 1 which shall be destroyed in accordance with the local procedure for destruction of documents.

5. This publication shall be handled in accordance with C-M(2002)60.

Edvardas MAŽEIKIS
Major General, LTUAF
Director, NATO Standardization Office

INTENTIONALLY BLANK

RESERVED FOR NATIONAL LETTER OF PROMULGATION

INTENTIONALLY BLANK

RECORD OF RESERVATIONS

CHAPTER	RECORD OF RESERVATION BY NATIONS
Note: The reservations listed on this page include only those that were recorded at time of promulgation and may not be complete. Refer to the NATO Standardization Document Database for the complete list of existing reservations.	

INTENTIONALLY BLANK

RECORD OF SPECIFIC RESERVATIONS

[nation]	[detail of reservation]
Note: The reservations listed on this page include only those that were recorded at time of promulgation and may not be complete. Refer to the NATO Standardization Document Database for the complete list of existing reservations.	

INTENTIONALLY BLANK

Table of Contents

Figures

Tables

1 Executive Overview

Software and programmable logic devices[1] play an increasingly important role in the operation and control of hazardous, safety-related[2] functions. Software first played a role in performing complex ballistic computations for ordnance, a safety-related function. However, applications still required people in the process, providing assurance of the validity, and hence, the safety of the results. In decades since the engineering community has relinquished human control of hazardous operations. This change is primarily due to the ability of software to perform critical control tasks reliably at speeds unmatched by its human counterparts. Other factors influencing this transition are our ever-growing need and desire for increased versatility, greater performance, higher efficiency, and decreased life cycle cost. In most instances, software can meet all of the above attributes when properly designed, developed, tested, and integrated. Increasingly the human operator is being taken out of the control loop as software logic can be designed to make accurate decisions without emotion or doubt.

There is a critical need to perform system safety engineering on safety-related systems to not only reduce the potential safety risk in all aspects of a program, but also to accurately assess the safety risks attributed to software. Software System Safety (SSS) activities involve the specification, design, code, test, Verification and Validation (V&V), operation and maintenance, and change control functions of the software engineering development process. The definition of system safety engineering, which includes SSS, is:

"The application of engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle."

The ultimate responsibility for the development of a "safe system" rests with program management and the system development team. The commitment to provide qualified people and an adequate budget and schedule for a software development program begins with the program director or Program Manager (PM). Top management must be a strong voice of safety advocacy and must communicate this personal commitment to each level of program and technical management. The PM must support the integrated safety process between systems engineering, software engineering, and safety engineering in the design, development, test, and operation of the system software. The purpose of this document is as follows:

Provide management and engineering guidelines to achieve an acceptable level of assurance that software will execute within the system context and operational environment with an acceptable level of safety risk.

[1] Software-like devices include programmable logic devices and similar hardware
[2] The term "safety-related" encompasses both safety critical and safety significant

2 Introduction to the AOP

2.1 Introduction

The system development team should read this section of this Allied Ordnance Publication (AOP). This section discusses the following major subjects:

- The major purpose for writing this AOP
- The scope of the subject matter that this AOP will present
- How this AOP is organized and the best procedure for you to use, to gain its full benefit

The individual responsible for System Safety (generally a System Safety Engineer) is a key billet and is crucial to the design, development, testing and redesign of modern systems. Whether a hardware engineer, software engineer, "safety specialist," or safety manager, it is his/her responsibility to ensure that an acceptable level of safety risk is achieved and maintained throughout the life cycle of the system(s) being developed. This AOP provides a rigorous and pragmatic process of SSS planning and analysis to be used by the System Safety Engineer.

SSS cannot function independent of the total effort nor can it be ignored. Systems, both "simple" and highly integrated multiple subsystems and systems of systems, are experiencing an extraordinary growth in the use of computers, software, and software-like devices to monitor and/or control safety-related subsystems and functions. A software specification error, design flaw, or the lack of initial safety-design requirements can contribute to or cause a system failure or erroneous human decision. Preventable death, injury, loss of the system, equipment damage, or environmental damage can result. To achieve an acceptable level of safety for software and software-like devices used in critical applications, software safety engineering must have primary emphasis early in the requirements definition and system conceptual design process. Safety-related software must then receive a continuous emphasis from management as well as a continuing engineering analysis throughout the development and operational life cycles of the system.

The process described in this AOP, which is an application of System Safety Engineering to software design, development, testing and maintenance, is also applicable to the safety assessment of software-like devices including programmable logic devices used in weapons and related systems. Like software, the application of these devices in safety-related roles is increasing. There are finite differences in the process when applied to these devices that will be noted in the subsequent sections of the AOP.

To summarize, this AOP is a "how-to" guide for use in the understanding of SSS and the contribution of each functional discipline to the overall goal. It is applicable to all types of weapons and related systems in all types of operational uses.

2.2 Purpose

The purpose of the AOP is to provide management and engineering guidelines to achieve a reasonable level of assurance that the software and software-like devices will execute within the system context and operational environment with an acceptable level of safety risk.

2.3 Scope

This AOP is both a reference document and management tool for aiding managers and engineers at all levels, in any government or industrial organization. It documents "how to" in the development and implementation of an effective SSS process. Effective implementation should minimize system hazards caused by software in safety-related applications.

The primary responsibility for management of the SSS process lies with the system safety manager/ engineer in both the supplier and acquirer's organizations. However, nearly every functional discipline has a vital role and must be intimately involved in the SSS process. The SSS tasks, techniques, and processes outlined in this AOP are basic enough to apply to any system that uses software or software-like devices in critical areas. It serves the need for all contributing disciplines to understand and apply qualitative and quantitative analysis techniques to ensure the safety of hardware systems controlled by software.

This AOP is a guideline and is not intended to supersede any National Government or Agency policy, standard, or guidance pertaining to system safety (e.g., US MIL-STD-882 series, UK Def-Stan 00-56) or software engineering and development standards. It is written to clarify the SSS requirements and tasks specified in governmental and commercial standards and guideline documents. This AOP is not a compliance document but a reference document. It provides the program management, especially the system safety manager and the software development manager with sufficient information to perform the following:

- Properly scope the SSS effort
- Identify the data needed to effectively monitor the developer's compliance with system safety requirements
- Evaluate the residual risk associated with the software or software-like devices in the overall system context

The AOP is not a tutorial on software engineering. However, it does address some technical aspects of software function and design to assist with understanding software safety. It is an objective of this AOP to provide each member of the SSS team with a basic understanding of sound systems and software safety practices, processes, and techniques. Another objective is to demonstrate the importance of the interaction between technical and managerial disciplines in defining software safety requirements (SSR) for the safety-related software components of the system. A final objective is to show where the team can design safety features into the software to eliminate or control identified hazards.

2.4 AOP Overview

2.4.1 Introduction to the "Systems" Approach

System safety engineering demonstrated the benefits of a "systems" approach to safety risk analysis and mitigation in numerous systems. When conducting a hazard analysis on a hardware subsystem as a separate entity, System Safety identifies a set of unique hazards applicable only to that subsystem. However, when they analyze that same subsystem in the context of its interfaces with the rest of the "system components," the analysis produces numerous other hazards not discovered by the original analysis. Conversely, the results of a system-level analysis may demonstrate that hazards identified in the subsystem analysis are either reduced or

eliminated by other components of the system. Regardless, the identification of critical subsystem interfaces (such as software) with their associated hazards is a vital aspect of safety risk minimization for the total system.

The systems approach should be applied when analyzing software that performs, and/or controls, safety-related functions within a system. The success of a software safety program is predicated on it. Software is a key component of the safety risk potential of systems being developed and fielded. Not only are the internal interfaces of the system important to safety, but also are the external interfaces.

Figure 2-1 depicts specific software internal interfaces within the "system" block (within the ovals) and external software interfaces to the system. Each identified software interface may possess safety risk potential to the operators, maintainers, environment, or the system itself. The acquisition and development process must consider these interfaces during the design of both the hardware and software systems. To accomplish this, the hardware and software development life cycles must be fully understood and integrated by the design team.

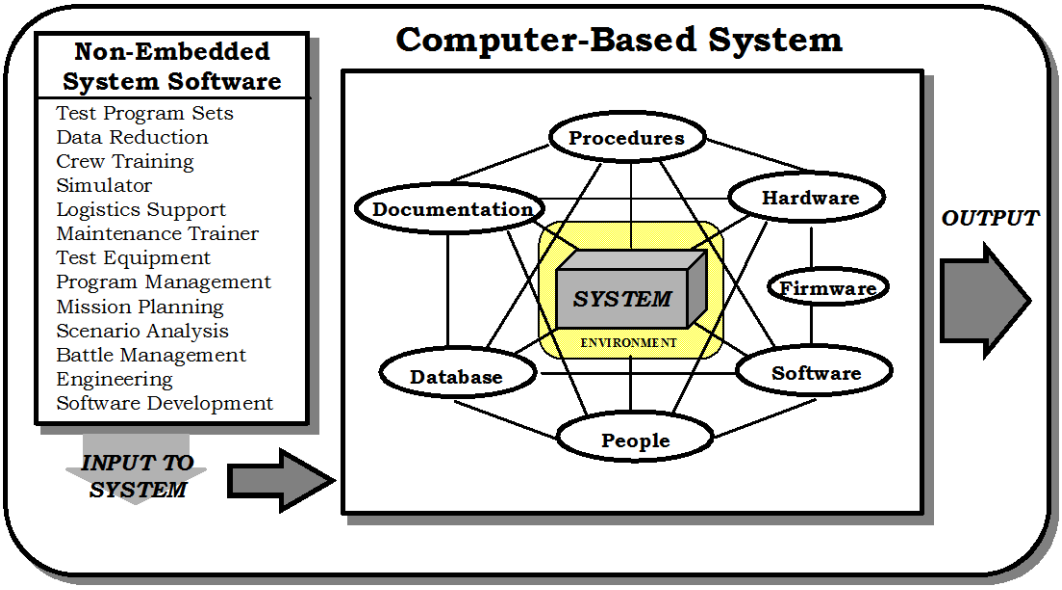


Figure 2-1: Example of Internal System Interfaces

2.4.1.1 The Hardware Development Life Cycle

The typical hardware development life cycle (shown in Figure 2-2) varies from country to country and occasionally from program to program. A proven acquisition model that produces the desired engineering results in the design, development, manufacturing, fabrication, and test activities is essential to ensuring the operational capabilities and maintainability of the system throughout its life cycle. In general, different elements of the system will be in different phases of development until the project is complete. Between each phase, an assessment of the system design and program status should occur before proceeding into subsequent phases of the development or deployment life cycle.

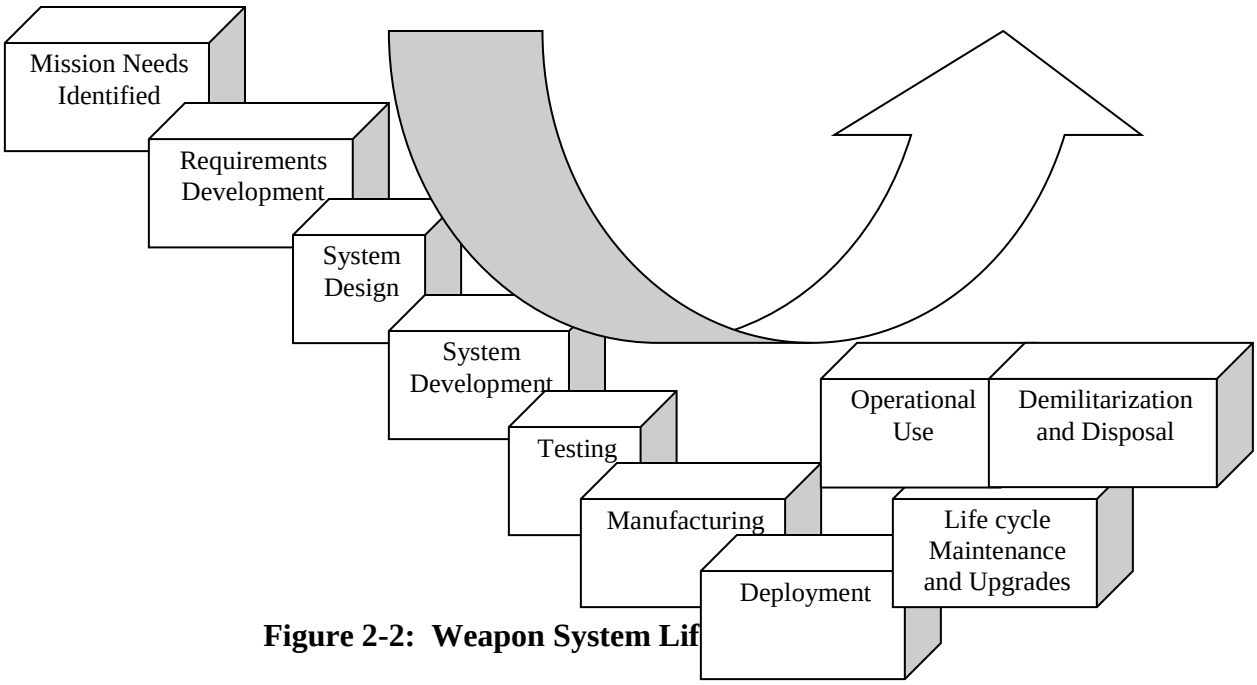


Figure 2-2: Weapon System Lif

2.4.1.2 The Software Development Life Cycle

The system safety team must be fully aware of the software life cycle used by the development activity. Software Engineering teams have developed numerous life cycle models that system supplier's use in some capacity on a variety of development programs. This AOP will not enter into a discussion as to the merits and limitations of different life cycle process models. From a software engineering perspective, the software engineering team must choose the life cycle model best suited for the individual development. The important issue is for the system safety team to recognize and understand which model the Software Engineering Team is using, and how they should correlate and integrate safety activities with the chosen model to achieve the desired outcomes and safety goals. Appendix C of this AOP presents several different models to introduce examples to the reader and discuss the integration of the SSS process into those models.

2.4.1.3 The Integration of Hardware and Software Life Cycles

The life cycle process of system development was instituted so managers would not be forced to make snap decisions. A structured life cycle, complete with controls, audits, reviews, and key decision points, provides a basis for sound decision making based on knowledge, experience, and training. It is a logical flow of events representing an orderly progression from a "user need" to finalize activation, deployment, and support.

The "systems approach" to software safety engineering supports a structured, well-disciplined, and adequately documented system acquisition life cycle model that incorporates both the system development model and the software development model. Program plans must describe in detail how each engineering discipline will interface and perform within the development life cycle. Providing graphical representations of the life cycle model of choice for a given development activity during the planning processes will aid in the planning and implementation processes of software safety engineering. It allows for the integration of safety-related requirements and

guidelines into the design and code phases of software development. It also assists in the timely identification of safety-specific test and verification requirements to prove the intended implement of the original design requirements. Further, it allows the incorporation of safety inputs to the prototyping activities in order to demonstrate safety concepts.

3 Software Safety Engineering

3.1 Introduction

This section of the AOP introduces the managerial process and the technical methods and techniques inherent in the performance of software safety tasks within the context of a systems safety engineering and software development program. It will include detailed tasks and techniques for the performance of safety analysis, and for the traceability of Software Safety Requirements (SSRs) from design to test. It will also provide the current "best practices" which may apply as one considers the necessary steps in establishing a credible and cost-effective SSS program (Figure 3-1).

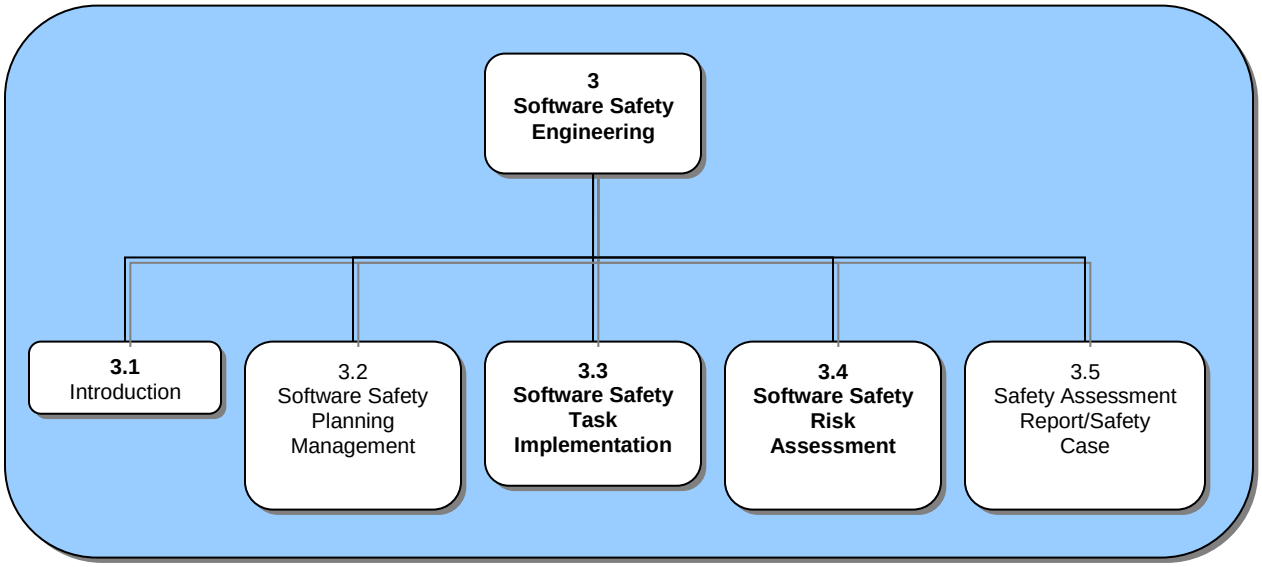


Figure 3-1: Chapter Contents

This chapter applies to all managerial and technical disciplines. It describes the processes, tools, and techniques used to reduce the safety risk of software operating in safety-related systems.

Its primary purposes are as follows:

- To establish the relationship between software safety and the overall system safety
- Define a software safety engineering best practice
- Describe essential tasks required of each professional discipline assigned to the SSS Team
- Identify interface relationships between professional disciplines and the individual tasks assigned to the SSS Team
- Identify "best practices" to complete the software safety process and describe each of its subprocesses

- Recommend "tailoring" actions to the software safety process to accommodate specific requirements such as different software engineering methodologies

This document assumes a novice's understanding of software safety engineering within the context of system safety and software engineering. Note that many topics of discussion within this section are constructs within basic system safety engineering. This is because it is impossible to discuss software safety outside of the domain of system safety engineering and management, systems engineering, software development, and program management.

3.1.1 Section Format

This section presents both graphical and textual descriptions of the managerial and technical tasks required for a successful software safety-engineering program. The format of each managerial process task and technical task, method, or technique will provide the following:

- Graphical representation of the process step or technical method
- Introductory and supporting text
- Prerequisite (input) requirements for task initiation
- Activities required to perform the task (including interdisciplinary interfaces)
- Associated subordinate tasks
- Critical task interfaces
- A description of required task output(s) and/or product(s)

This particular format helps to explain the inputs, activities, and outputs for the successful accomplishment of activities to meet the goals and objectives of the software safety program. For those that desire additional information, Appendices provide supplemental information to the main sections of this document.

3.1.2 Process Charts

The process charts in Appendix F graphically depict the process from a high, system-level safety assessment process to details regarding individual steps and tasks in the process. The intent of these charts is to complement the content of AOP-15: the system-level process both provides the basis for the Software Systems Safety Process and shows the inherent cohesiveness of the Software Systems Safety Process with the System Safety Process. Therefore, the system-level charts are essential to the overall process description. Following the process charts, the reader can gain an understanding of the process and the various steps involved in each task within the process.

Each system and software safety-engineering task has a supporting, intermediate level process chart. Each intermediate level process chart provides the software safety analyst with a list of the inputs required to perform the overall process, the sub-processes involved in completing the process, the expected outputs from the process, the personnel involved, and the entry and exit criteria for the task

3.1.3 Software Safety Engineering Products

The specific products produced by the software safety engineering tasks are difficult to segregate from those developed within the context of the System Safety Program (SSP). It is likely, within individual programs, that supplemental software safety documents and products will be produced to support the system safety effort. These may include supplemental analysis, Data Flow Diagrams (DFD), functional flow analysis, and software requirements specifications (SRS) and the development of Software Analysis Folders (SAF). This AOP will identify and describe the documents and products that the software safety tasks will either influence or generate. Specific documents include the following:

- System Safety Program Plan (SSPP)
- Software Safety Program Plan Appendix to SSPP (SwSPP)
- Generic Software Safety Requirements List (GSSRL)
- Safety-Related Functions List (SRFL)
- Preliminary Hazard List (PHL)
- Preliminary Hazard Analysis (PHA)
- Subsystem Hazard Analysis (SSHA)
- Safety Requirements Criteria Analysis (SRCA)
- System Hazard Analysis (SHA)
- Operating and Support Hazard Analysis (O&SHA)
- Safety Assessment Report (SAR)

3.2 Software Safety Planning Management

Inadequately specified safety requirements in the system-level specification documents generally lead to program schedule and cost impacts later when safety issues arise and the necessary system safety engineering is not complete. Program planning precedes all other phases of the SSS program and is perhaps the single most important step in the overall safety process. The software safety program must be integrated with and parallel to both the System Safety Program and the software development program. The software safety analyses must provide the necessary input to software development. These inputs to engineering processes include:

- Safety design requirements
- Safety design changes
- Safety tests

The software aspects of system safety tend to be more problematic since the risk associated with the software is often ignored or poorly understood until late in the system design. Establishing the safety program and/or performing the necessary safety analyses later in the program results in delays, cost increases, and a less effective safety program.

The history of software-related safety issues, as derived from lessons learned, re-enforces the need for a practical, logical, and disciplined approach to reducing the safety risk of software

performing safety-related functions within a system. Establishing this managerial and engineering discipline "up front" is essential as is including it in the planning activities that both describe and document the breadth and depth of the program. Detailed planning ensures the identification of critical program interfaces and support and establishes formal lines of communication between disciplines and among engineering functions. The potential for program success increases through sound planning activities that identify and formalize the managerial and technical interfaces of the program.

This section is applicable to all members of the SSS Team. It assumes a minimal understanding and experience with safety engineering programs. The topics include the following:

- Identification of managerial and technical program interfaces required by the SSS Team
- Definition of user and supplier contractual relationships to ensure that the SSS Team implements the tasks, and produces the products, required to meet contractual requirements
- Identification of programmatic and technical meetings and reviews normally supported by the SSS Team
- Identification and allocation of critical resources to establish a SSS Team and conduct a software safety program
- Definition of planning requirements for the execution of an effective program

The planning for an effective SSP and software safety program requires extensive forethought from both the supplier and the acquirer. Although they both envision a perfect SSP, there are subtle differences associated with the identification, preparation, and execution of a successful safety program from these two perspectives, which must be understood by both. The contents of Figure 3-2 and

Figure 3-3 represent the primary differences between agencies that both must understand before considering the software safety planning and coordinating activities. These differences will be discussed in section 3.2.1

3.2.1 Planning

Comprehensive planning for the software safety program requires an initial assessment of the degree of software involvement in the system design and the associated hazards. The development and safety teams know little about the system other than operational requirements during the early planning stages. Therefore, the contractual safety requirements must encompass all possible designs. This generally results in a generic set of requirements that will require later tailoring of a SSS program to the system design and implementation. Having third party certification (e.g. CMM, CMMI, DO-178B, ISO 9001) alone does not guarantee that safety concerns will be properly addressed.

Figure 3-2 represents the basic inputs, outputs, tasks, and critical interfaces associated with the planning requirements associated with the procuring office or agency (acquirer). Planning and budgeting for all these billets and functions result in proper execution and success of the safety program. Failure to do so should be captured as program risk and safety risk and reported at program reviews.

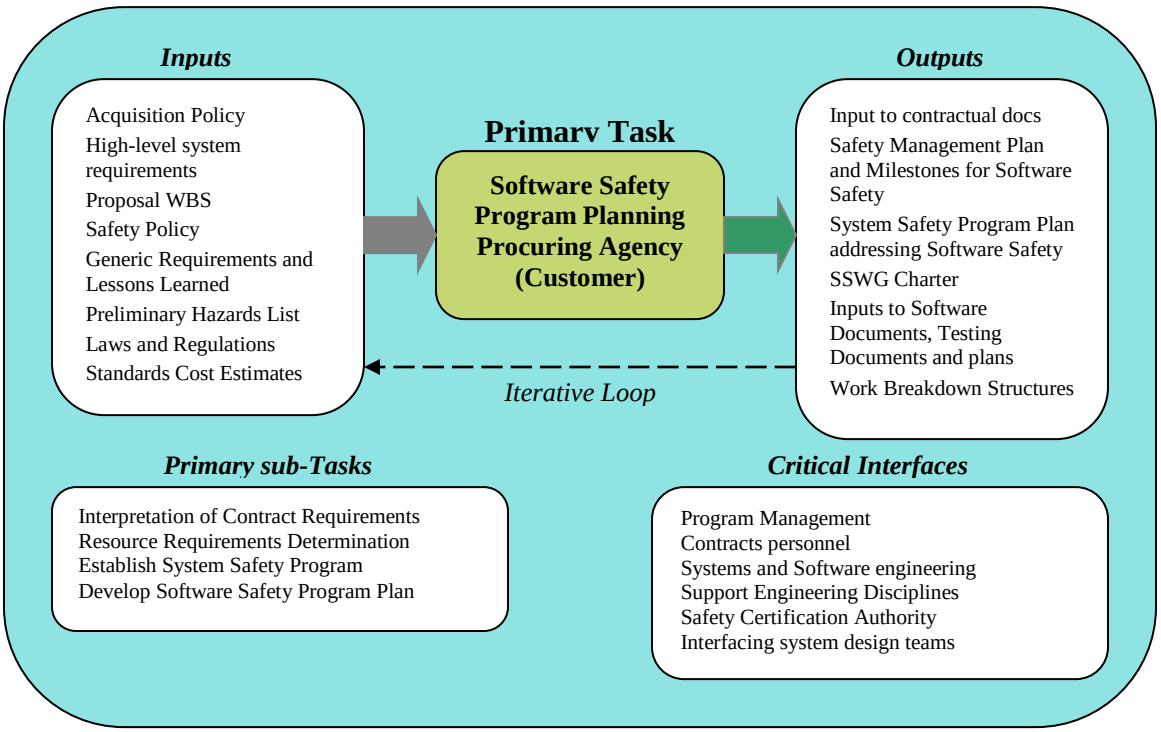


Figure 3-2: Software Safety Planning Viewpoint of the Procuring Authority

For the acquirer, software safety program planning begins as soon as they identify the need for the system. The acquirer must identify points of contact within the organization and define the interfaces between various engineering disciplines, administrative support organizations, program management, contracting group, and Integrated Product Teams (IPT) within the acquirer to develop the necessary requirements and specifications documents. The acquirer must incorporate the necessary language into any contractual documents to ensure that the system under development will meet the safety goals and objectives.

Acquirer safety program planning continues through contract award and may require periodic updating during initial system development and as the development proceeds through various phases. However, management of the overall SSP continues through system delivery and acceptance and throughout the system's life cycle. After deployment, the acquirer must continue to track system hazards and risks and monitor the system in the field for safety concerns identified by the user. The acquirer must also make provisions for safety program planning and management for any upgrades, product improvements, maintenance, technology refreshment, and other follow-on efforts to the system.

The major milestones affecting the acquirer's safety and software safety program planning include release of contract requests for proposals or quotes, proposal evaluation, major program milestones, system acceptance testing and evaluation, production contract award, initial operational capability (release to the users), and system upgrades or product improvements.

Although the supplier's software safety program planning begins after receipt of a contract request for proposals or quotes, the supplier can significantly enhance his/her ability to establish an effective program through prior planning (see

Figure 3-3). Prior planning includes establishing effective systems engineering and software engineering processes that fully integrate system and software systems safety. Corporate engineering standards and practices documents that incorporate the tenets of system safety provide a strong baseline from which to build a successful SSP even though the contract may not contain specific language regarding the safety effort.

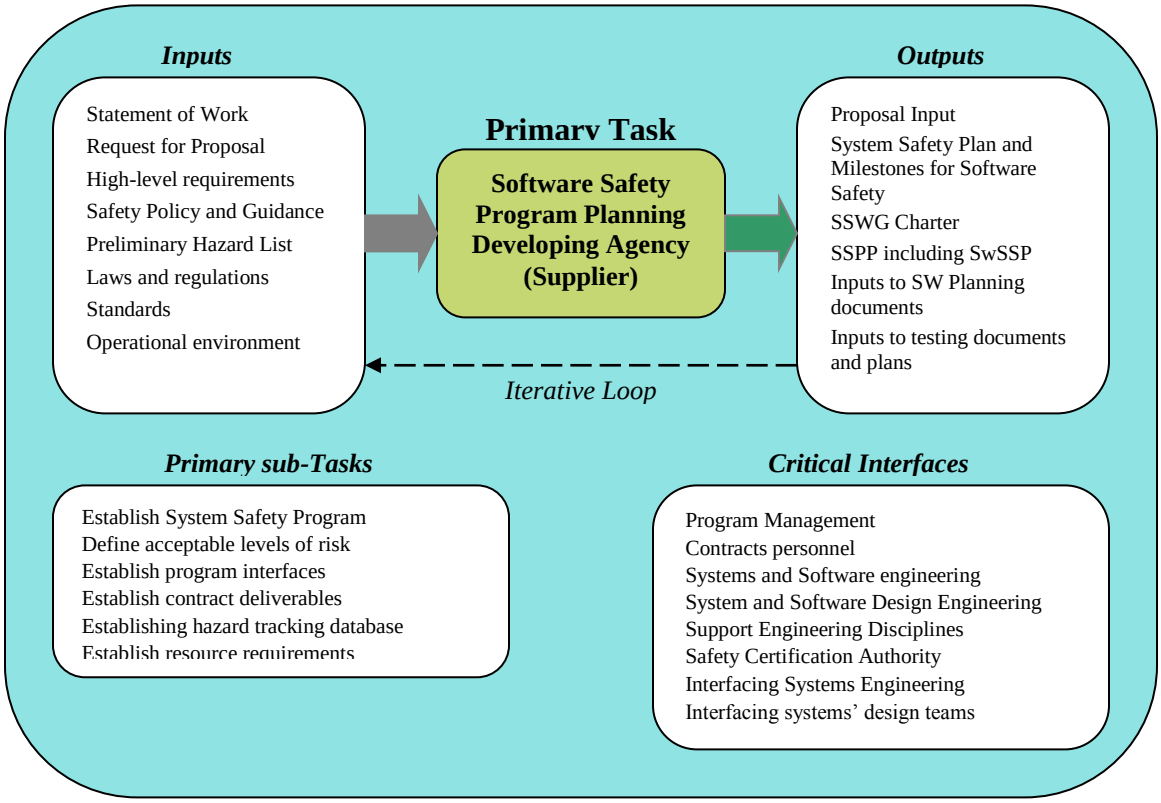


Figure 3-3: Software Safety Planning Viewpoint of the Developing Agency

Many acquisition reform initiatives recommend that the respective governments take a more interactive and/or proactive approach to system development without interfering with that development. The interactive aspect is to participate as a member of the supplier's Integrated Product Teams (IPTs) as an advisor without hindering development. This requires a careful balance on the part of the government participants. From the system safety and SSS perspective, that includes active participation in the appropriate IPTs by providing the government perspective on recommendations and decisions made in those forums. This also requires the government representative to alert the supplier to hazards known to the government but not to the supplier.

Often, contract language is non-specific and does not provide detailed requirements, especially safety requirements for the system. Therefore, the supplier must define a comprehensive SSP

that ensures the delivered system provides a level of safety risk to the acquirer that is as low as reasonably practicable (ALARP), not only for the acquirer's benefit, but for the supplier's benefit as well. At the same time, the supplier must remain competitive and reduce safety program costs to the lowest practicable level consistent with ensuring the delivery of a system with the lowest risk practicable. This (residual) risk has to be a documented agreement between acquirer and supplier. Although the preceding discussion focused on the interaction between the Government acquirer and the supplier, the same tenets apply to any contractual relationship, especially between prime and subcontractors.

The supplier's software safety planning continues after contract award and requires periodic updates as the system proceeds through various phases of development. These updates should be in concert with the acquirer's software safety plans. However, management of the overall system and SSS programs continues from contract award through system delivery and acceptance and may extend throughout the system life cycle, depending on the type of contract. If the contract requires that the supplier perform routine maintenance, technology refreshments, or system upgrades, the software safety program management and engineering must continue throughout the system's life cycle. Thus, the supplier must make provisions for safety program planning and management for these phases and other follow-on efforts on the system.

The major milestones affecting the supplier's safety and software safety program planning include the receipt of contract requests for proposals or quotes, contract award, major program milestones, system acceptance testing and evaluation, production contract award, release to the acquirer, system upgrades, and product improvements.

While the software safety planning objectives of the acquirer and supplier may be similar, the planning and coordination required to meet these objectives may come from different perspectives (in terms of specific tasks and their implementation), but they must be in concert. Regardless, both agencies must work together to meet the safety objectives of the program. In terms of planning, this includes the following:

- Establishment of a SSP
- Definition of acceptable levels of safety risk
- Definition of critical program, management, and engineering interfaces
- Definition of contract deliverables
- Development of a Software Hazard Criticality Matrix (SHCM) (see Section 3.2.1.5)

3.2.1.1 Establish the System Safety Program

The acquirer must establish the safety program as early as practical in the development of the system. The project manager should identify a safety manager early in the program to serve as the single point of contact for all safety-related matters on the system. The project manager should budget for, and the safety manager should establish and chair, a Software Systems Safety Working Group (SwSSWG) or SSS Team. The safety manager will interface with safety review authorities, the supplier safety team, acquirer and supplier program management, the safety engineering team, and other groups as required ensuring that the safety program is effective and efficient. For large system developments where software is likely to be a major portion of the development, a safety engineer for software may also be identified who reports directly to the

overall system safety manager. The size of the safety organization will depend on the complexity of the system under development, and the inherent safety risks. Another factor influencing the size of the project manager's safety team is the degree of interaction with the acquirer and supplier and the other engineering and program disciplines. If the development approach is a team effort with a high degree of interaction between the organizations, the safety organization may require additional personnel to provide adequate support.

The acquirer should prepare a System Safety Management Plan (SSMP) describing the overall safety effort within the acquirer's organization and the interface between the acquirer safety organization and the supplier's system safety organization. The SSMP is similar to the SSPP in that it describes the roles and responsibilities of the program office individuals with respect to the overall safety effort. The safety manager or project manager should coordinate the SSMP with the supplier's SSPP to ensure that the tasks and responsibilities are complete and will provide the desired risk assessment. The SSMP differs from the SSPP in that it does not describe the details of the safety program, such as analysis tasks, contained in the SSPP.

The acquirer must specify the software safety program for programs where software and software-like devices (e.g. PLDs) performs or influences safety-related functions of the system. The acquirer must establish IPTs in accordance with contractual requirements, managerial and technical interfaces and agreements, and the results of all planning activities discussed in previous sections of this AOP. Proper and detailed planning will increase the probability of program success. The tasks and activities associated with the establishment of the SSP are applicable to both the supplier and the acquirer. Unfortunately, the degree of influence of the software on safety-related functions in the system may be unknown until the design progresses to the point of functional allocation of requirements at the system level.

The project manager must predicate the software safety program on the goals and objectives of the system safety and the software development disciplines of the proposed program. The safety program must focus on the identification and tracking (from design, code, and test) of both initial SSRs and guidelines, and those requirements derived from system-specific, functional hazards analyses. A common deficiency in software safety programs is the lack of a team approach in addressing both the initial and the functional SSRs of a system. The software development community has a tendency to focus on only the initial SSRs while the system safety community may focus primarily on the functional SSRs derived through hazard analyses. A sound SSS program traces both sets of requirements through test and requirements verification activities. The ability to identify (in total) all applicable SSRs is essential for *any* given program.

3.2.1.2 Defining Acceptable Levels of Risk

One of the key elements in safety program planning is the identification of the acceptable level of risk for the system. AOP-15 provides the guidelines for defining levels of risk. The acquirer must also provide the suppliers with risk acceptance authorities and reporting requirements. The supplier must provide the acquirer supporting documentation for the risk acceptance.

3.2.1.3 Program Interfaces

The system safety engineer is responsible for the coordination, initiation, and implementation of the Software Systems Safety engineering program. While they cannot delegate this responsibility to any other engineering discipline within the development team, software safety

can (and must) assign specific tasks to the domain engineers with the appropriate expertise. Historically, system safety engineering performs the engineering necessary to identify, assess, and eliminate or reduce the safety risk of hazards associated with complex systems. Now, as software becomes a major aspect of the system, software safety engineering must establish and perform the required tasks and establish the technical interfaces required to fulfill the goals and objectives of the system safety (and software safety) program. However, the SSS Team cannot accomplish this independently without the inter-communication and support from other managerial and technical functions. Many product development agencies use the IPTs structure to ensure the success of the design, manufacture, fabrication, test, and deployment of weapon systems. These IPTs formally establish the accountability and responsibility between functions and among team members. This accountability and responsibility is both from the top down (management-to-engineer) and from the bottom up (engineer-to-management).

The establishment of a credible SSS activity within an organization requires this same rigor in the identification of team members, the definition of program interfaces, and the establishment of lines of communication. Establishing formal and defined interfaces allows program and engineering managers to assign required expertise for the performance of the identified tasks of the software safety engineering process. Figure 3-4 shows the common interfaces necessary to support an SwSSP. It includes management interfaces, technical interfaces, and contractual interfaces.

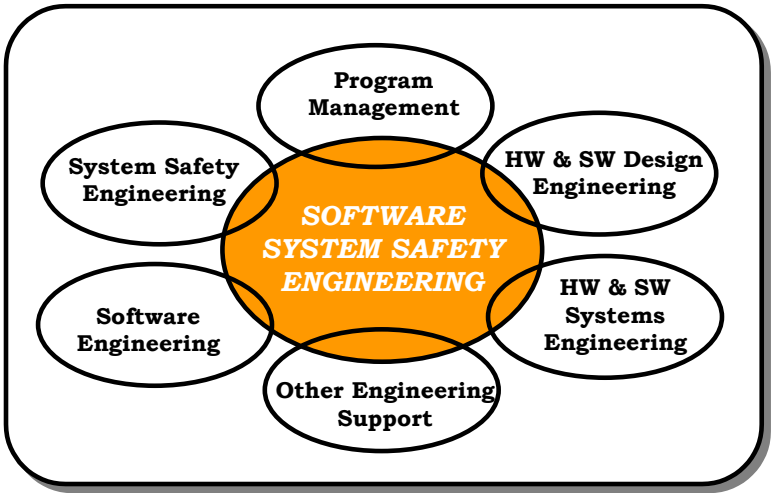


Figure 3-4: Software Safety Program Interfaces

3.2.1.3.1 Management Interfaces

The Project Manager:

- Coordinates the activities of each professional discipline for the entire program,
- Allocates program resources,
- Approves the programs' planning documents, including the SSPP, and
- Reviews safety analyses; accepts impact on system for Critical and higher category hazards (based upon acceptable levels of risk); and submits finding to senior acquisition personnel for acceptance of unmitigated, unacceptable hazards.

It is the project manager's responsibility to ensure that processes are in place within a program that meet, not only the programmatic, technical, and safety objectives, but also the functional and system specifications and requirements of the acquirer. The project manager must allocate critical resources within the program to reduce the sociopolitical, managerial, financial, technical, and safety risk of the product. Therefore, management support is essential to the success of the SSS program.

The project manager ensures that the safety team develops a practical process and implements the necessary tasks required to:

- Identify system hazards
- Categorize hazards in terms of severity and likelihood
- Perform causal factor analysis
- Derive hardware and software design requirements to eliminate and/or control the hazards
- Provide evidence for the implementation of hardware and software safety design requirements
- Analyze and assess the residual safety risk of any hazards that remain in the design at the time of system deployment and operation
- Report the residual safety risk and hazards associated with the fielded system to the appropriate acceptance authority

The safety manager and the software engineering manager depend on program management for the allocation of necessary resources (time, tools, training, money, and personnel) for the successful completion of the required tasks.

3.2.1.3.2 Technical Interfaces

The engineering disciplines associated with system development must also provide technical support to the SSS Team (Figure 3-5). Engineering management, design engineers, systems engineers, software development engineers, Integrated Logistics Support (ILS), and other domain engineers supply this essential engineering support. Other domain engineers include reliability, human factors, quality assurance (QA), test and evaluation, verification and validation, maintainability, survivability, and supportability. Each member of the engineering team must provide timely support to the defined processes of the SSS Team to accomplish the safety analyses and for specific design influence activities that eliminate, reduce, or control hazard risk. This includes the traceability of SSRs from design-to-test (and test results) with its associated and documented evidence of implementation.

A sure way for the software safety activity to fail is to fail to secure software engineering acceptance and support of the software safety process, functions, and implementation tasks. One must recognize that most formal education and training for software engineers and suppliers does not present, teach, or rationalize system safety. The system safety process relating to the derivation of functional SSR through hazard analyses is foreign to most software suppliers. In fact, the concept that software can be a causal factor to a hazard is foreign to many software engineers.

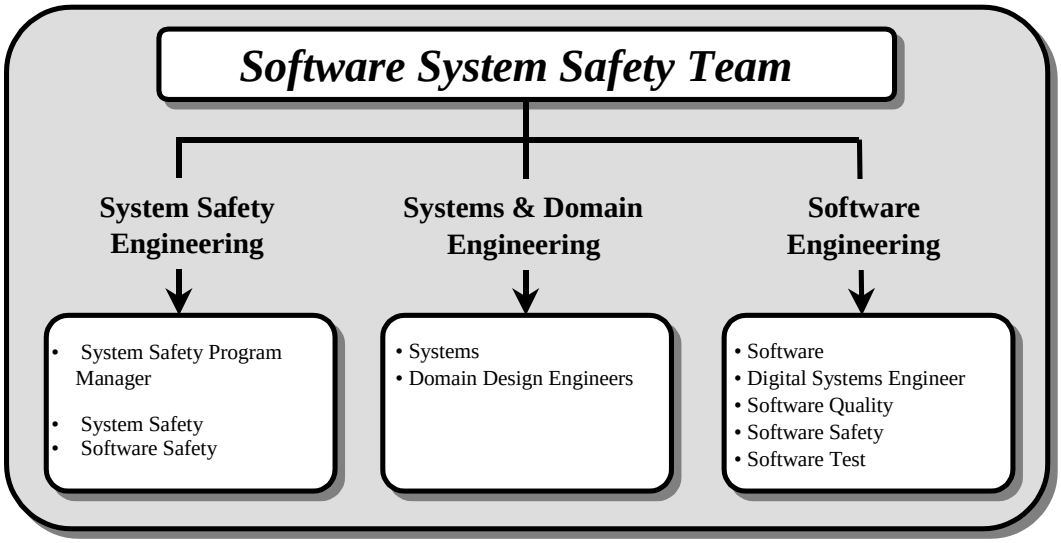


Figure 3-5: Proposed SSS Team Membership

Without the historical experience of cultivating technical interfaces between software development and system safety engineering, several issues may need resolution. They include:

- Software engineers may feel threatened that system safety has the responsibility for activities considered part of the software engineering realm
- Software suppliers are confident enough in their own methods of error detection, error correction, and error removal, that they ignore the system safety inputs to the design process. This is normally in support of initial SSRs
- There is insufficient communication and resource allocation between software development and system safety engineering to identify, analyze, categorize, prioritize, and implement both generic and derived SSRs

A successful SSS effort requires the establishment of a technical SSS Team approach. The SSP Manager, in concert with the systems engineer and software engineering team leaders must define the individual tasks and specific team expertise required and assigns responsibility and accountability for the accomplishment of these tasks. The SwSPP must include the identification and definition of the required expertise and tasks in the software safety portion or appendix. The team must identify both the generic SSRs and guidelines and the functional safety design requirements derived from system hazards and failure modes that have specific software input or influence. Once the team identifies these hazards and failure modes they can identify specific safety design requirements through an integrated effort. All SSRs must be traceable to test and be correct, complete, and testable where possible. The Requirements Traceability Matrix (RTM) within the Safety Requirements Criteria Analysis (SRCA) is a mechanism to this traceability. The implemented requirements must eliminate, control, or reduce the safety risk ALARP while meeting the user requirements within operational constraints.

3.2.1.3.3 Contractual Interfaces

Management planning for the SSS function includes the identification of contractual interfaces and obligations. Each program has the potential to present unique challenges to the system

safety and software development managers. These may include a request for procurement document that does not specifically address the safety of the system, to contract deliverables that are extremely costly to develop. Regardless of the challenges, the tasks needed to accomplish a SSS program must be planned to meet both the system and user specifications and requirements and the safety goals of the program.

3.2.1.4 Contract Deliverables

The contract tasking defines the deliverable documents and products desired by the acquirer. The SSPP should address deliverable documents and include the necessary activities and process steps required for its production. Completion of contract deliverable documentation may have links to the acquisition life cycle of the system and the program milestones identified in any systems engineering management plan. The planning required by the system safety manager ensures that the system safety and software safety processes provide the necessary data and output for the successful accomplishment of the plans and analysis. The system safety schedule should track closely to the systems engineering management plan and be proactive and responsive to both the acquirer and the design team. The safety master schedule and the SSPP should address contractually required documentation whether these documents are contractually deliverable or internal documents required supporting the development effort.

The procuring agency must also specify the content and format of each deliverable item. As existing government standards transition to commercial standards and guidance, the safety manager must ensure that the team does sufficient planning to specify the breadth, depth, and timeline of each deliverable document. The breadth and depth of the deliverable documents must provide the necessary audit trail to ensure that acceptable levels of risk are achieved (and are visible) during development, test, support transition, and maintenance in the out-years. The deliverables must also provide the necessary evidence or audit trail for validation and verification of SSRs. The primary method of maintaining a sufficient audit trail is the utilization of a supplier's safety data library (SDL). This library would be the repository for all safety documentation.

3.2.1.5 Developing a Software Safety Criticality Index Matrix

A Software Safety Criticality Index (SSCI) matrix is a table used to assign levels of criticality for software functions dependent on the severity of the hazard that a fault in the software function could cause and the autonomy of the software which could cause the hazard. The supplier's safety team may use this type of matrix or may propose an alternate matrix if one more appropriate to the system under development exists. The purpose of a SSCI matrix is to enable the safety engineer to assess and assign levels of rigor to the development process for software functions when the proposed autonomy of the software is known and the severity of any hazard linked to the software has been established. See section 3.4 for guidelines on levels of autonomy, SSCI's and levels of rigor.

3.2.2 Management

Effective management of the safety program is essential to the effective and efficient reduction of system risk. This section discusses the managerial aspects of the software safety tasks and provides guidance in establishing and managing an effective software safety program. Establishing a software safety program includes establishing a Software System Safety Working

Group (SwSSWG). This is normally a sub-group of the SSWG and chaired by the safety manager. The SwSSWG has overall responsibility for the following:

- Monitoring and control of the software safety program
- Coordinating with the national safety review authority to ensure the adequacy of the planned software safety process
- Identifying and resolving hazards with software causal factors
- Interfacing with the other IPTs
- Performing final safety assessment of the system design

In the early planning phases, the specific design configuration of the system and the degree of interaction of the software with the potential hazards in the system are largely unknown. However, knowledge from previous, similar systems and lessons learned can provide significant information and insight to aid safety planning. The higher the degree of software involvement, the greater the resources required to perform the assessment. To an extent, the software safety program manager can use the early analyses of the design, participation in the functional allocation, and high-level software design process to minimize the amount of safety-related software. If the architecture distributes safety-related functions throughout the system and its related software, then the software safety program must encompass a much larger portion of the software. However, if the safety-related functions are associated with as few software modules as practical, the level of effort may be significantly reduced.

Effective planning and integration of the software safety efforts into the other IPTs will significantly reduce the software safety-related tasks performed by the SSS Team. Incorporating the generic SSRs into the plans developed by the other IPTs allows them to assume responsibility for their assessment, performance, and/or evaluation. For example, if the SSS Team provides the quality assurance generic SSRs to the Software Quality Assurance (SQA) IPT, they will perform compliance assessments with requirements, not just for safety, but also for all aspects of the software engineering process. In addition, if the SQA IPT "buys-into" the software safety program and its processes, it significantly supplements the efforts of the software safety engineering team, reduces their workload, and avoids duplication of effort. The same is true of the other IPTs such as CM and Software Test and Evaluation. In identifying and allocating resources to the software safety program, the software safety program manager can perform advance planning, establish necessary interfaces with the other IPTs, and identify individuals to act as software safety representatives on those IPTs

Identifying the number of analyses and the level of detail required to adequately assess the software involves a number of processes. The process begins with the identification of the system-level hazards in the PHL. This provides an initial idea of the concerns that must be assessed in the overall safety program. From the system specification review process, the functional allocation of requirements results in a high-level distribution of safety-related functions and system-level safety requirements to the design architecture. The safety-related functions and requirements are thus known in general terms. Software functions that have a high safety-criticality (e.g., warhead arming and firing) will require a significant analysis effort that may include code-level analysis. Safety Team's early involvement in the design process can reduce the amount of software that requires analysis; however, the software safety manager must

still identify and allocate resources to perform these tasks. Those safety requirements that conflict with others (e.g., reliability) require trade-off studies to achieve a balance between desirable attributes.

The software control categories discussed in Section 3.2.1.5 provide a useful tool for identifying software that requires in depth analysis and testing. Obviously, the more critical the software, the higher the level of effort necessary to analyze, test, and assess the risk associated with the software. In the planning activities, the SwSSWG identifies the analyses necessary to assess the safety of specific modules of code. One of the most important aspects of software safety program management is monitoring the activities of the safety program throughout system development to ensure that tasks are on schedule and within cost, and to identify potential problem areas that could affect the safety or software development activities. The software safety manager must:

- Monitor the status and progress of the software and system development effort to ensure that the software safety program schedule and milestones reflect program schedule changes
- Monitor the progress of the various development teams and ensure that the safety interface to each is working effectively. When problems are detected, either through feedback from the software safety representative or other sources, the software safety manager must take the necessary action to mitigate the problem
- Monitor and receive updates regarding the status of analyses, open hazards and other safety activities on a regular basis. The SwSSWG should discuss significant hazards at each meeting and update the status as required
- Periodically provide updates and discuss the status and progress of the software safety program with the appropriate safety review authority; obtain recommendations for adjustments to the software safety program to ensure a smooth completion

The system safety manager must identify the appropriate review authorities and adjust the schedule during the development process to accommodate these reviews. These reviews generally involve significant effort outside of the other software safety tasks. The supplier must determine the level of effort required for each review and the support that will be required during the review, incorporate these into the SwSPP, and coordinate with the SwSSWG. Complex systems generally require multiple reviews to update and satisfy the requirements of the appropriate review authorities.

3.2.3 Configuration Control

Configuration control must be established as soon as practical in the system development process. Prior to their implementation, the Software Configuration Control Board must approve all software changes occurring after an initial baseline has been established. A member of the system safety engineering team must be a member of the Board and tasked with the evaluation of all software changes for their potential safety impact. A member of the hardware Configuration Control Board must be a member of the software Board and vice versa to keep members apprised of hardware changes and to ensure that software changes do not conflict with or introduce potential safety hazards due to hardware incompatibilities.

There have been many instances where the "wrong version" of a component has accidentally been introduced into a deployed system and as a result caused unexpected failures or at least presented a potential hazard.

- Does the Software Development Plan (SDP) describe a thorough CM process that includes version identification, access control, change audits, and the ability to restore previous revisions of the system?
- Does the CM process rely entirely on manual compliance, or is it supported and enforced by tools?
- Does the CM process include the ability to audit the version of specific components (e.g., through the introduction of version identifiers in the source code that are carried through into the executable object code)? If not, how is process enforcement audited (i.e., for a given executable image, how can the versions of the components be determined)?
- Is there evidence in the design and source code that the CM process is being adhered to (e.g., Are version identifiers present in the source code if this is part of the CM process described)?
- During formal testing, do any problems with inconsistent or unexpected versions happen?

A second issue that affects confidence of correlation between the artifacts analyzed and those deployed is "tool integrity." Software tools (i.e., computer programs used to analyze, transform, or otherwise measure or manipulate products of a software development effort) clearly can have an impact on the level of confidence placed in critical software. All of the analysis of source code performed can easily be undermined if we discover that the compiler used on the project is very buggy, for example. In many situations where this is a potential issue (e.g., the certification of digital avionics), a distinction is drawn between two classes of tools:

- Those that transform the programs or data used in the operational system; (and can therefore actively introduce unexpected behavior into the system)
- Those used to evaluate the system (and therefore can at worst contribute to not detecting a defect)

3.2.4 Software Quality Assurance Program

Suppliers of systems having safety-related computing system functions should establish a SQA program guided by an approved SQA plan. The SQA plan should identify and define the interface between the Software Quality Assurance team and the System Safety Engineering team. The System Safety Engineering Team should review the SQA plan and provide recommendations for ensuring that the SQA plan adequately covers likely safety concerns, including the applicable generic requirements and guidelines of this section. The System Safety Engineering Team should evaluate the results of SQA evaluations to ensure that they evaluated safety concerns and that the development team followed safety-related quality assurance guidelines.

3.3 Software Safety Task Implementation

This section of the AOP describes the primary task implementation steps required for a baseline SSS engineering program. It presents the necessary tasks required for the integration of software safety activities into the functional areas of system and software development. Remember, software systems safety (or software safety) is a **subset** of both the system safety engineering process and the software engineering and development process.

As the AOP introduces the software safety engineering process, it will identify the inputs to the described tasks and the products that the specific process step produces. Each program and engineering interface tied to software safety engineering must agree with the processes, tasks, and products of the software safety program and must agree with the timing and scope of effort to verify that it is in concert with the objectives and requirements of each interfacing discipline. If other program disciplines do not agree or do not see the functional utility of the effort, they will usually default to a "non-support" mode.

Figure 3-6 provides a graphical depiction of the software safety activities required for the implementation of a credible SSS program. Remember that the process steps identified in this AOP represent a baseline program that has a historical lessons learned base and includes the best practices from successful programs. Each procurement, software acquisition, or development has the potential and probability to be uniquely diverse: the safety manager *must* use this section as a guide only. The safety manager should analyze each of the following steps and identify where minor changes are required or warranted for the software development program proposed. If these tasks, with the implementation of minor changes, are incorporated in the system acquisition life cycle, the SSS effort has a very high likelihood of success.

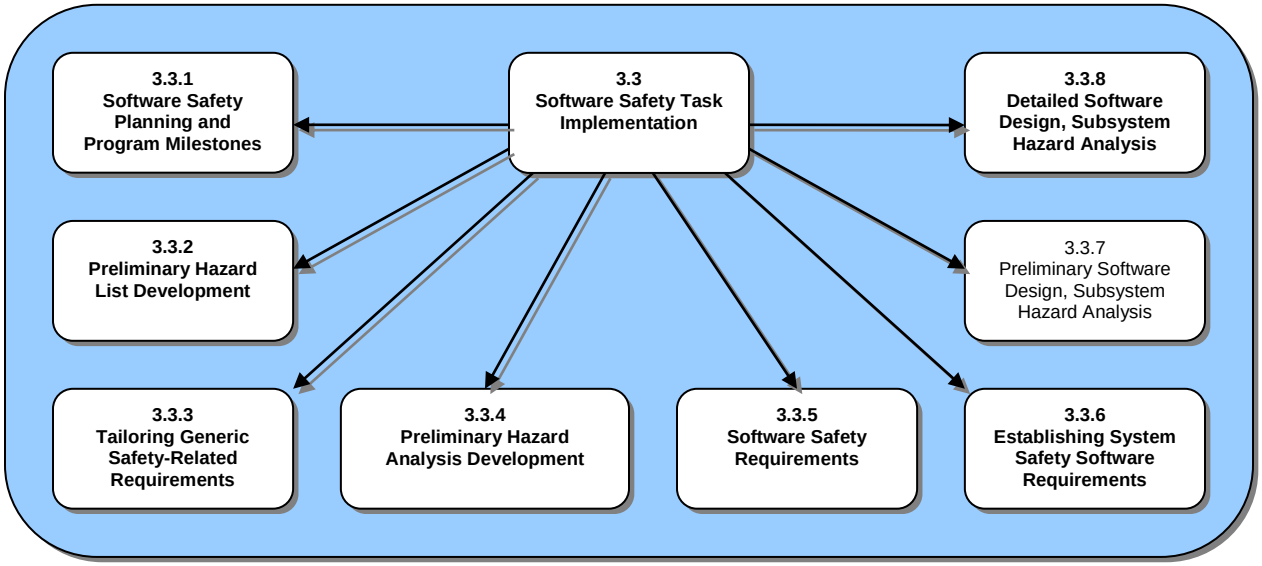


Figure 3-6: Software Safety Task Implementation

The credibility of software safety engineering activities within the hardware and software development project depends on the credibility of the individual(s) performing the managerial and technical safety tasks. It also depends on the identification of a logical, practical, and cost effective process that produces the safety products to meet the safety objectives of the program. The primary safety products include hazard analyses, initial safety design requirements, functionally derived safety design requirements (based on hazard causes), test requirements to produce evidence for the elimination and/or control of the safety hazards, and the identification of safety requirements pertaining to operations and support of the product. The managerial and technical interfaces must agree that the software safety tasks defined in this section will provide the documented evidence for the resolution of identified hazards and failure modes in design, manufacture (code in software), fabrication, test, deployment, and support activities. It must also thoroughly define and communicate residual safety risk to program management at any point in time during each phase of the development life cycle.

3.3.1 Software Safety Planning and Program Milestones

Planning for the software safety must include:

- Identification of the organizations, key personnel roles and responsibilities for assuring the effective implementation and execution of the software safety process
- Reference to competence records (qualifications/training/experience), or reference to a competence management system covering the individuals who will be involved with the assurance of the safety related software
- A description of the safety management interfaces including those with the procurement manager, system design authority, software design authority and subcontractors

- Details of adequate resource planning, including, but not limited to, finance, personnel, equipment and tools
- A list and table of contents of the expected software safety products and documents to be produced
- Safety management processes including scheduling of safety audits and safety reviews
- Details of the certification, regulatory, approval or acquirer acceptance process
- Identification, scheduling, and production of safety assurance deliverables
- Metrics to be collected to monitor the effectiveness of the safety management processes

Each system procurement is unique and will have subtle differences associated with managerial and technical interfaces, timelines, processes and milestones. Program planning must integrate program-specific differences into the schedule and support the practical assumptions and limitations of the program.

The software safety efforts, hazard analysis development, and safety review schedules must support program milestones, for example:

- Software safety requirements resulting from generic requirements tailoring should be available as early as practical in the design process for integration into design, software development, and system safety
- Specific safety requirements from the PHA and an initial set of safety design requirements should be available prior to early design reviews for integration into the design documents
- System safety and software safety must participate in the system specification reviews and provide recommendations during the functional allocation of system requirements to hardware, software, operation, and maintenance. This activity requires an analysis of proposed system and software architectures with the goal of providing recommendations for architectures and functional allocations that best support desirable safety attributes

After functional allocation is complete, the high-level software requirements will be developed. At this point, the preliminary software safety assessment should be completed, with hazards identified and initial software safety criticality indices created. The safety design requirements (hardware, software, and human interfaces) must be complete prior to the milestone at which Software Engineering freezes the requirements (e.g., Critical Design Review). Requirements added after this can have a major impact on program schedule and cost.

During the development of the high-level software requirements, the preliminary software design is analyzed from a safety viewpoint and the system and software architecture assessed to provide design recommendations to reduce the associated risk. These recommendations should be entered in the safety assessment folder and addressed as the design progresses. Further analysis of the design of each module containing safety-related functions and the software architecture will continue throughout development; in the case of safety-critical software, the analysis will extend to the source code to ensure that the intent of the software safety requirements has been met.

Throughout the development, the software safety organization must ensure that the test plans and procedures will provide the desired validation of software safety requirements, demonstrating that they meet the intent of the requirement[3]. Detailed safety tests are derived from the hazard analyses, causal factor analysis, and the definition of software hazard mitigation requirements in order to validate the software safety requirements.

3.3.2 Preliminary Hazard List Development

This process step ensures that the project manager, systems and design engineers, in addition to the software suppliers and engineers, are aware of each safety-related function in the design. It also ensures that software documentation designates each individual module of code that performs these functions as "safety critical" or "safety significant" which mandates defined levels of design and code analysis and testing. Figure 3-7 provides an example of possible safety-related functions of a tactical aircraft.

There are two benefits to identifying the safety-related functions of a system. First, the identification assists the SSS Team in the categorization and prioritization of safety requirements for the software architecture early in the design life cycle. If the software performs or influences the safety-related function(s), that module of code becomes safety-related. This is true at all levels of complexity from programmable logic devices and microcontrollers to complex systems of systems. Second, it reduces the level of activity and resource allocations to software not identified as safety-related: the benefit is cost avoidance.

At this phase of the program, specific ties from the PHL to the software design are immature. At this point in the development, the safety team cannot define many specific hazard causal factors in the software. However, there may be identified hazards, which have preliminary ties to safety-related functions with functional links to the preliminary software architecture. If this is the case, the safety team should document this functional link in the safety analysis for further development and analysis. At the same time, there are likely to be specific "generic" SSRs applicable to the system. These requirements are available from multiple sources and must be specifically tailored to the program as they apply to the system and software architecture.

[3] English Language and other specifications may have multiple interpretations: implementing the intent of a requirement ensures that the software designers properly interpret and implement the desired safety requirements.

Figure 3-7: An Example of Safety-related Functions in a Tactical Aircraft

3.3.3 Tailoring Generic Safety-Related Requirements

Figure 3-8 depicts a process for tailoring the Generic Software Safety Requirements (GSSR). GSSR are those design features, design constraints, development processes, "best practices," coding standards and techniques, and other general requirements levied on a system containing safety-related software, regardless of the functionality of the application. GSSR are not safety specific (i.e., not tied to a specific system hazard). They are, however, based on lessons learned from previous systems where failures or errors occurred that either resulted in a mishap or a potential mishap. The PHL and initial software safety analyses may help determine the disposition or applicability of many individual generic requirements.

Chapter 4 of this AOP contains a set of GSSR. GSSR should be tailored for each specific program. The requirement to implement a tailored set of GSSR may be one of the safety tasks in the Request for Proposal (RFP) or Statement of Work (SOW). Tailoring GSSR is coordination between the Acquiring and Supplier to maximize implementation of GSSR within the overall program system and safety objectives, cost and schedule. Tailoring agreements should be documented, signed by Acquirer and Supplier management, and incorporated into applicable software development (e.g. Software Development Plan) and safety documentation.

Documentation of tailoring agreements is important to both the Acquirer and Supplier. There is typically significant management and personnel turnover over the life of a system development. The tailoring agreement should be valid over the life of the system and changes to the agreement must be accompanied by the commensurate change in cost and schedule to implement, or the potential safety risk of removal of one or more tailored requirements.

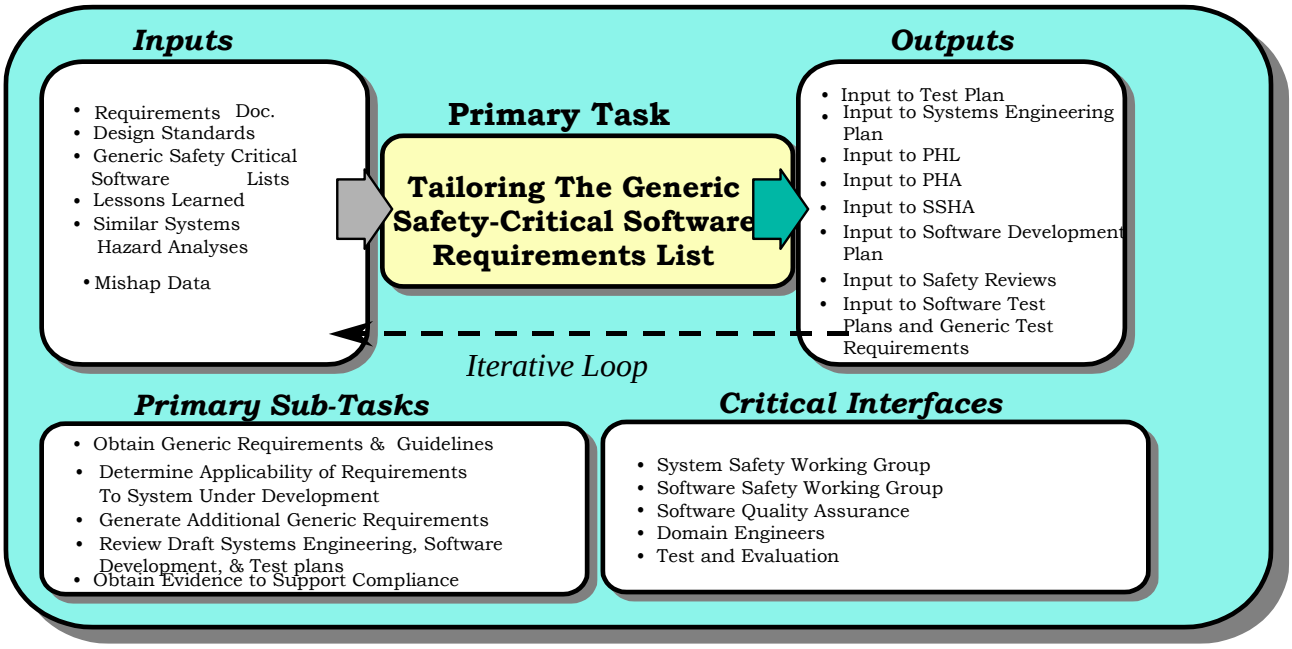


Figure 3-8: Tailoring the Generic Safety Requirements

Tailoring should be performed as early in the program as possible. If the Acquirer can implement some level of tailoring the GSSR prior to Supplier selection, then there may be savings in terms of Supplier cost. Generally, the tailoring effort is performed after a Supplier has been selected. Tailoring GSSR should be the priority activity of the SwSSWG after a Supplier has been selected. Tailoring GSSR should be accomplished during the System Concept and Software Requirements and Architecture Development phases. Tailoring GSSR after the supplier has implemented software development processes, initiated coding, and selected Previously Developed Software (PDS) suppliers is difficult and has cost/schedule impact. The SwSSWG developed GSSR tailoring must be coordinated with the SSWG and program management, and approved by the Acquirer.

GSSR can be levied against PDS suppliers, but are generally not implementable in PDS. However, the PDS supplier can be requested to provide a GSSR compliance matrix and evidence of GSSR that are complied with as a result of the PDS supplier software development processes. This PDS compliance matrix, or lack thereof, should be identified as a safety risk area and provided as an input artifact to the safety case.

Tailoring is not solely a system safety responsibility and the project team should be involved in tailoring safety requirements. The tailoring effort could involve the Software Development Lead, SQA, CM, V&V, and HFI SwSSWG representatives. The Supplier's software development and safety processes are reviewed and assessed to determine compliance with the GSSR in this AOP, or to identify new GSSR that may need to be levied based upon Supplier unique processes. A preliminary compliance checklist, indicating GSSR compliance results (Y, N, N/A), should be developed. These results provide the basis for tailoring of GSSR. The SwSSWG uses the preliminary compliance assessment, initial safety analyses and PHL to determine the additional GSSR necessary. The SwSSWG will develop a recommended complete set of tailored GSSR, with the understanding that the tailoring recommendation will be the basis

for the contractually binding GSSR to be levied on the software developer. The recommended set of tailored GSSR is provided to the SSWG. The SSWG will provide management with the tailored set of GSSR and the rationale for GSSR not already levied on the Supplier. The SwSSWG will support the SSWG in change control actions required.

The results of the tailoring discussions are documented and preserved in the appropriate program artifacts. The Acquirer approved version of the tailored GSSR will be signed and maintained under Configuration Control. The following artifacts may reflect GSSR tailoring agreements:

- Prime Contract
- Subcontracts
- System Safety Management Plan
- System Safety Program Plan
- Software System Safety Program Plan
- Software Development Plan

An evaluation of safety risk must be made for tailored GSSR non-compliances. The safety risk evaluation is input data to Acquirer and Supplier management decisions on whether to apply the resources to comply with the GSSR or to incur the potential safety risks of continued non-compliance. For example, a GSSR non-compliance that has a low safety risk assessment, but high cost and schedule impact for implementation, may be acceptable to management. However, a GSSR non-compliance assessed as medium or high with an acceptable level of cost and schedule impact (i.e. resources available, little or no schedule slip to comply) may be an unacceptable risk.

Table 3-1 is an example of a worksheet form used to track generic SSR implementation. The EVIDENCE block describes whether the program is complying with the requirement, the location of the requirement and its implementation and the location of the evidence of implementation. If the program is not complying with the requirement (e.g., too late in the development to impose a safety kernel) or the requirement is not applicable (e.g., an Ada requirement when developing in assembly language), the RATIONALE block must include a statement of explanation. It should also describe an alternative mitigation of the source risk that the requirement addresses, possibly pointing to another generic requirement on the list.

Generally, the "going in" position for GSSR is to levy the entire set on the Supplier, prior to tailoring. A caution regarding the "blanket" approach of establishing the entire list of guidelines or requirements for a program: each requirement will cost the program critical resources; people to assess and implement; budget for the design, code, and testing activities; and program schedule. Unnecessary requirements will affect these factors and result in a more costly product with little or no benefit. Additionally, each GSSR levied on the Supplier may spawn one or more lower level SRS requirements in order to effectively implement. Thus, the SwSSWG, in cooperation with systems engineering and software engineering, must assess and prioritize GSSR according to the applicability to the development effort. Inappropriate or unnecessary requirements should be tailored out. The SwSSWG must assess each requirement individually and introduce only those that may apply to the development program.

Tailored GSSR must be verified by one or more of the accepted methods (I, A, D, T). Some requirements only necessitate a sampling of evidence to provide implementation (e.g., no conditional GO-TO statements). Others may be verified by audit and inspection. HMI

requirements are typically demonstrated. The Supplier's SwSSWG software developer or quality assurance team member will often be the appropriate individual to gather the implementation evidence of the generic SSRs. The Supplier may assign Software Quality Assurance (SQA), Configuration Management (CM), Verification and Validation (V&V), human factors, software designers, or systems designers to fill out individual worksheets. The SwSSWG should provide approval recommendations for the entire tailored list of completed forms, include it in the SDL, and refer to it in the SAR. The Acquirer representative on the SwSSWG must approve Supplier compliance recommendations and present them to Acquirer management for formal approval. This provides the desirable evidence of generic SSR implementation.

GENERIC SOFTWARE SAFETY REQUIREMENTS IMPLEMENTATION	INTENDED COMPLIANCE		
	YES	NO	N/A
Item: Program Patch Prohibition Are patches prohibited throughout the development process?	X		
Acquirer: (Indicate Acquirer POC, Date, Acceptance, Rejection or assessed Risk Factor for non-compliance)	Risk – N/A (H, M, L)		
Rationale: (If NO or N/A, describe the rationale for the decision and resulting risk.) SW changes coded in source language, SW build under CM control, re-compiled & V&V'd prior to release (discussed at checklist review 1/11/08)			
Evidence: Inspection (If YES, describe the kind of evidence that will be provided. Note: Specify sampling percentage per SwSPP, if applicable.) Reqts. In SDP, SQA audit of SDP, peer review checklists, Audit of CM plan			
Action: (State the Functional area with responsibility.) Software Development POC: Herman Glutz, Lead Software Developer			

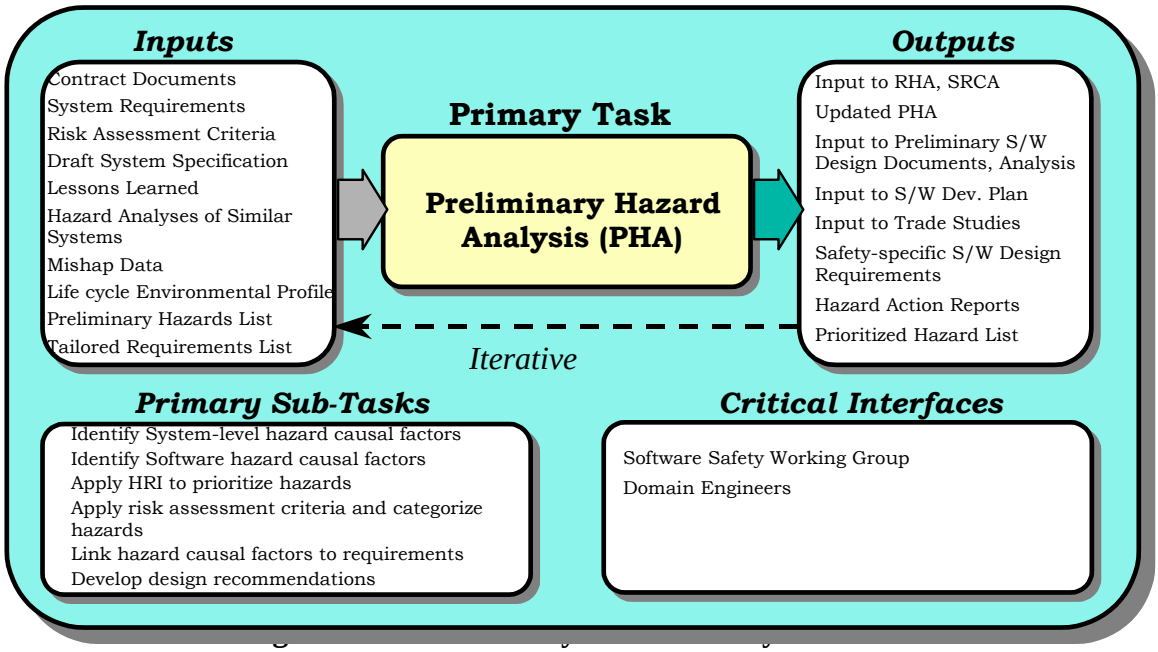
Table 3-1: Generic Software Safety Requirements Tracking Worksheet Example

The SwSSWG should ensure that the tailored list of GSSR is integrated into the appropriate software artifacts and that the software development team understands their responsibility in implementing GSSR. Compliance assessments of GSSR should be performed as a part of each significant software development milestone review. If any GSSR non-compliances are discovered as a result of these assessments, then a safety risk assessment of the non-compliances must be performed, documented in the appropriate safety artifacts, and brought to Acquirer and Supplier management attention.

3.3.4 Preliminary Hazard Analysis Development

The Preliminary Hazards Analysis (PHA) is a safety engineering and software safety engineering analysis performed to identify and prioritize hazards and their causal factors in the system under

development. AOP-15 provides guidance on performing the PHA at the system level. Figure 3-9 depicts the safety engineering process for the PHA. There is nothing unique about the software aspects other than the identification of the software causal factors in the PHA. Many safety-engineering texts provide guidance for developing the PHA therefore, this AOP will not describe the processes. Many techniques provide an effective means of identifying system hazards and the determination of their causal factors.



The PHA provides input to trade-off studies through the early stages of development. The trade-off studies offer alternative considerations for performance, producability, testability, survivability, compatibility, supportability, reliability, and system safety during each phase of the development life cycle. In the early phases of the development many of these tradeoff studies will address system and software architectures and the allocation of system-level functions to the hardware, software and/or operator.

After developing the prioritized list of preliminary hazards, the analysts determine the potential hardware, software, and human interface causal factors to the individual hazards as shown in Figure 3-10. The safety team should use the identified hazard causal factors as a basis to support recommendations for, or against, certain architectures, implementations, and/or functional allocations.

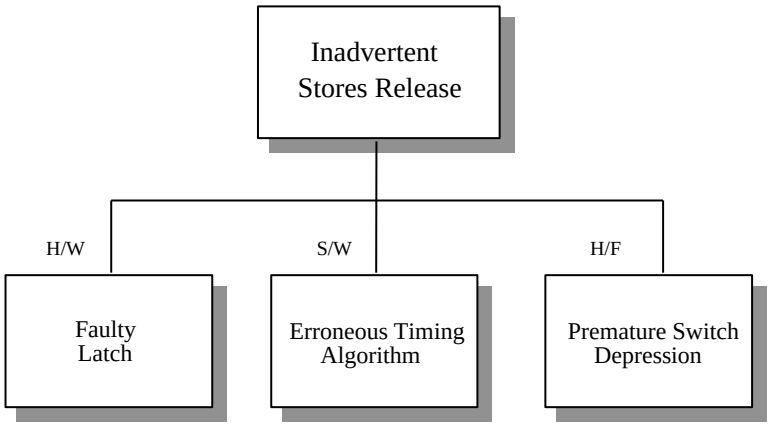


Figure 3-10: Hazard Analysis Segment

This differentiation of hazard causal factors assists in the separation and derivation of specific design requirements for implementation in software. For example, as the analysis progresses, the analyst may determine that software or hardware could contribute to a hardware casual factor. A hardware component failure may cause the software to react in an undesired manner leading to a hardware-influenced software causal factor. The analyst must consider all paths to ensure coverage of the software safety analysis.

Although this tree diagram can represent the entire system, software safety is particularly concerned with the software causal factors linked to individual hazards in addition to ensuring that the mitigation of each causal factor is traceable from requirements to design to code, and ultimately to test procedures that verify its implementation. These preliminary analyses and subsequent system and software safety analyses identify when software is a potential cause or contributor to a hazard, or will support the control of a hazard.

Requirements designed to mitigate the hazard causal factors do not have to be one-to-one, i.e., one software hazard causal factor does not necessarily yield one software safety control requirement. *Safety requirements can be one-to-one, one-to-many, or many-to-one in terms of controlling hazard causal factors to acceptable levels of safety risk.* In many instances, designers can use software to compensate for hardware design deficiencies or where hardware alternatives are impractical. As software is considered cheaper to change than hardware, software safety design requirements may control specific hardware hazard causal factors. In other instances, one design requirement (hardware or software) may eliminate or control numerous hazard causal factors (e.g., some generic requirements). *This emphasizes the importance of performing the hardware safety analysis and software safety analysis at the same time, in consultation with each other.* A system level or subsystem-level hazard may have a single causal factor or a combination of many causal factors. The safety analyst must consider all aspects of what causes the hazard and what is required to eliminate or control the hazard. Hardware, software, and human factors cannot be segregated from the hazard and cannot be analyzed separately except in rare instances. Safety requirements must address how the system will react safely to operator errors, component failures, functional software faults, hardware/software interface failures, and data transfer errors. As detailed design progresses, the system safety team will develop derived software requirements and mature them to address specific hazard causal factors and failure pathways to hazardous conditions or events.

During the PHA activities, the system safety team must establish the link from the software hazard causal factors to the system-level requirements. If there are causal factors that the team cannot link to a requirement, they should report these to the SSWG for additional consideration: the SSWG may have to develop and incorporate additional requirements into the system-level specifications or recommend different implementations.

3.3.5 Software Safety Requirements

3.3.5.1 Introduction

Safety is a property of the overall system rather than any given sub-component. The safety requirements should be identified by system-level hazard analysis and as well as functional behavior, may include timing behavior or capacity, fail-safety, maintainability, modifiability, security and usability. These system safety requirements should be mapped onto a set of functional and non-functional software safety requirements that have to be implemented to a given level of safety integrity.

In defining software safety requirements it is important to consider both positive and negative safety requirements (e.g. "the angle of rotation of the barrel shall be kept within the following limits" and "this function shall be inoperative during the loading process").

Safety functions could be part of the normal, safe operation of the system (e.g. functions of a safety related control or information system), or they could to be functions that are performed in exceptional circumstances (e.g. if the software is part of a protection or emergency system). The software safety requirements should specify only those functions and mitigations that are necessary for safety.

When the safety requirements are applied to individual subsystems (typically as a result of applying hazard analysis methods), a set of derived requirements should be produced for the subsystems that are necessary to support the top-level safety goal. These derived requirements can be represented by attributes of the subsystems that can affect system safety.

For safety related software, it is important that safety requirements are fully and unambiguously defined. Even in the case of an existing, in-service, legacy system, for which none of the original design documentation exists, it is not acceptable from a safety perspective to assume that 'it does what it does'. While it may not be necessary to define detailed functional requirements, it is important that those properties and functions that are essential to safety are defined.

It should also be borne in mind that safety has to be maintained over the lifetime of the equipment. So the system design and the associated safety arguments should consider potential "attacks" on the design integrity over its lifetime (e.g. through normal operation, maintenance, product improvements, upgrades and replacement).

The system supplier needs to specify any assumptions and pre-requisites applicable to the software. Before proceeding with the specification and development of the safety related software, it is necessary to verify that:

- Sufficient information about the system level hazards is available to enable the generation of the safety assurance arguments

- Software safety requirements be complete[4], self-consistent, unambiguous and consistent with the other (non-safety-related) software requirements

Where safety is dependent on the safety related function fully meeting all of its requirements, demonstrating safety is equivalent to demonstrating correctness with respect to the software requirements. In other cases, safety may be dependent on the safety related software behaving in accordance with a smaller identifiable set of safety requirements and satisfying the safety integrity requirements. Because of the difficulties of separating safety characteristics from the other behavioral characteristics of the safety related software and the need to demonstrate adequate partitioning between these characteristics, many projects will choose to treat all requirements as being safety related, with associated level of rigor.

The specification of the software safety requirements should include the following:

- All functional and all non-functional requirements and constraints (e.g. timing resource usage) should be explicitly detailed
- All safety functions and safety attributes should be explicitly identified as derived safety requirements
- Validity and integrity requirements are specified for all safety-related data

3.3.5.2 Safety Integrity Requirements

Safety integrity requirements define the required confidence in the evidence including types of evidence, rigor and extent of safety assurance evidence. The software safety requirements define what the software is required to do but as no software is perfect, it is also necessary to define a tolerable system failure rate. This has to be at a system level as there are currently no accepted methods for measuring failure rates for software, particularly for specific functions in a software configuration. Safety integrity requirements may also specify how the software should meet the safety requirements in terms of timing requirements, use of resources and/or robustness.

The software may be involved in more than one system level hazard and there may be more than one failure mode of the software that contributes to each of the hazards. A safety related software system may provide several safety related services, and different tolerable failure rates may be applicable to these services. In practice, there may be a whole set of safety requirements and safety integrity requirements with which the software must comply.

3.3.5.2.1 Failure Rates

A failure rate is a probabilistic value given to something based on its likely failure in some sort of random or at least complex/non repeatable manner. Usually this is a figure based on the average of a distribution of failures from a large sample. Software and software-like devices have no random behavior in this sense. For example, given a certain routine with inputs in a certain combination it will always fail, conversely in a different configuration with another set of inputs it will never fail. This repeatability with a lack of degradation due to time or use demonstrates that software fails systematically.

[4] Complete in this context means that there are safety requirements identified that mitigate each software related hazard causal factor.

But software always contains faults and the more faults there are then the greater the likelihood that a fault will be exercised whilst the software is running. Whether the fault causes an error severe enough to lead to a failure is another thing. The Safety Integrity Requirements are intended to reduce the number of faults by putting greater emphasis on the development process. However, there is no absolute closed loop measure to say how many faults there are remaining.

3.3.5.2.2 Failure Handling

A failure handling strategy should be defined at the system level and be included in the hardware/software specifications. This will be a design decision in the system documentation and should be defined appropriately in the safety requirements allocated to software. This will be a key safety requirement and should be stated in the software safety plan. It is also possible for the software supplier to identify opportunities to improve failure handling. These should be fed back as derived software safety requirements.

Differences in safety integrity requirements may affect the software design as well as the failure handling strategy. For example if overall availability requirements are less onerous than reliability requirements, the software could shut down in the event of detecting possible corruption to prevent a hazard.

3.3.5.2.3 Confidence

Safety integrity requirements have an additional dimension in the form of the confidence necessary for assurance that safety requirements have been met. For software, confidence would be stated qualitatively in the form of the type, extent, and rigor of the evidence necessary to support the arguments for the software safety claims.

3.3.5.3 Safety Integrity Level Schemes

While many standards implicitly treat confidence qualitatively by specifying different sets of processes for various integrity levels, it is recommended that an evidence-based approach be used. This requires that evidence is produced to show that the requirement is met, and also that the quantity and quality of the evidence is commensurate with the risk.

There are a number of safety integrity level schemes offered by other standards. However, where a safety integrity level scheme is required by a project any scheme to be used should be agreed by all interested parties.

3.3.5.4 Derived Safety Requirements

The arguments for the safety of the software should start with the system architecture. The safety argument should demonstrate why the system architecture implements desirable safety features. The architectural safety argument will support the claim that the safety requirements will be satisfied if correctly[5] implemented in the hardware and software components.

When safety attributes are implemented in a system design, the hardware and software elements must not be considered in isolation. In order to implement the safety attributes, the derived safety requirements will impose additional requirements to the software and hardware components. For example, in order to maintain the overall integrity of the system, the software

[5] The term “correctly” is not synonymous with the software engineering term “correctness”.

may rely on separate checking hardware (such as external watchdogs), while the hardware status might be monitored using software.

However complete the software requirements, there will be times when the choice of implementation leads to additional derived requirements. For example, the division of the software system into smaller subsystems would result in interface requirements; the choice of a particular microprocessor and memory unit will result in specific timing and capacity requirements; the decision to write defensive code may result in the source code having a need to report particular failures via failure codes.

The non-functional attributes at the system architecture level can lead to requirements for additional functionality within the software. These can be a major source of complexity for the software. In many projects the software required for the actual core communication, information, control and protection functions can be quite modest compared with the code needed to implement these additional functions.

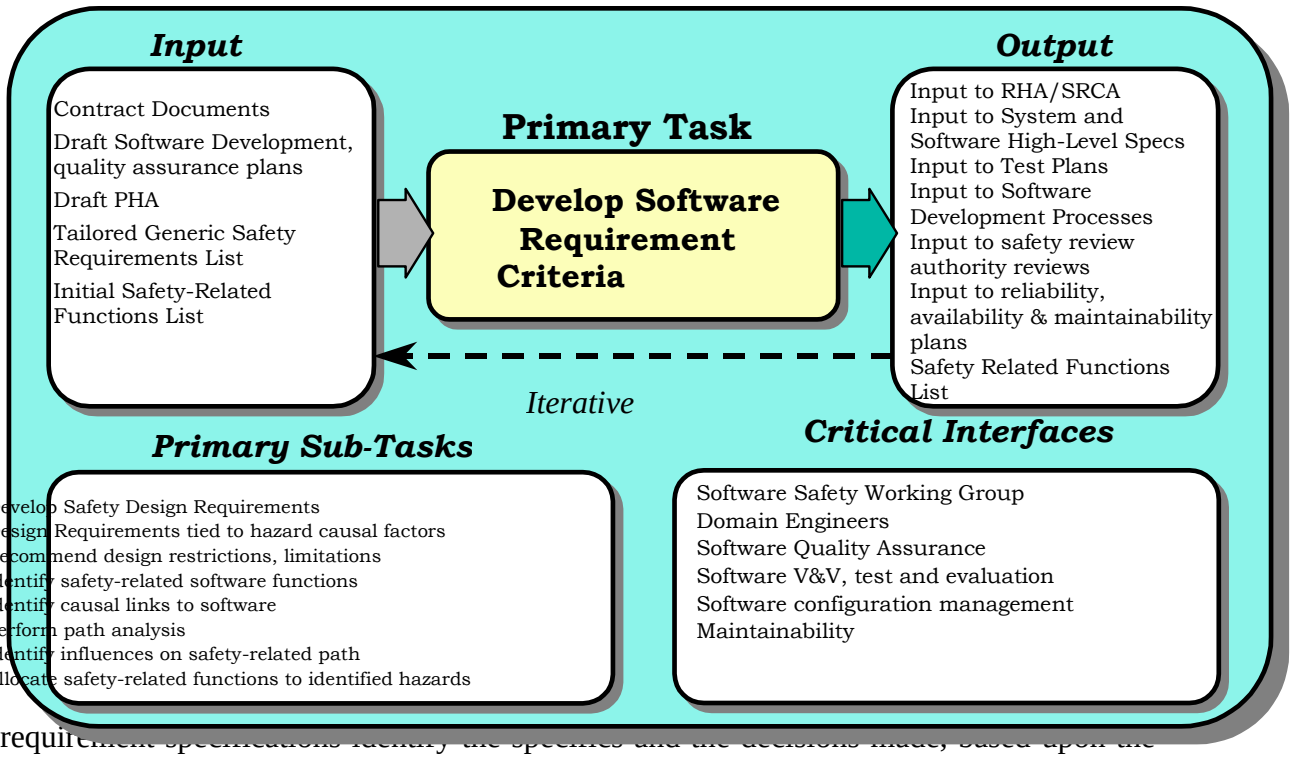
The safety requirements at the software level consist primarily of functional requirements (e.g. to provide control functions, process information, implement required hardware diagnostics, or to service a watchdog) but some attributes may simply be targets to be achieved by the software design (e.g. tolerable failure rates, worst case time response, or security levels). Since attributes at high level of design can be changed into functional requirements as the design proceeds, it is important to maintain traceability of all safety requirements between the various levels of the overall Safety Case.

Safety requirements that evolve during the course of the development of the software (derived safety requirements) should be fed back to the system design process for incorporation into the higher-level system requirements or software safety requirements in accordance with project processes. The Software Design Authority should ensure that the eventual users and maintainers of the software have visibility of relevant safety requirements.

3.3.6 Establishing System Safety Software Requirements

The safety team derives safety-related SSRs from known safety-related functions, tailored generic SSRs, and hazard causal factors determined from previous activities.

Figure 3-11 identifies the software safety engineering process for developing the SRCA.



Safety requirement specifications identify the specifics and the decisions made, based upon the level of safety risk, desired level of safety assurance, and the visibility of software safety within the supplier organization. Methods for doing so are dependent upon the quality, breadth, and depth of initial hazard and failure mode analyses and on lessons learned and/or derived from similar systems. As stated previously, the generic list of requirements and guidelines establishes the starting point, which initiates the system-specific SSR identification process. Identification of system-specific software requirements is the direct result of a complete hazard analysis methodology (see Figure 3-12).

The safety team derives SSRs from four sources: generic lists, analysis of the system functionality (safety design requirements), causal factor analysis, and from implementation of hazard controls. The analysis of system functionality identifies those functions in the system that, if not properly executed, can result in an identified system hazard. Therefore, the correct operation of the function related to the safety design requirements is critical to the safety of the system making them safety-related as well. The software causal factor analysis identifies lower-level design requirements that, based on their relationship to safety-related functions or the context of the failure pathway of the hazard make them safety-related as well. Finally, design requirements developed to mitigate other system-level hazards (e.g., monitors on safety-related functions in the hardware) are also SSRs.

The Software Safety Engineer must present the SSRs to the acquirer via the SwSSWG for concurrence with the assessment as to whether they eliminate or resolve the hazardous condition to acceptable levels of safety risk prior to their implementation. For most SSRs, there must be a direct link between the requirement and a system-level hazard. The following paragraphs provide additional guidance on developing SSRs other than the generics.

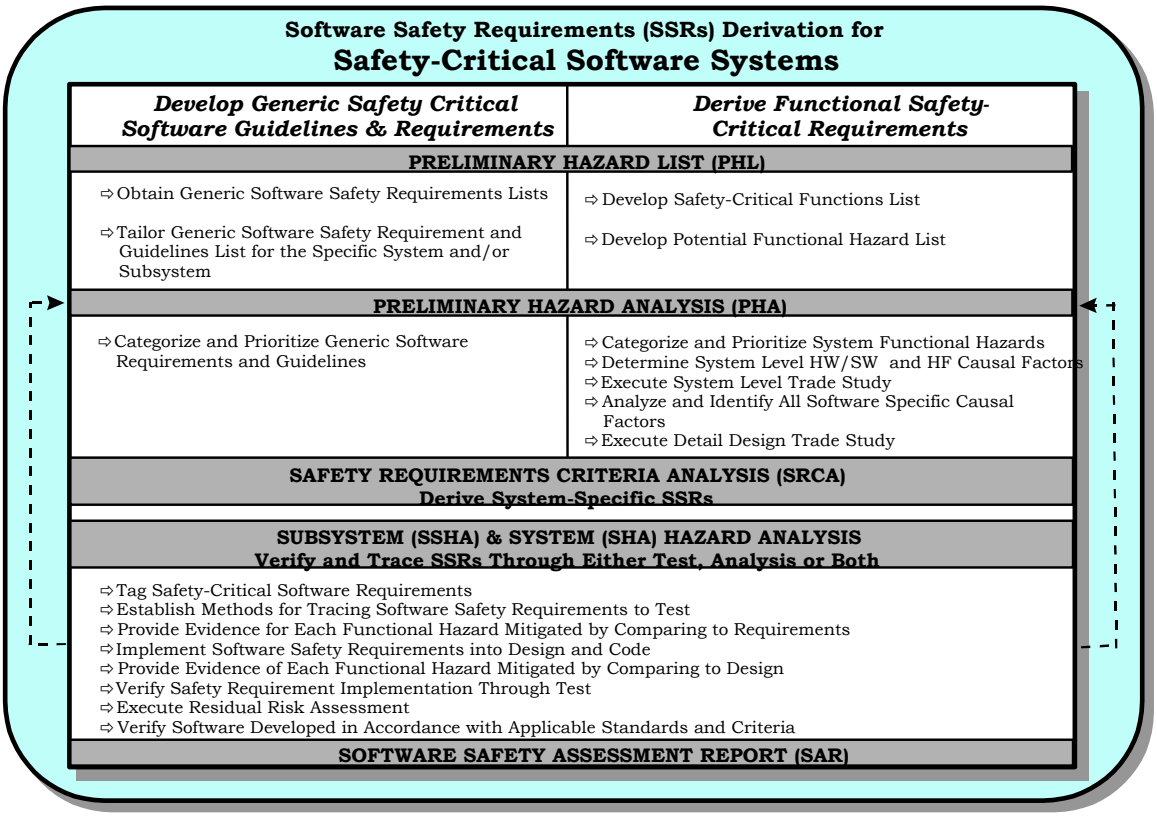


Figure 3-12: Software Safety Requirements Derivation

3.3.6.1 Preliminary Software Safety Requirements

The initial attempt to identify system-specific SSRs evolves from the PHA performed in the early phase of the development program. As previously discussed, the PHL/PHA hazards are a product of the information reviewed pertaining to systems specifications, lessons learned, analyses from similar systems, common sense, and preliminary design activities. The analyst ties the identified hazards to functions in the system (e.g., inadvertent rocket motor ignition to the rocket motor ARM and FIRE functions in the system software). The analyst flags these functions and their associated design requirements as safety-related and enters them into the Requirements Traceability Matrix (RTM) within the SRCA. The analyst should develop or ensure that the system documentation contains appropriate safety requirements for these safety-related functions (e.g., ensure that all safety interlocks are satisfied prior to issuing the ARM command or the FIRE command). Lower levels of the specification will include specific safety interlock requirements satisfying these preliminary SSRs. These types of requirements are safety design requirements.

The safety engineer also analyzes the hazards identified in the PHA to determine the potential contribution by the software. For example, a system design requires the operator to actually commit a missile to launch; however, the software provides the operator a recommendation to fire the missile. This software is also safety-related and must be designated as such and included in the RTM. Other safety-related interactions may not be as obvious and will require more in-depth analysis of the system design. The analyst must also analyze the hazards identified in the

PHA and develop preliminary design recommendations to mitigate other hazards in the system. Many of these design recommendations will include software thus making that software safety-related as well. During the early design phases, the safety analyst identifies these requirements to design engineering for consideration and inclusion. This is the beginning of the identification of the functionally derived SSRs.

These design considerations, along with the generic SSRs, represent the preliminary SSRs of the system, subsystems, and their interfaces (if known). The safety team must accurately define these preliminary SSRs in the hazard tracking database for extraction when reporting the requirements to the design engineering team.

3.3.6.2 Matured Software Safety Requirements

As the system and subsystem designs mature, the requirements unique to each subsystem also mature via the SSHA. The safety engineer, during this phase of the program, identifies and defines the subsystem hazards. The safety engineer documents the identified hazards in the hazard tracking database and analyzes the hazard causal factors. When using fault trees as the functional hazard analysis methodology, the causal factors leading to the root hazard determine the derived safety-related functional requirements. At this point in the design, the safety formalizes and defines preliminary design considerations or eliminates them if they no longer apply with the current design concepts. The SSRs mature through analysis of the design architecture to connect the root hazard to the causal factors. The analyst continues the causal factors' analysis to the lowest level necessary for ease of mitigation.

This helps mature the functional analysis started during preliminary SSR identification. The deeper into the design that the analysis progresses, the more simplistic (usually) and cost effective the mitigation requirements tend to become. The safety team may also derive additional SSRs from the implementation of hazard controls (i.e., monitor functions, alerts to hazardous conditions outside of software, unsafe system states, etc.). The PHA phase of the program should define causes to the lowest level practical in the software for the stage of development whereas the SSHA and SHA should analyze the causes to the algorithm level for areas designated as safety-related.

3.3.6.3 Subsystem Hazard Analysis

The subsystem analysis begins during concept exploration and continues through the detailed design until design requirements are "frozen". The safety analyst must ensure that the safety analyses keep pace with the design. As the design team makes design decisions and defines implementations, the safety analyst must reevaluate and update the affected hazard records.

3.3.6.4 Documenting Software Safety Requirements

The SRCA should document all identified SSRs. The objective of the SRCA is to ensure that the implementation of the requirements in the system software meets the intent of the SSRs and that the SSRs eliminate, mitigate, and/or control the identified causal factors. Mitigating and/or controlling the causal factors reduce the probability of occurrence of the hazards identified in the PHA. The SRCA also provides the means for the safety engineer to trace each SSR from the system level specification, to the design specifications, to individual test procedures and test results' analysis. The safety engineer uses this traceability, known as a RTM, to verify that all SSRs can be traced from system level specifications to design to test. The safety engineer should

also identify all safety-critical SSRs to distinguish them from safety-significant SSRs in the RTM. Safety-critical SSRs are those that directly influence a safety-critical function (software control categories 1 and 2), while safety-significant SSRs are those that indirectly influence safety-critical functions or directly influence safety-significant functions. The RTM provides a useful tool to the software development group. They will be immediately aware of the safety-critical and safety-significant functions and requirements in the system. This will also alert them when making modifications to safety-critical software configuration items and modules that may affect SSRs. The SRCA is a "living" document that the analyst constantly updates throughout the system development.

3.3.6.5 Software Analysis Folders

At this stage of the analysis process, it is also a good practice to start the development of Safety Analysis Folders (SAFs). The purpose of a SAF is to serve as a repository for all of the analysis data generated on a particular software configuration item. The safety team should develop SAFs on a software configuration item basis and make them available to the entire SSS Team during the software analysis process. Items to be included within the SAFs include, but are not limited to:

- Purpose and functionality of the software configuration item, source code listings annotated by the safety engineer
- Safety-Related Functions (SRF) and SSRs pertaining to the software configuration item under analysis, SSR traceability results
- Test procedures and test results pertaining to the software configuration item
- Record and disposition of all Program Trouble Reports (PTR)/Software Trouble Reports (STR) generated against the particular software configuration item
- A record of any and all changes made to the software configuration item

The safety team needs to update the SAFs continuously during the preliminary and detailed design SSHA phases. The SAFs should include the results of tests that verify the "safe" implementation of the software configuration item. The SAF should be referenced in the safety case.

3.3.6.6 Programmable Logic Device Analysis Folders

The safety team should develop PLD analysis folders (PAFs), or similar, for safety-related PLDs. The PAF is similar in function and content to the SAF. The safety team should develop the folders on a device basis and make them available to the SSS team throughout the development process. The PAFs should include at a minimum:

- Hardware item configuration data
- Schematics of the circuit containing the PLD
- Purpose and functionality of the PLD
- Source code listings annotated by the safety engineer

- Safety-Related Functions (SRF) and SSRs pertaining to the item under analysis, SSR traceability results
- Test procedures and test results pertaining to the item
- Record and disposition of all Program Trouble Reports (PTR)/Software Trouble Reports (STR) generated against the particular item
- A record of any changes made to the PLD software
- A record of any changes made to the PLD hardware and associated circuits
- A copy of the master net-list for the PLD

The SSS team updates the PAF continuously during the preliminary and detailed analysis phases and includes test data demonstrating the desired risk mitigation. The PAF should be referenced in the safety case

3.3.7 Preliminary Software Design, Subsystem Hazard Analysis

The identification of subsystem and system hazards and failure modes inherent in the system under development is essential to the success of a credible software safety program. Today, the primary method of reducing the safety risk associated with software performing safety-critical or safety-significant functions is to first identify the system hazards and failure modes and then determine which hazards and failure modes are *caused* or *influenced by* software or lack of software. This determination includes scenarios where information produced by software could potentially influence the operator into a wrong decision resulting in a hazardous condition (design-induced or information-induced human error). Moving from hazards to software causal factors and consequently to design requirements that eliminate or control the hazard allows for traceability of the hazards and their mitigations for future reference. If performed in a timely manner, the analysis can influence preliminary design activities with little or no impact on the overall development effort.

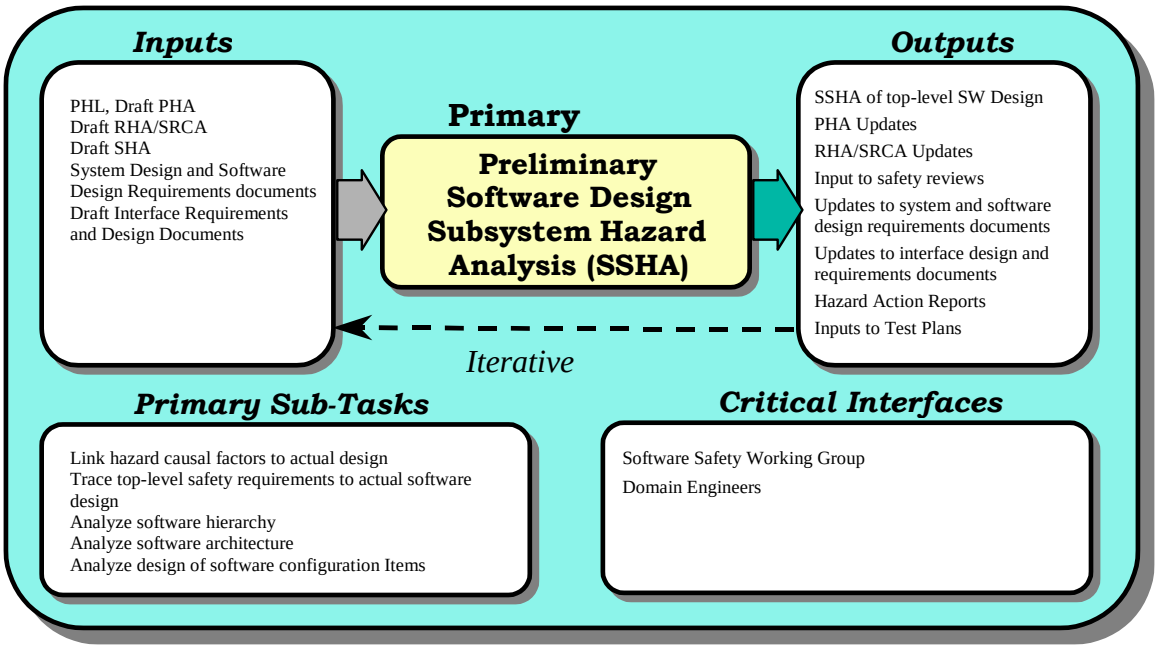


Figure 3-13: Preliminary Software Design Analysis

The fundamental basis and foundation of a SSP is a systematic and complete hazard analysis process. One of the most helpful steps within a credible software safety program is to categorize the specific causes of the hazards and software inputs in each of the analyses (PHA, SSHA, SHA, and O&SHA). Hazard causal factors can be those caused by hardware (e.g., failure of a hardware component), software inputs (or lack of software input), software design errors, software implementation errors, human error, software-influenced human error, or hardware or human errors propagating through the software… Hazards may result from one specific cause or any combination of causes. As an example, "loss of thrust" on an aircraft may have causal factors in different categories. Examples are as follows:

- **Hardware**: foreign object ingestion
- **Software**: software commands engine shutdown in the wrong operational scenario
- **Human error**: pilot inadvertently commands engine shutdown
- **Software-influenced pilot error**: computer provides incorrect information, insufficient or incomplete data to the pilot causing the pilot to execute a shutdown (e.g., erroneous Engine Fire signal)

Whatever the cause, the safety engineer must identify and define hazard control considerations (PHA phase) and requirements and implementation recommendations (SSHA, SHA, and O&SHA phases) for the design and development engineers for each individual causal factor. The preliminary software design SSHA begins upon the identification of the software subsystem and uses the derived system-specific SSRs. The purpose is to analyze the system and software architecture and preliminary software configuration item design. At this point, the analyst has identified (or should have identified) all SSRs (i.e., safety design requirements, generics, and functional derived requirements and hazard control requirements) and begins allocating them to the identified safety-related functions and tracing them to the design.

The choice of analysis and/or testing to verify the SSRs is up to the SwSSWG, whose decision is based on the criticality of the requirement to the overall safety of the system and the nature of the SSR.

The next step of the preliminary design analysis is to trace the identified safety requirements and causal factors to the design (to the actual software components). The RTM is the easiest tool to accomplish this task. Note that the RTM is just as useful for the safety assessment of PLDs as it is for software.

SSR	Requirement Description	SW Configuration Item	SW Module	Test Procedure	Test Results	Analysis Results

Table 3-2: Example of a partial RTM

Note: All software safety requirements should be traceable to system-level safety requirements.

3.3.7.1 Component Safety-Criticality Analysis

The purpose of the component safety-criticality analysis is to validate the appropriate level of development, validation & verification, configuration management and quality assurance activities to perform for this component. The safety analyst bases the level of activities on the hazard, in which the component is implied that have the highest criticality. The analyst can then add some architectural considerations to lower this level by taking into account such solutions has : redundancy, dissimilarity, monitoring, etc …

3.3.7.2 Traceability Analysis

The analyst develops and analyzes the RTM to identify where the SSRs are implemented in the code, SSRs that are not being implemented, and code that does not fulfill the intent of the SSRs. The traced SSRs should not just be those identified by the top-level specifications, but those identified by the software requirements and design documentation as well as the interface requirements and design documentation. This trace provides the basis for the analysis and test planning by identifying the SSRs associated with all of the code. This analysis also ties in nicely with the SRCA (see Section 3.3.6), which not only traces SSRs from specifications to design and test but also identifies what is safety-related and what is not.

Tracing encompasses two distinct activities: a requirement-to-code trace and a code-to-requirement trace. The forward trace, requirement-to-code, first identifies the requirements that belong to the functional area (if they are not already identified through requirement analysis). The forward trace then locates the code implementation for each requirement. A requirement may be implemented in more than one place thus making the matrix format very useful.

The backward trace, code-to-requirement, is performed by identifying the code that does not support a requirement or a necessary "housekeeping" function. In other words, the code is extraneous (e.g., "debugging" code left over from the software development process). The safety analyst performs this trace through an audit of the applicable code after he/she has a good understanding of the corresponding requirements and system processing. Code that is not traceable should be documented and eliminated if practical. The following items should be documented for this activity:

- Requirement-to-code trace
- Unit(s) [code] implementing each requirement
- Requirements that are not implemented
- Requirements that are incompletely implemented
- Code-to-requirement trace
- Unit(s) [code] that are not directly or indirectly traceable to requirements or necessary "housekeeping" functions

3.3.8 Detailed Software Design, Subsystem Hazard Analysis

Detailed-design level analysis (Figure 3-14) follows the preliminary design process that traced the software safety requirements to the software configuration item level. Prior to performing this process, the safety engineer should complete development of any fault trees for all of the top-level hazards, identifying all of the potential software-related hazard causal factors and deriving generic and functional safety design requirements for each causal factor.

This section provides the necessary guidance to perform a Detailed Design Analysis (DDA) at the software architecture level. It is during this process that the SSE works closely with the software supplier and the verification & validation engineers to ensure that the implementation of the safety design requirements meets the intent, and to ensure that their implementation does not introduce any other potential safety concerns.

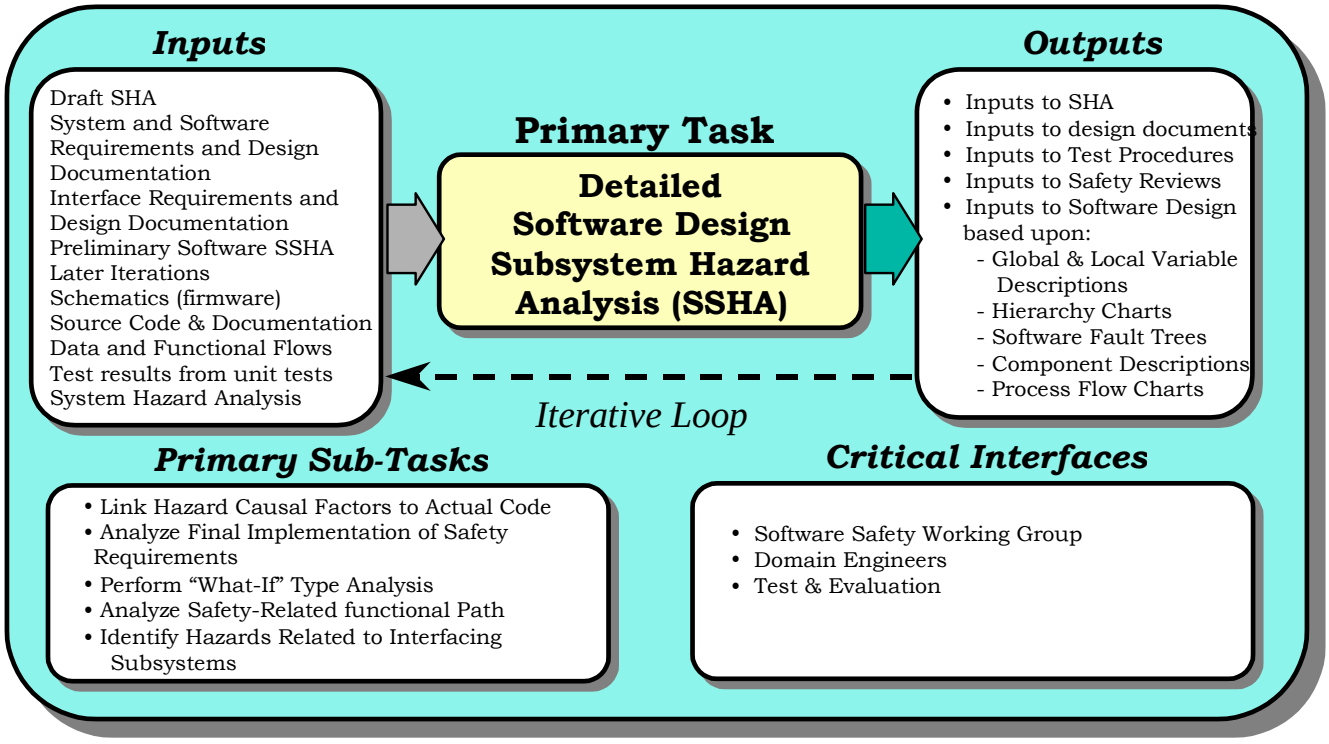


Figure 3-14: Detailed Software Design Analysis

3.3.8.1 Detailed Design Analysis

Detailed Design Analysis (DDA) provides the software safety engineer and the software development engineers an opportunity to analyze the implementation of the software safety requirements at the software unit level. The analysis begins at the software configuration item level determined from the preliminary design analysis into the computer software architecture implementation. As the software development process progresses from preliminary design to detailed design and code, the safety engineer must provide the software safety requirements to the appropriate engineers and programmers of the software development team. In addition, the safety engineer must monitor the design and development process to ensure that the software engineers are implementing the requirements into the architectural design concepts at all levels of granularity. This requires real-time, interactive communication with the software engineers. The software safety engineer does not need to be an expert in all computer languages, software development methodologies, and software architectural patterns. Software design reviews, code walkthroughs, and technical interchange meetings will provide a conduit of information flow for the safety engineer's assessment of the software development program from a safety perspective. The assessment should include how well the software design and programming team understands the system hazards and hazardous failure modes attributed to software inputs or influences. It also includes their willingness to assist in the derivation of safety-specific requirements, their ability to implement the requirements, and their ability to derive test cases and scenarios to verify the resolution of the safety hazard.

There are four methods of verifying software safety requirements: inspection, analysis, testing, and demonstration. Later sections of this AOP discuss recommended approaches and techniques for analysis as well as approaches for SSR verification through testing.

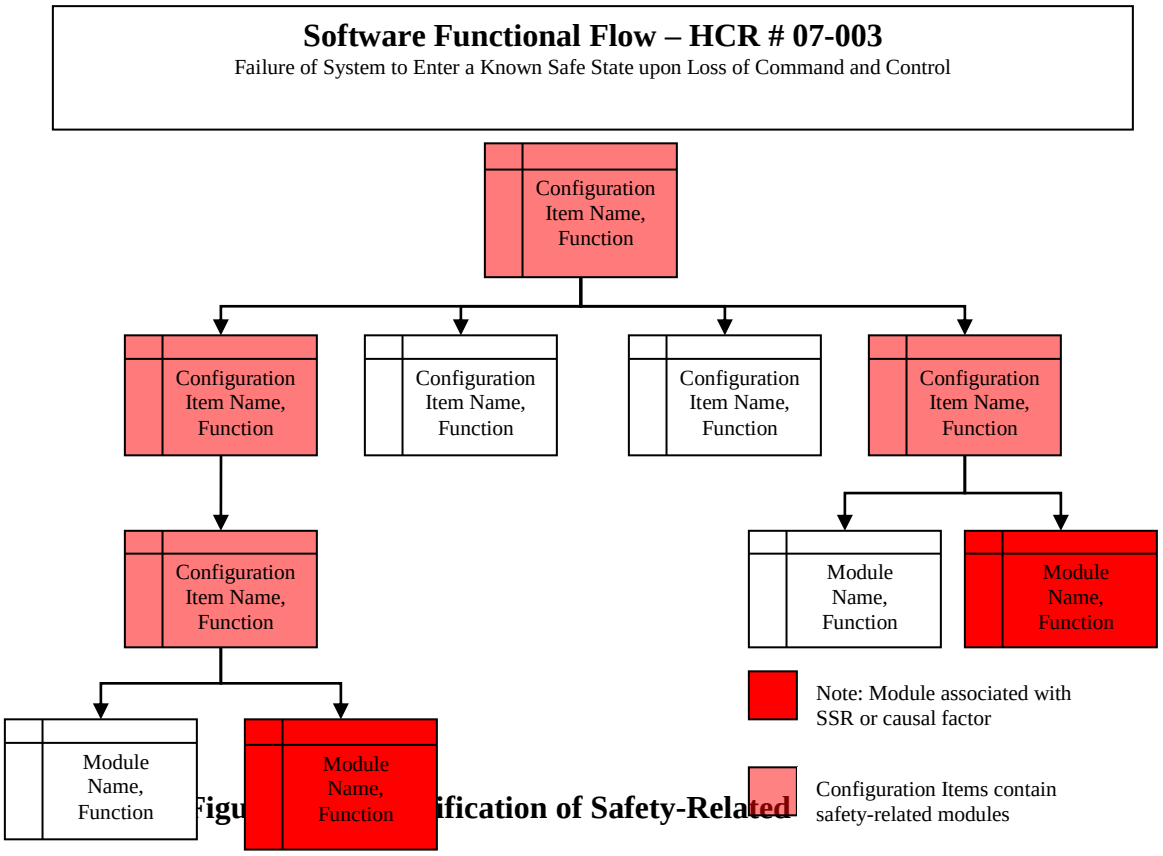
3.3.8.2 Detailed Design Software Safety Analysis

One of the primary analyses performed during DDA is the identification of software units that implement software safety requirements. The term software unit refers to the code-level routine, function, or module. The best ways to accomplish this task is for the software safety team to meet with the software supplier, test engineer, or QA engineer and begin to link individual software safety requirements to software units, as illustrated in Figure 3-15. This accomplishes two goals: First, it helps focus the software safety team on the safety-related processing, which is more important on large-scale development projects than on smaller, less complex programs. Secondly, it provides an opportunity to continue development of the RTM. A critical aspect of this portion of the analysis is identifying interfacing functions to the safety-related modules.

As a result of the analysis, the software safety engineer will likely identify additional safety issues and develop design and implementation recommendations to mitigate the risks. The following list provides examples of techniques and processes that the software safety engineer may use.

- Safety Interlocks
- Checks and Flags
- Firewalls

- Come-from programming
- Bit Combinations
- "What If" analysis



Figu ification of Safety-Related

Detailed design analysis also allows the system safety team to identify potential hazards related to interfacing systems. Erroneous safety-related data transfer between system-level interfaces can be a contributing factor (causal factor) to a hazardous event. Interface analysis should include the identification of all safety-related data variables while ensuring that the software exhibits strong data typing for all safety-related variables. The interface analysis should also include a review of the error processing associated with interface message traffic and the identification of any potential failure modes that would result if the interface fails or the data transferred is erroneous. The safety team should tie identified failure modes to the identified system-level hazards.

3.3.8.3 Detailed Design Analysis Related Sub-Processes

The following list contains some analysis techniques that are useful in performing detailed design analysis of safety-related functionality in software. None of these analysis techniques alone will provide all of the information required to assess and/or mitigate the risk. Therefore, individual programs may require a combination of analysis techniques to adequately address the risk.

- Process flow diagrams
- Data flow diagrams
- Code Level Analysis
- Data Structure Analysis
- Control Flow Analysis
- Interface Design Analysis
- Interrupt Analysis
- Analysis by Inspection

3.3.9 Safety Risk as a System Property

Safety Risk is a system property, not a property of the components or subsystems, including the software, that comprise the system. Components, subsystems, and software exhibit certain safety-related attributes that affect the overall risk associated with the system. These safety attributes are a function of the interaction of the component with the safety-related functionality of the system, failure modes of the component and their impact on the system, and the reaction of the component to other failures in the system. Therefore, hazard causal factors are assessed for mishap potential at the system level. Further, just as changes within a system affect the safety-related functionality of components within the system, changes to the system context will subsequently change the safety-related functionality and/ or the interaction of components with that functionality, and thus change the safety properties of the system. This is true whether the boundaries of the system remain the same or the boundaries of the system increase, which happens when we integrate a system with another system or into a larger system of systems. We may also change the criticality of the system's or a component's functionality in the context of the new system configuration or environment.

3.3.10 System Hazard Analysis

The purpose of the System Hazard Analysis (SHA) is to:

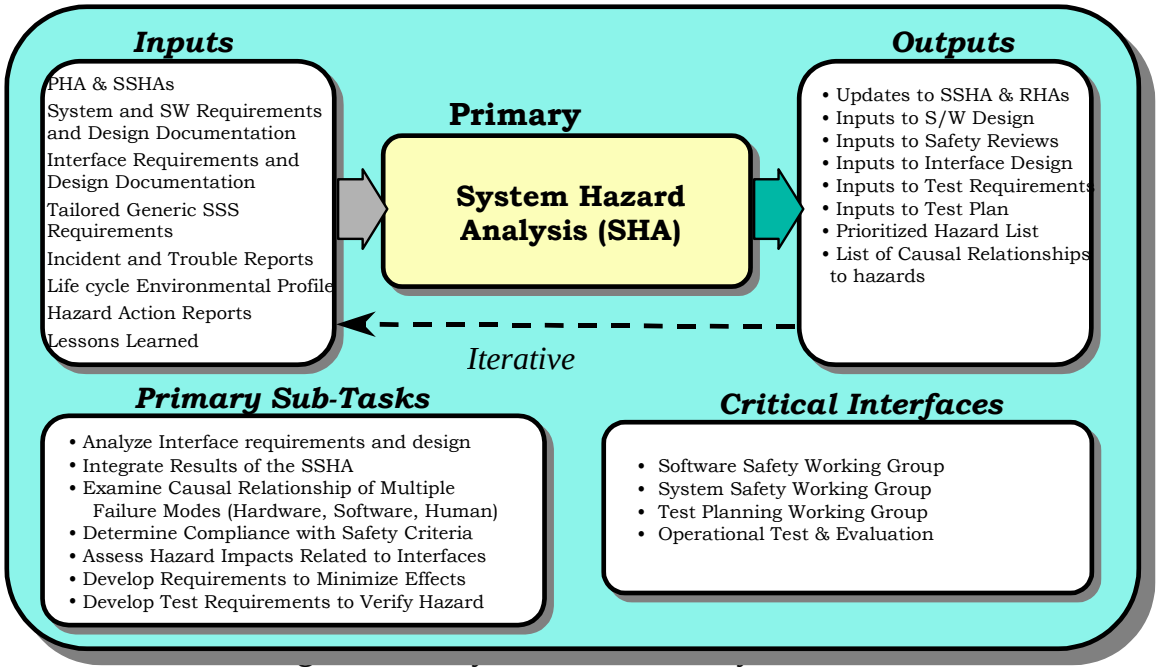
- Verify system compliance with safety requirements and other applicable documents. These system-level requirements and specifications are traceable "down" to the software requirements. The software requirements analyses are compared back "up" to those safety-related system level specifications to assure that all system hazards are mitigated at the software sub-system level and that lower derived software requirements do not introduce new system level hazards.
- Identify previously unidentified hazards associated with the subsystem interfaces and system functional faults. The lower level interface analyses results are compared back "up" with the system interfaces to assure that the original safety risk level assigned to each system interface is still the same level. Differences are highlighted when detailed design and software architecture analyses results are:
    - Compared "up" to original system level hazards,
    - Assessed whether low level design and code faults induce system level hazards,

  - Assessed to assure that system level mitigations are not bypassed or left in an unsafe state by low level functional faults.
  - Traced back to software and system architecture components previously declared as safety-related software system components
  - Used to highlight the hazards which are dependent on PDS and where that PDS operates within the architecture and interface of the system.

- Identify and confirm:
  - Original software system interface safety risk level assignment agrees with feedback from low level safety risk level interface assignments;
  - low level safety risk level assignments to components in the architectures agree with system level component safety risk level assignments and partitions:
    - selected operational safety threads are traced through and highlight each system interface, software interface, and software component in architectures (software structure, software interfaces, software activity or node architecture, and data exchange(s));
  - software system architecture fault tolerance designs (partitions, initializations, shut-downs, status monitoring, stand-by functions) and interfaces to system level architecture fault tolerance design and functionality is correct;
  - residual actions necessary for identified hazards:
    - document their associated risk
    - the stakeholder (may be a different interfacing system supplier, a subcontractor supplier, GOTS, or a COTS supplier) responsibilities with respect to the hazards (acceptable, change required, fault tolerance to be increased) has been determined
    - recommend further mitigation to achieve acceptable safety level
    - recommend appropriate system level V&V (inspections, test cases, and especially demonstrations – since system testing may not be discrete enough to close interface hazard).

The software contribution to an SHA can begin once the project has an approved Preliminary Hazard Analysis (PHA), the system requirements have been linked to the systems hazards, system architecture (at least the physical, activity, and data exchange models) have been designed, and at least some of the requirements have been allocated to software functions. SHA updates are made for each software milestone review (Software Requirements Review, Preliminary Design Review (PDR), Critical Design Review (CDR), Test Readiness Review, Initial Operating Commencement (IOC), Low Rate of Production Decision, and Initial Operating Capability), and for Review Authority submittals. Key SHA events are requirements reviews, swPDR, swCDR, system integration, and IOC. The SHA is iteratively updated throughout the development process. The SHA is complete when software changes cease and the final version is delivered

Figure 3-16 graphically represents the inputs, outputs, and primary sub-tasks associated with the SHA activity. Like other software related analyses the SHA is iterative in nature: changes to architecture elements and issues raised during integration will require updating analysis portions. The inputs to the SHA iterations include: all lower level design and interface analyses results, hardware causes, software causes, and human error causes.



In a majority of the hazards, the in-depth causal factor analysis will identify failure modes (or causal factor pathways) which will cross physical subsystem interfaces, functional subsystem interfaces, and even contractor/subcontractor interfaces. Figure 3-17 graphically depicts the crossing of interfaces and subcontracts.

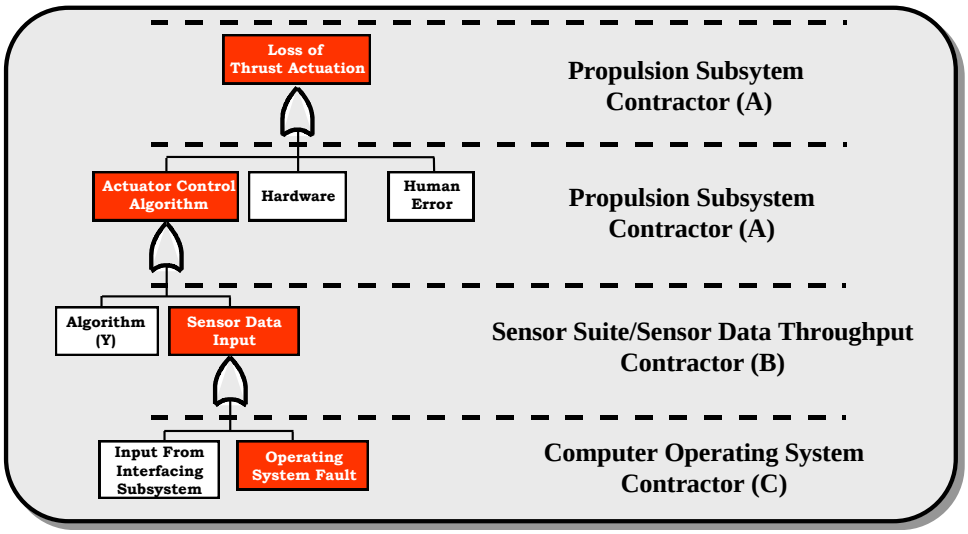


Figure 3-17: System Hazard Analysis Interface Analysis Example

The software safety threads also delineate timing and sequence, which is dependence, of software functions at interfaces. It is possible to meet interface requirements tests but fail integration testing due to data refresh and sequence issues. The threads also have a physical attribute that highlights pathways through the computer hardware, data busses, memory, data structures, and PDS. The safety analyst must ensure that both hardware and software design architecture safety risk assignments, or "tags", comply with the architectures, design specification, safety requirements, and interface descriptions.

3.3.11 Systems Integration and System of Systems (SoS)

Systems integration is a crucial development phase which should be supported with software safety system hazard analyses, interface analyses, and safety test analyses and results. Lessons learned highlight that this phase is prone to new faults leading to new hazards, especially affecting the SHA.

Integration into a System of Systems is more difficult and complex than systems integration, especially due to the lack of binding contracts and up-to-date information among SoS members to assist them as they integrate together

3.3.11.1 System Integration

Integrating even stable systems invariably affects the safety properties of the each system. Integration of systems may happen in a variety of ways, each with a different distribution of safety risks:

- Interfacing one system within another system (e.g., linking a sensor and tracking system to a weapon control system which makes all firing decisions) creating one complex system where safety monitor and control is centralized.
- Incorporating one system within another system (e.g., incorporating a sensor system within a weapon or fire control system but in a closed system with no outside system sharing data or commands) creating one complex system, with safety still "centralized" but interfaces and timing become more difficult.
- Integrating systems together over a communications network (e.g., a battle space coordination created by digitally interfacing weapon systems and sensor systems) a global integration but each system only shares advisory awareness data and no system commands another. Safety continues to be "local" only but inputs and redundancy are remote.
- Integrating systems through a command and control system (e.g., a battle space command and control with a capability to command an interfacing weapon systems to fire based on safe data received from a remote sensor systems, all transporting data over a network system that interfaces individually with each) to create a distributed system of systems . This is the most difficult safety monitor and control problem.

The safety risk and the safety-related functionality shifts from the un-integrated first state to the integrated new state of the system or system-of-systems[6]. These shifts depend on functionality changes created at the new system level, and the re-distribution of safety decisions and authority. Integrating systems may create new hazards[7] (emergent hazards) or just change existing hazards. SoS integration creates new associations of sometimes both existing and new causal factors, and/or the severity of the hazards. The occurrence of new hazards is usually the result of new functionality or changes to existing functionality in the system-of-systems context, whether intended or inadvertent.

3.3.11.2 SoS Hazard Analysis Techniques

The hazard analyses of SoS is a collection of current techniques that are analyzed together for interactions. There are very few well-defined techniques for identifying hazards and causal factors associated with integrating systems into a system of systems: Interface Hazards Analyses is probably the most commonly used but does not cover physical, architectural, messaging, or nodal faults. It is possible to "pass" interface control requirements and yet fail systems integration due to timing or sequencing faults. Other useful techniques usually consist of using the architecture to walk through a planned operational thread to and from each of the interfacing systems and determine how the new functionality added by the integrated system affects the current hazards. The next step is to use the CONOPS for each interfacing system and for the SoS. Identifying causal factors associated with the system-of-systems proceeds much as it did in the analysis of the individual weapon systems however; the complexity of and number of stakeholders in the new System-of-Systems increases the complexity of the interactions between the systems, multiplies the number of potential causal factors, and multiples the mitigation parameters.

3.3.11.3 System Interoperability

The ability of systems to function together or system interoperability is a significant source of hazard causal factors when integrating systems. These interoperability hazards include: systems not performing as expected but still on-line (degraded, failed, maintenance mode), systems closer together than expected (exhaust from one system burns another, pointing and shooting into another system), loss of awareness. Software safety needs to assess safety from various states and conditions that are best derived from CONOPS and review of current operations using a subject matter expert.

One significant aspect is the compatibility of the data across the interfaces. Compatibility refers to a variety of factors including data rate, data format, data type, coordinate system and associated reference system used, date and time format, references, units of measure, etc. It also

[6] A System-of-Systems is an integration of heterogeneous systems by means of a loosely coupled network that provides some level of control over the systems. This integration can occur at all levels of complexity.

[7] Hazards are generally a function of the systems at the lowest level. Creating new hazards generally requires new functionality involving the control of new energetic components in the system or new applications of existing energetic components, such as using a weapon in a new manner (e.g., providing over-the-horizon fire support).

includes whether the data means the same thing to the interfacing systems[8]. Another data compatibility issue is the ability of a system to provide the data required by other systems. Often, the information provided by the sending system is very different from that the receiving system expects to receive since its original application served different needs. The accurate conversion of that data is critical to ensuring the safety of the operations. This is one safety-critical aspect of systems interoperability and affects the safety of integrating systems into systems-of-systems. In our safety analysis, we will focus on those modules providing the conversion of the information and the interface of those modules to the rest of the fire control system software. Part of the conversion should be a sanity check: a check to verify that the information provided is meaningful in the context of the capabilities of the receiving system and the operations in progress. The latter may not be practical due to the limitations on the information available to the fire control system.

When analyzing the integration of systems, one of the most critical factors is analyzing the interfaces between the systems, especially if the information passed across the interface includes control data or safety-critical data. Very simple differences between the designs of the interfaces often go unnoticed, such as whether the serial data interface is least significant or most-significant bit first. The analysis of the interfaces is critical to assessing the potential risk associated with integrating systems into systems-of-systems.

3.3.11.4 Risk Escalation: Other Factors

Integrating systems into more complex systems is not the only means of escalating the associated safety risk. Using systems outside their intended application can also result in the escalation of risk associated with weapon systems. An example of this is the trend toward using joint warfare information systems, normally used for battle coordination, for directing fire support. In their original inception, these systems were to provide commanders with improved situational awareness from multiple vantage points. These systems provided information regarding coalition and enemy troops and installations that the commanders could use to plan attacks, assess battle damage, etc. In that context, their application was safety related but not safety critical. However, the military soon began using these systems to request fire support, direct fire from artillery batteries, etc. The integrity of the data passed over the information system is essential to ensuring the safety of these operations (e.g., weapons firing). With the trend toward coalition operations growing, the suppliers took the next logical step and integrated the joint warfare information system with the fire control systems, including those aboard ships. In that new system context, the joint warfare information system becomes at least safety-significant: it directly passes target designation data to the fire control systems. If the implementation in the fire control system permits automatic processing of engagement orders, that functionality in the joint warfare system is now safety critical. Errors in the information could result in the laying of fires at the wrong location, possibly killing and injuring the personnel the fire support was to protect.

The above example addresses a relatively complex system evolution. However, applying much simpler systems outside their intended applications, including those that appear to have little safety risk, can significantly increase the safety risk. An example is the use of a Global

[8] For example, target elevation to one system may mean the altitude of the target above mean sea level and, to another system, the elevation angle to the target.

Positioning System device. A coalition company was using the device to pin point the location of an enemy convoy. The device up linked the enemy position data to an attack aircraft that downloaded the data into a precision-guided weapon. During the operation, the GPS device experienced a momentary power interruption. When it came on-line, it transmitted its own location vice the location of the enemy convoy resulting in the precision-guided weapon striking the coalition company. The intended use of the GPS device was as a locator beacon for search and rescue operations.

Many factors affect the integration of these systems that increase the risk associated with the new system-of-systems. These include personnel factors, changes in the warfare scenario, and technological factors. Earlier generations of personnel had an inherent distrust of computer systems: they relied on them to provide information but made decisions based on what their other senses, including intuition, told them. They tended to double-check their information before committing to a course of action recommended by the computer system. Each succeeding generation of users however, tends to put more faith in the information provided by the computer, even to the extent that now, many users ignore information their senses provide them and believe the data provided by the computer. They are also less likely to double-check the information before committing to a recommended course of action.

Changes in the warfare scenario also increase the risk. Modern weapon systems, even those available to third world countries, are faster and stealthier. This makes their detection more difficult and shortens the allowable reaction time from detection to engagement. The quicker reaction time necessary to engage these threats requires the speed of computers and allows little time for the user to interpret the data and second guess the computer. Modern information systems are also faster and more complex and their interactions with other systems are far more complex. All of these factors influence the risk associated with the new systems-of-systems.

Within our engineering design space, we have limited options for reducing the risk associated with this kind of system integration. We must examine the functionality of the integrated systems in the context of the mishaps and hazards identified within the weapon systems. The weapon system has the energies we are concerned about and the combat system of which our weapon system is an element, directly or indirectly exercises control over some of those energies.

3.4 Software Safety Risk Assessment

A great deal of the confidence placed on a critical software system is based upon the results of analyses and V&V performed on specific artifacts produced during system development (e.g., source code modules, executable programs produced from the source code). These results contribute to confidence in the deployed system only to the extent that we can be sure that the tested and analyzed components, and only them, are actually in the deployed system.

3.4.1 Software Mishap Definitions

The difficulty of assigning useful probabilities to faults or errors within software requires an alternate method of determining both the initial and the residual risk for software hazard causal factors. This assessment methodology allows one to assign a qualitative measure of the software risk as opposed to the quantitative assessment traditional to that of hardware components. However, to be useful in the risk assessment of the host system, the methodology must provide a means of equating the risk associated with the software to the quantitative risk assessment

associated with hardware. The initial risk assessment for software functions inherits the MRI/HRI from system level as determined by the PHA or FHA and following the requirements or functional allocation.

The initial risk assessment for software hazard causal factors uses the influence or control that the software has over the hazard or mishap and the severity of the outcome of the hazard or mishap. Table 3-3 provides definitions for the Software Control Categories. This risk assessment is not a traditional safety risk assessment, in that it is not a metric for reporting safety risk. Rather, it is an assessment of the programmatic risk associated with the architecture and design of safety-related functions in the software. It provides a means of identifying the level of effort and level of rigor required to verify that the software functionality does not pose an unwarranted safety risk at the system level.

Software Control Category Definitions	
I	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 mishap.
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 mishap.
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 that relates to safety or is used to make safety decisions. There are several, redundant, independent safety measures for each hazardous event.
IV	Software does not control safety related hardware systems, sub-systems or components and does not provide safety related information.

Table 3-3: Software Control Categories

3.4.2 Software Risk Assessment

The safety team performs risk assessments throughout all phases of the design, development, and testing. These risk assessments provide a means of tracking the progress of the system development effort with the system safety program. The continuous risk assessment of software-related hazard causal factors provides a means of tracking the progress of the software design, development, and testing efforts as well as the software systems safety program. However, the actual assessment metric differs due to the inability to assign useful quantitative probabilities to software failures and especially to failures in specific functional threads through the software.

3.4.2.1 Software Safety Criticality Index (SSCI) Matrix

The SSCI presented in Table 3-4 uses the same definitions for hazard severity as the traditional hazard or mishap risk matrix described in AOP-15. However, the SSCI matrix is different from the traditional hazard risk matrix as it is a tool to assess the risk associated with software control of or influence on hazards. Each program should tailor this matrix to the specific requirements of the program in the beginning of the acquisition life cycle preferably in a system safety management plan or program plan. The safety team should develop a matrix for the system under development that best reflects the needs of the system, the system of systems into which it will be integrated, and regulations that apply to the safety assessment of the system.

The SSCI is not the same as the mishap or hazard risk index used for the overall system, though they appear similar. A low index number from the SSCI Matrix does not mean that a design is unacceptable. Rather, it indicates that a more significant level of effort is necessary for the requirements definition, design, analyses, and V&V of software and its interaction with the system.

The Mishap Risk Index (MRI) and the Hazard Risk Index (HRI) are safety designations at the system level (integrated hardware/software). For communicating risk to the software engineer, the SSCI Matrix summarizes safety links to functional software and designates a level of rigor required by the software and test development group.

Hazard Severity / Software Control Category	Catastrophic	Critical	Marginal	Negligible
I Autonomous	1	1	2	3
IIa/IIb Semi-Autonomous	1	2	3	4
IIIa/IIIb Redundant Backup	2	3	4	5
IV Not Safety Related	3	4	5	5

Table 3-4: Software Safety Criticality Index Matrix

3.4.2.2 Software Safety Criticality Interpretation

The interpretation of the SSCI determines the level of rigor in the analysis and testing that the safety and the software development teams need to apply to the software as well as the level of scrutiny necessary during requirements definition and the design, implementation and validation of safety-related requirements. If a system development program does not implement the recommended safety level of rigor the level of uncertainty rises and so does the level of risk.

PHASE / SSCI	DESIGN	CODE	UNIT TEST	INTEGRATING UNIT TEST	SYSTEM INTEGRATION
1 High Risk	• Design Team Review • Safety Review • SCF Linked to SW Rqmts • SCF Linked to Design Architecture • Safety Fault Tolerant Design	• Design Code Walkthrough • Independent Code Review • Safety Code Analysis • SCF Code Review • Safety Fault Detection, Fault Tolerance	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • Safety Test Result Review	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • 100% Regression Testing • Safety Test Result Review	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • 100% Regression Testing • Safety Test Result Review
2 Serious Risk	• Design Team Review • Prioritizing Safety Review • SCF Linked to SW Rqmts • SCF Linked to Design Architecture	• Design Code Walkthrough • Safety Code Analysis for Prioritized Modules • SCF Code Review • Safety Fault Detection, Fault Tolerance	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • 100% Thread Testing • Safety Test Result Review	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • 100% Regression Testing • Safety Test Result Review	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • 100% Regression Testing • Safety Test Result Review
3 Moderate Risk	• Design Team Review • Minimal Safety Review • SCF Linked to SW Rqmts • SCF Linked to Design Architecture	• SCF Code Review • Safety Fault Detection, Fault Tolerance	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • Safety Test Result Review	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • Safety Test Result Review	• Test Case Review • Independent Test Review • Failure Mode Effect Testing • Safety Test Result Review
4 Low Risk	• Design Team Review • Minimal Safety Review • Normal Software Design Process IAW SDP	No specific tasks	• Test Case Review • Independent Test Review • Safety Test Result Review	• Test Case Review • Independent Test Review • Safety Test Result Review	• Test Case Review • Independent Test Review • Safety Test Result Review
5 No Safety Risk	• Normal Software Design Activity IAW the Software Development Plan	• Normal Software Code Activity IAW the Software Development Plan	• Normal Software Unit Test Activity IAW the Software Development Plan	• Normal Software Unit Integration Test Activity IAW the Software Development Plan	• Normal Software System Integration Test Activity IAW the Software Development Plan

Table 3-5: Level of Rigor Matrix

Using the indices from Table 3-4, Table 3-5 provides recommendations for the level of rigor required for the safety assessment processes and the I&V that the safety team should apply.

3.4.2.2.1 System-Level Hazards Analysis

All safety efforts begin with a high-level assessment of the mishaps and hazards associated with a system, generally referred to as a Preliminary Hazards List. This process requires identification of safety-related events and safety-related functions associated with the system. In themselves, these functions and events may not constitute a hazard or mishap however, should they occur when not desired (e.g., inadvertently), not occur when required (e.g., failure of a fire sensor), occur out of sequence, etc., they likely contribute to a mishap (event) or hazard (function). At a high level, the events and functions are very general in nature and, until the development defines the system architecture, the contribution of the software is likely unknown. However, without this system-level assessment, further identification of mishaps and hazards is more difficult.

3.4.2.2.2 Architectural Assessment of System and Software

The assessment of the system architecture uses the Preliminary Hazards List as a baseline to identify the safety-related events, sequences, and functions with associated hazards that require mitigation. At the system architecture level, the safety team's goals are to ensure that the architecture:

- Supports desirable safety attributes,
- Provides timely (does not block or delay) knowledge or control of safe or hazardous states or modes;
- Provides survivable access to knowledge or control of safety mitigations or operators;
- Does not contain constructions that make the overall safety assessment difficult or impossible, and
- To influence the allocation of safety-related functions between hardware, software, operator, and maintainer in a manner that ensures the system will have a final residual risk that is as low as reasonably practical in the operational environment.

There are a large number of architectural patterns that designers can use to develop a system. A large portion of these patterns is inherently unsafe because the resultant system does not permit the effective application of the overall safety assessment process. High interaction (coupling in software engineering terms) between safety-related and non-safety related functions, non-modularization of mission-related and safety-related functionality, lack of visibility into the design to allow detailed testing, (PDS), etc. are a few of the many attributes an architecture can create that make it unsuitable for safety assessment. Additional information on generic software safety guidelines is contained in Chapter 4 [Generic Software Safety Design Requirements].

The allocation of events to hardware, software, the operator, and the maintainer occurs largely during the design of the system architecture. The development team will generally evaluate several allocations, determine the benefits and drawbacks of each, and make a final recommendation based on these and other factors. Inputs and recommendations from the safety team can affect the inherent risks of particular system architectures.

3.4.2.2.3 Requirements Hazards Analysis

The Requirements Hazards Analysis provides a means of analyzing all requirements to determine or designate safety-related requirements, derived safety-design requirements, and generic safety-design requirements throughout the software development. It begins with the identification of safety-related events and the associated functions at the system level, such as those found in a Functional Hazard Analysis (FHA). The analysis will also incorporate the tailored generic safety-design requirements, correlating them to specific safety-related functionality in the system design as practical.

3.4.2.2.4 Code Level Analysis

Code-level analysis, often referred to as "static testing", may address one of two types of code: the source code (programming language) and/or the object (or machine) code (binary code that executes on the processor). The level of detail in the analysis can range from simple inspection to formal proofs of correctness of the source or object code. The level and detail of the analysis depends on the requirements for the particular safety-related function, the nature of the analysis, the reason for the analysis, and the focus of the analysis. A shallow analysis (e.g., inspection) may address only the program layout, ensuring that it complies with the next higher-level architecture and follows the recommended coding constraints. A "medium-depth" analysis may look at code structure, the implementation of safety-related functionality, and may address specific properties of the code (e.g., memory management) or specific functionality related to other aspects of the code such as exception or interrupt handling. A deep analysis may analyze the object code and its execution on the processor in a manner that allows the analyst to trace the functionality in the processor, its registers and cache memory, and the address and data busses.

3.4.2.3 Testing

Testing provides the evidence necessary to demonstrate that system-level, the software, the generic and derived safety-design requirements and their implementation provide the desired risk reduction. However, to provide sufficient argument that the software does not pose an unacceptable or undesirable risk, the test program must include sufficient safety-specific testing to verify the design and implementation of the software mitigates all of the identified software-related hazard causal factors. Chapter 4 [Generic Software Safety Design Requirements] includes recommendations for "generic" tests for safety-related software. Generic tests are mainly structural test, relating to the selected hardware, the selected compiler and software language, and the selected architecture(s). However, generic tests alone cannot provide the necessary evidence that the software can execute safely in the system context. That is due to the subtle and complex interactions between the systems-of-systems, the operator/maintainer, the software and hardware, within the software, and in the software control of safety-related hardware.

Analysts will identify software-related hazard causal factors during the analysis phases and develop safety-design requirements and/ or recommendations to mitigate either the hazard causal factor or the effects of the hazard causal factor. Normally, requirements-based testing will include safety-related requirements and specify test cases to verify the implementation of these requirements. The safety team must review both the test plan and the test procedures to ensure that the test cases provide the required evidence of risk mitigation validation. This testing should include fault insertion and failure mode testing that verifies the correct response of the software

to these anomalies. As the criticality of the software increases, the depth of analysis increases (e.g., detailed design, code-level analysis) and the level of necessary testing increase. Analysts should use the analysis of the implementation to develop detailed tests of the safety-critical software to ensure that it achieves its objectives.

3.4.2.3.1 Software Independent Verification and Validation

Software Independent Verification and Validation, or IV&V, is a requirements-based test program run by a group independent of the organization that developed the software. The testing first verifies that the computer correctly and completely implements the software-related requirements, from the highest level of abstraction to the lowest level of software requirements. The testing also validates that the requirements implemented in the software are those necessary to achieve the system-level (or mission-level) requirements. The IV&V organization will identify functionality that does not appear to relate to the high-level requirements associated with the system. It verifies (to the extent practical) that there is no functionality that does not have a requirement at a higher level of abstraction.

The effectiveness of the IV&V effort requires that the team understand not only the requirements but also the intent and objectives of the requirements. Therefore, the IV&V team should work closely with the software development team to fully understand the functionality and its implementation in the final product. IV&V interaction and test may occur at various stages of development, depending on the nature of the development, its complexity, and the ability to break the system software into segments that lend themselves to this form of testing.

Safety testing must be an integral part of all IV&V processes. The safety team and the IV&V team should work together to ensure that the IV&V process includes adequate testing of safety-related functionality, verifies the correct implementation of safety-design requirements, and provides another level of evidence that the safety analyses achieved the desired risk mitigation. For the IV&V team to verify the correct implementation of safety-design requirements, the safety team must provide them with specific guidance on the safety-design requirement, its intent, and both the desired and undesired outcomes of the testing. Likewise, the safety team should also review the IV&V tests and test cases to ensure they meet these objectives. Furthermore, the safety team should provide the IV&V team with guidance on the specific concerns associated with safety-related functionality in the system. These tasks will ensure that the IV&V team can provide the data required by the program and secondarily that safety team use the IV&V output to confirm the system-level risk mitigation achieved.

3.4.2.3.2 Validated Development Tools

Validation of development tools (tools beyond the software engineering tools) requires that the development team, in concert with the system safety team, ensure that all tools used in the design, development, analysis, and testing of the software product receive an appropriate level of assessment to the product under development and their purpose in the overall development effort. Suppliers frequently use models, simulators, stimulators, and emulators as part of the development effort, particularly for testing purposes. Validation of these tools requires that the development team ensure that they provide as realistic a simulation as possible within the bounds of their intended purpose. For example, a simulated sensor should provide inputs (including failure modes, degraded operation, etc.) as close to the actual simulator as practical. Timing and

state or mode transitions are the most difficult to validate. In early testing, the ability to test failure modes or degraded operation may not be as important as it is in later test efforts. Therefore, the level of fidelity required of the simulator varies with each development phase and must be validated for each phase or use.

3.4.3 Residual Risk Assessment

Accomplishing recommended tasks as shown in Table 3-5 is not sufficient to demonstrate that the risk is as low as reasonably practical, although accomplishing these tasks will provide substantial risk reduction. The safety team must perform the "engineering" on the output safety data and accumulation of results required to ensure that the safety requirements derived has provided the desired risk reduction, that the implementation of the requirements achieves their intent, and that testing of the final product provides the evidence that it achieves the desired risk reduction. They must also be an active participant in the overall software engineering and system engineering processes.

3.5 Safety Assessment Report/Safety Case

A Safety Assessment Report (SAR) is a structured argument, supported by a body of evidence, that provides a compelling, comprehensive and validated assessment that a system is safe for a given application in a given environment. That body of evidence is the Safety Case[9], which contains all necessary information to enable the safety[10] of the system to be assessed. It will contain an accumulation of both product and process documentation. The Safety Case will evolve over the life of the system and will be structured such that safety arguments are developed. While the structure of the Safety Case will broadly remain constant, the status of the evidence will change e.g. planned test coverage will be replaced by evidence of test results.

3.5.1 Safety Case Overview

A Safety Case is required for all systems and may be produced at the sub-system, system or system-of-systems level. Where a system includes sub-systems that have separate Safety Cases, these Safety Cases should be integrated and reconciled within the higher-level system Safety Case. This will assist in demonstrating that interface and other safety issues have been managed effectively, and those assumptions and cascaded safety requirements have been properly addressed.

3.5.2 Safety Case Summary

Safety Case content will be dependent on the nature of the system and the potential risk associated with it but as a minimum the Safety Case should be sufficient to demonstrate that any activities underway at that time (including tests or trials) can be carried out safely. The progress of a project at particular stages of its life may be dependent on acceptance of a safety assessment report that summarizes the arguments and evidence.

3.5.3 Safety Case Contents

The Safety Case should typically provide evidence at least that:

---

[9] Some safety review authorities refer to the Safety Case as a Technical Data Package, Technical Munitions Safety Study, SAR or similar title

[10] Also referred to as the residual risk

- All applicable legislation, regulations, standards and policy have been complied with.
- The Safety Management System is effective and is compliant with the Safety Management Plan.
- The staff undertaking key roles with defined responsibilities had the appropriate competencies for those roles.
- The set of safety requirements is valid and traceable. The safety requirements should have been derived by thorough analysis and should correspond to the system as designed and implemented. Many safety requirements are derived from actions to reduce risk that are identified as part of the Hazard Analysis. Because of this, the evidence should show that Hazard Analysis and risk management processes have been carried out in accordance AOP 15.
- Arguments and evidence are provided to demonstrate that all safety requirements, including contractual requirements and relevant process and procedural safety requirements, have been satisfied, or there is adequate mitigation for any non- or partial-compliance with the safety requirements.
- Any residual system failures are within the tolerable limits determined by the risk management process and are managed safely.
- Safety claims, safety arguments and evidence have been subjected to independent scrutiny.
- The quantity, quality and rigor of evidence is commensurate with the arguments it supports

The Safety Case is a comprehensive set of artifacts and in general, it is not practicable or necessary for it to be physically delivered, at least in paper form.

3.5.4 Safety Assessment Report

The purpose of the SAR is to provide management an overall assessment of the risk associated with the system including the software executing within the system context of an operational environment. This safety assessment report provides a snapshot summary of the Safety Case at key milestones. In addition, the report will provide details of the progress made in managing safety since the previous report and should be structured around the safety claims for the system and the planned activities. The safety assessment report should provide justifiable confidence that the Safety Case is, or will be, comprehensive and that the expected progress is being made on planned activities. This safety assessment report must be an encapsulation of all of the analysis performed as a result of the recommendations provided in the previous sections.

3.5.5 Safety Assessment Report Contents

The contents of the safety assessment report will vary according to the maturity of the Safety Case and the intended readership. It has two main functions. Firstly, to assure the reader that safety risks are being managed effectively and so should include a clear and concise summary of the Safety Case and safety progress. Secondly, to highlight key areas of risk to the operators and users and so should provide information that will support operational decision-making. The safety assessment report should contain meaningful information and be as concise as possible, without sacrificing the need to provide the necessary information. References should be provided to supporting material within the Safety Case. A suggested structure is as follows:

- Executive Summary.

  - The executive summary should provide assurance to stakeholders that safety requirements have been, or will be, met by:
    - Confirming that Safety Case work has been, or is being, progressed satisfactorily.
    - Confirming that all other stakeholders have formally acknowledged their safety responsibilities.
    - Recommending or otherwise progression to the next stage of the acquisition cycle or the next defined milestone confirming that safety risks associated with the next stage can be satisfactorily managed.
- Summary of System Description.
  - A brief description of the system should be provided, noting that a full System Description will be contained within the Safety Case. The summary should be sufficient to enable the boundaries and scope of the Safety Case and its interfaces with other Safety Cases to be clearly defined and understood.
- Assumptions.
  - Assumptions that underpin the scope of the Safety Case, or the safety requirements, argument or evidence should be stated. For example, this may include numbers of personnel, training levels, operational profiles, time in service, operating environment, etc.
- Progress against the Program.
  - An assessment of progress against the safety programme should be provided that describes:
    - An indication of the current status relative to the expectations documented within the programme, including an assessment of any impacts on future progress.
    - Progress on safety management since the previous safety assessment report, including identification of any new hazards and accidents and progress on risk management activities.
    - Progress against agreed actions placed on stakeholders.
- Meeting safety requirements: The following should be included:
  - A statement describing the principal, agreed safety requirements.
  - A summary of the argument and evidence that demonstrates how the safety requirements have been, or will be, met. This will describe:
    - Summary of the hazards and likely accidents associated with the system, noting the main areas of risk. Note: The aggregation of risk should be documented and main areas of risk will also be highlighted under the Operational Information heading.
    - Safety requirements that are unlikely to be met, either in part or in full, with remedial/follow-up actions identified.
    - Risk management actions that are outstanding, identifying both the risk and the organisation responsible for its management.
    - The residual risk that is, or is anticipated to be, posed by the System.

- The safety criteria and methodology used to classify and rank software related hazards (causal factors). This includes any assumptions made from which the criteria and methodologies were derived;
- A discussion of the engineering decisions made that affect the residual risk at a system level.
- Issues of particular sensitivity, e.g. use of restricted materials, or with significant project or corporate risk.
- Regulatory approvals/certificates, and any associated restrictions that are currently in place.
- Any counter-evidence found that may invalidate the Safety Case, including a description of the activities taken to address this counter-evidence.
- Feedback, reporting and auditing arrangements for defects and shortfalls.
- Particular issues related to interfaces between different systems.

- Emergency/Contingency Arrangements.
  - A statement confirming that appropriate Emergency/Contingency Arrangements (e.g. procedures) have been or will be put in place and identification of any areas where such arrangements do not exist or are inadequate.
- Operational Information. (This section will be aimed specifically at the operator).
  - Outputs from the Safety Case that are relevant to the management of operational safety, including:
  - A description of the operational envelops.
  - Any limitations on use or operational capability.
  - The main areas of risk (i.e. high risk areas).
  - Relevant information that can assist the operator in balancing the operational imperative against safety risk.
  - Demonstration that operating and maintenance procedures and publications have been, or will be, developed.
  - Report on any independent safety assessment.

- Conclusions and Recommendations.
  - Conclusions should be provided, including an overall assessment of the safety of the system and any recommendations to enable issues identified within the report to be resolved.

- References.
  - A list of key reference documents should be provided (e.g. parts of the Safety Case documentation)

3.5.6 Overall Risk

The final section of the SAR should include a statement describing the software contribution to the overall residual risk. Failure to conform to the guidelines of this AOP may result in unacceptable risk. The acceptance of that risk by an appropriate authority is mandatory.

3.5.7 System Development

During system development, safety assessment reports show the progress in risk reduction and in producing safety evidence. A major phase in the safety assessment reporting cycle will be at the completion of the development phase, prior to acceptance of a system into service; at this stage, the safety assessment report is likely comprise a number of versions, presented to varying degrees of granularity according to the expected reader (e.g. at a high level for top management down to very low level for detailed analysis). In operation, safety assessment reports support the operational use of the system, and present data on the rate of occurrence of safety-relevant events and the remedial action, if any, needed to preserve safety.

Note: The maintenance of the Safety Case for a system continues until disposal of the system.

3.6 Complex Electronic and Programmable Systems

The previous section defines the requirement for a Safety Case and specifies the associated requirement to produce periodic safety assessment reports demonstrating the status of the Safety Case. This Section is concerned with demonstrating safety and in the context of complex electronic systems ("programmable" systems).

In defense systems, complex electronic elements are an integral part of the overall system and the safety requirements should be driven top down from the overall system requirements. Some complex electronic elements will have a greater potential impact on safety than others. This section indicates the types of evidence that may be used for assurance, but it should be noted that the question of sufficiency of evidence would generally involve expert judgment. The role of the safety related complex electronic element, its criticality and the application domain of the system all affect judgments of tolerability and hence of the sufficiency of evidence necessary for assurance.

AOP 15 defines the requirements for risk management and Section 3.4 discusses analysis, conducted as part of the risk management process, that specifically addresses complex electronic elements. Within the Safety Case, safety requirements, both specified and derived, must be clearly demonstrated to be:

- Valid and consistent with the ALARP principle.
- Safety Integrity requirements are valid and achievable

In addition, evidence should be provided that the following safety claims have been met:

- The functionality of the software is safe.
- Software failures are safe.
- Safety can be maintained over the lifetime of the software.
- The evidence to support the safety argument is commensurate with the safety integrity of the software.

3.6.1 Provision of Evidence

The body of evidence required to demonstrate that a complex electronic element is adequately safe is likely to be extensive and this section expands upon the types of evidence required, as part of the system Safety Case, based on meeting the above safety claims.

Direct evidence, process evidence and counter-evidence relating to the complex electronic element should be documented and analyzed. Direct evidence relates directly to the properties of the complex electronic element, to its implementation or its behavior in an operational or simulated environment. Direct evidence may also relate to the requirements, role or mitigation associated with the use of the complex electronic element in its system context. Process evidence is evidence about the processes for risk assessment, procurement, development, verification, validation (including demonstration) or operation of the complex electronic element. Such process evidence serves to increase confidence in the direct evidence.

The quality and quantity of evidence provided for assurance of complex electronic elements should be proportionate to the safety integrity requirements.

The evidence should be selected so that it supports claims for the safety of the complex electronic elements. Evidence should be traceable via arguments to the safety claims that it supports and to the derived safety requirements and safety requirements (see Section 3.4). Safety claims should typically address the correctness and sufficiency of the safety requirements (including safety integrity requirements) and the satisfaction of the safety requirements (including that the failure rate satisfies the safety integrity requirements).

3.6.2 Direct Evidence

This section provides guidance for complex electronic elements on the following potential types of direct evidence:

- Analysis evidence
- Demonstration evidence
- Quantitative evidence
- Review evidence
- Qualitative evidence

3.6.3 Analysis Evidence

Evidence from analysis may be used to demonstrate absence of dangerous faults and achievement of derived safety requirements in the complex electronic element. Analysis evidence may also be used to derive the safety requirements and to provide evidence of the types of failure mode that are possible (or prevented from occurring). If analysis evidence forms part of the Safety Case, the following recommendations apply.

- Reasoned justification should be provided for the context and limitations of the evidence generated by analysis.
- Analyses should be fully documented and work products should be held under configuration control, so that the analyses are repeatable, auditable and verifiable.

- Analyses should be rigorous and automated where possible. Justification should be provided for the competence of the people performing the analyses and for the processes and tools used.
- The analysis may be used to provide evidence that the safety requirements are satisfied and the derived safety requirements of the complex electronic element hold. Such analyses may include timing (e.g. worst case execution times), use of resources, computational accuracy, possibility of run-time error and functional properties. Analysis evidence has the potential to show the absence of all known classes of particular types of fault (e.g. absence of run-time error).
- The analysis evidence may also define the safety requirements for the complex electronic element via modeling of the system context and external mitigation.

3.6.4 Demonstration Evidence

Operational experience and verification and validation evidence may be used to demonstrate that the behavior of the complex electronic element is safe. If demonstration evidence forms part of the Safety Case, the following recommendations apply:

- Verification and validation of dynamic behavior should be fully documented and work products should be held under configuration control, so that test cases are repeatable, auditable and verifiable.
- Operational experience should be documented and auditable.
- The extent and coverage of dynamic behavior through verification and validation or operational experience should be justified.
- The differences between any test environments and the operational environment should be documented and evidence provided to show that the test environment and test cases provide a valid demonstration of operational behavior

This demonstration evidence may be from testing or from exercising the complex electronic element in an operational context and the detailed evidence may be further analyzed to form quantitative evidence. Requirements based testing generally produces evidence that is easier to link to safety claims.

3.6.5 Quantitative Evidence

Quantitative evidence should be used to show how the complex electronic element performs against its quantitative safety requirements. As quantitative requirements should be specified for complex electronic elements, quantitative evidence should form part of the Safety Case. Quantitative evidence usually relies on statistical models, and the appropriateness of the model used should be demonstrated.

3.6.6 Review Evidence

Review evidence may be used to show that the complex electronic element is capable of satisfying its safety requirements. If review evidence forms part of the Safety Case, it is recommended that the reviews should cover:

- Traceability to ensure that safety requirements are translated into the derived safety requirements, and hence into the implementation.

- Maintainability, where required as part of the safety requirements, to ensure that the complex electronic element is designed in a way that facilitates future modification or correction.
- Compliance to ensure that design and implementation practice conforms with specified standards of good practice.
- Validity to ensure that the complex electronic element implements the safety requirements and does so correctly (verification).
- Robustness to ensure that faults in the complex electronic element, as well as failures originating in other system elements, is managed safely.

3.6.7 Qualitative Evidence of Good Design

Qualitative evidence may be used to show that good practice has been used in the selection of derived safety requirements, architecture and design features of the complex electronic element. If qualitative evidence of good design forms part of the Safety Case, the following recommendations apply:

- Qualitative evidence of good design should be provided for all necessary safety features and derived safety requirements of the complex electronic elements.
- The evidence should include the rationale, benefits and limitations of the design.
- Evidence of good design should include references to significant examples or case studies illustrating successful use, where available.
- Evidence should be provided that this design is appropriate given the system context and the functionality and safety related role of the complex electronic element.

3.6.8 Process Evidence

Process evidence should support the direct evidence. For all systems process evidence should encompass all assurance and risk assessment and risk mitigation processes including Hazard Analysis, system selection, integration, commissioning and modification processes.

3.6.8.1 Process and Tool Qualification

Evidence should be provided that the tools and processes used have sufficient safety assurance to ensure that they do not undermine the integrity of the complex electronic element.

The competence requirements for personnel undertaking each process should be stated and evidence should be retained that the personnel performing the processes have the required competence.

Each tool and process should be evaluated to determine its role and significance to safety. The following list is specific to safety issues, however this is not an exhaustive list of factors relevant to tool selection (others may include usability, interoperability, stability, commercial availability, maintenance support, familiarity to safety personnel):

- The role of the tool or process in assuring the safety of the complex electronic element.
- Whether the tool or process could introduce a safety significant fault.
- Whether the tool or process could fail to detect a safety significant fault.
- How failures of the tool or process could be detected and corrected by human supervision and by other tools and processes

Further details on the use of tools may be found in sections 4.8 and 4.9 of Appendix D.

3.6.9 Good Development Practice

The Supplier should provide evidence that the processes used in the risk assessment, procurement, development, implementation, verification, validation, modification and correction of complex electronic elements comprise good practice.

The evidence of good practice should be appropriate to the application, domain and safety requirements. Examples of good practice in process include:

- Evidence of compliance with appropriate, respected standards – preferably standards that are international, relevant to domain and mature but still considered good practice.
- Evidence of selection of good practice methods, tools, technology etc (this will typically be at a more technical level of detail than is covered by an international standard and may include specific software language, hardware technology, development toolsets etc).
- Evidence of good practice in applying methods, technology, tools etc (e.g. internal procedures, processes and standard, tool supplier's recommended best practice, user group recommendations etc).

3.6.10 Sufficiency and Composition of Evidence

The body of evidence, taken as a whole, should be sufficient to provide confidence, commensurate with the required safety integrity requirements, in the safety of the complex electronic element as part of the system Safety Case.

The primary arguments for the safety of the complex electronic element should be based upon direct evidence, which may include analysis evidence, demonstration evidence, review evidence and quantitative evidence. Process evidence should be used to support the primary arguments (and may additionally support claims for future maintainability if this is a safety concern within the Safety Case). Qualitative evidence for good design should support the other forms of direct evidence.

3.6.11 Strength and Rigor

The rigor of the evidence and arguments should be proportionate with the required level of confidence. The type of evidence provided for primary arguments should be based on the precedence below (preferred first). The primary safety arguments should be based on the strongest types of evidence and then supported by other types of evidence.

- Analysis evidence for the absence of dangerous faults, the satisfaction of safety requirements and the implementation of derived safety requirements.
- Quantitative analysis of operational or realistic demonstration of the required behavior that shows that availability and reliability requirements are satisfied, to a level of confidence commensurate with the safety integrity requirements of the system.
- Demonstration evidence and review evidence.
- Qualitative evidence of good design, and process evidence

3.6.12 Coverage

The evidence provided should be representative of all aspects of the argument that it supports and sufficiently extensive to provide the required level of confidence. This requires that:

- All safety requirements and derived safety requirements should be covered by the evidence.
- Specific safety requirements and derived safety requirements should direct the selection of evidence. For example, when using formal proof, proofs should be constructed and discharged to prove that specific derived safety requirements are true.
- The assumptions, dependencies and limitations of the evidence for all safety claims should be documented.
- Analysis evidence should be supported by diverse demonstration evidence.
- Quantitative evidence should be supported by at least diverse traceability evidence and evidence from review of architecture and implementation quality.

4 Generic Software Safety Design Requirements

The goal of this chapter is to provide generic software safety design guidelines for the design and development of systems containing software that have safety-related applications. Additional discussion of these guidelines is provided in Appendix D.

4.1 System Design Requirements

4.1.1 Two Person Rule

At least two people shall be thoroughly familiar with the design, code, testing, and operation of each software module in the system.

4.1.2 Program Patch Prohibition

Patches shall be prohibited throughout the development process. All software changes shall be coded in the source language and compiled prior to entry into operational or test equipment.

4.1.3 Designed Safe States

The system shall have at least one safe state identified for each logistic and operational phase.

4.1.4 Safe State Return

The software shall return hardware subsystems terms under the control of software to a designed safe state when unsafe conditions are detected. Conditions that can be safely overridden by the battle short shall be identified and analyses performed to verify that the risk is acceptable.

4.1.5 Circumvent Unsafe Conditions

The system design shall not permit detected unsafe conditions to be circumvented. If a "battle short" or "safety arc" condition is required in the system, it shall be designed such that it cannot be activated either inadvertently or without authorization.

4.1.6 External Hardware Failures

The software shall be designed to detect failures in external hardware input or output hardware devices and revert to a safe state upon their occurrence. The design shall consider potential failure modes of the hardware involved.

4.1.7 Safety Kernel Failure

The system shall be designed such that a failure of the safety kernel (when implemented) will be detected and the system returned to a designated safe state.

4.1.8 Fallback and Recovery

The system shall be designed to include fallback and recovery to a designed safe state of reduced system functional capability in the event of a failure of system components.

4.1.9 Computing System Failure

The system that shall be designed such that a failure of any computing system will be detected and the system returned to a safe state.

4.1.10 Maintenance Interlocks

Maintenance interlocks, safety interlocks, safety handles, and/or safety pins shall be provided to preclude hazards to personnel maintaining the computing system and its associated equipment. While overridden, a display should be made on the operator's or test conductor's console of the status of the interlocks, if applicable.

4.1.11 Interlock Restoration

Where interlocks must be overridden or removed to perform tests, training or maintenance, they shall be designed such that they cannot be inadvertently overridden, or left in the overridden state once the system is restored to operational use. The override of the interlocks shall not be controlled by a computing system.

4.1.12 Simulators

If simulated items, simulators, and test sets are required, the system shall be designed such that the identification of the devices is fail safe and that operational hardware cannot be inadvertently identified as a simulated item, simulator or test set.

4.1.13 Logging Safety Errors

Errors in safety-related routines shall be logged and brought to the operator's attention as soon as practical after their occurrence.

4.1.14 Positive Feedback Mechanisms

Software control of critical functions shall have feedback mechanisms that give suitable positive indications of the function's occurrence. These feedback mechanisms must not cause unsafe interference with other functions.

4.1.15 Peak Load Conditions

The system and software shall be designed to ensure that design safety requirements are not violated under peak load conditions.

4.1.16 Ease of Maintenance

The system and its software shall be designed and documented for ease of maintenance.

4.1.17 Endurance Issues

The system and software must be designed, developed, and tested to continuously operate for a specified period of time without safety anomalies occurring. The specified time period should be 1.5 times the specification requirement, if specified, or the maximum expected mission time. Verification should be accomplished in an environment that is representative of the operational environment.

4.1.18 Error Handling

The system and software shall be designed to remain robust and safe in the presence of errors, faults, failures, and exceptions generated by the application, operating system or processor.

4.1.19 Standalone Processors

Where practical, safety-related functions should be performed on a standalone computer. If this is not practical, safety-related functions shall be isolated to the maximum extent practical from non-critical functions.

4.1.20 Input/Output Registers

Input/output registers and ports shall not be used for both safety-related and non-safety-related functions unless the same safety design criteria are applied to the non-safety-related functions.

4.1.21 Power-Up Initialization

The system shall be designed to power-up into a predetermined safe state. A verifiable system monitor check shall be incorporated in the design that verifies that the system is in this safe state. This check shall also verify memory integrity and program load.

4.1.22 Power-Down Transition

The system shall be designed to power-down into a predetermined safe state. A verifiable system monitor check shall be incorporated in the design that verifies that the system is in this safe state.

4.1.23 Power Faults

The system and computing system shall be designed to ensure that the system is in a safe state during power-up faults, power-down faults, intermittent faults or fluctuations in power that could adversely affect the system, or in the event of power loss.

4.1.24 System-Level Check

The software shall be designed to perform a system-level check at power up to verify that the system is safe and functioning properly prior to application of power to safety-related functions including hardware controlled by the software. Periodic software tests shall be performed to monitor the state of the system to insure safe operating conditions.

4.1.25 Redundancy Management

If the supplier's design includes redundancy, any potential failure modes associated with the redundancy scheme shall be identified to ensure that the system requirements adequately mitigate the risk.

4.2 Computing System Environment Requirements and Guidelines

The requirements and guidelines of this section apply to the design and selection of computers, microprocessors, programming languages, and memories for safety-related applications in computing systems.

4.2.1 Hardware and Hardware/Software Interface Requirements

- CPU
- Memory

- Failure in the computing environment
- Hardware and software interfaces
- Self-test Features
- Watchdog timers, periodic memory checks, operational checks
- System utilities
- Compilers, assemblers, translators, and OSs
- Diagnostic and maintenance features
- Memory diagnosis

4.2.2 Failure in the Computing Environment

An application program exists in the context of a computing environment - the software and hardware that collectively support the execution of the program. Failures in this environment can result in a variety of failures or unexpected behavior in the application program and, therefore, must be considered in a hazard analysis. For some of these failure modes (e.g., program overwrite of storage), it is particularly difficult to completely predict the consequences (e.g., because it depends on what region is overwritten and what pattern is written there); the burden of proof is, therefore, on the supplier to provide evidence either that there is no exposure to these kinds of failure or that such failures do not represent a potential hazard.

- Has the supplier identified the situations in which the application can corrupt the underlying computing environment? Examples include the erroneous writing of data to the wrong locations in storage (by writing to the 11^{th} element of a 10 element array, for example, or through pointer manipulation in "C" or unchecked conversion or use of pragma Interface in Ada). Has Ada's pragma Suppress been used? If so, how does the supplier ensure that such storage corruption is not being missed by removing the runtime checks? Note that if pragma Suppress is used and the detection of a constraint violation is masked, the results are unpredictable (the program is "erroneous"). Has the supplier provided evidence that the software's interaction with the hardware does not corrupt the computing environment in a way that introduces a hazard (e.g., setting a program status word to an invalid state, or sending invalid control sequences to a device controller)?
- Has the supplier analyzed potential failure modes of the Ada Runtime Environment (ARTE), the host OS or executive, and any other software components (e.g., Data Base Management System) used in conjunction with the application for any hazards that they could introduce? What evidence does the supplier provide that either there are no failure modes that present a hazard or that the identified hazards have been mitigated [e.g., what evidence does the supplier provide for the required level of confidence in the ARTE, OS, etc.? (e.g., for commercial avionics certification and other safety-related domains, high assurance or even "certified" subset ARTEs have been used)]
- Has the supplier provided evidence that data consistency management has been addressed adequately where it can affect critical functions? For example, is file system integrity checked at startup? Are file system transactions atomic, or is there a mechanism for backing out from corrupted transactions?

4.2.3 CPU Selection

The following guidelines apply to the selection of CPUs:

- CPUs that process entire instructions or data words are preferred to those that multiplex instructions or data (e.g., an 128-bit CPU is preferred to a 64-bit CPU emulating a 128-bit machine)
- CPUs with separate instructions and data memories and busses are preferred to those using a common data/instruction buss. Alternatively, memory protection hardware, either segment or page protection, separating program memory and data memory is acceptable
- CPUs, microprocessors and computers that can be fully represented mathematically are preferred to those that cannot

4.2.4 Minimum Clock Cycles

For CPUs that do not comply with the guidelines above, or those used at the limits of their design criteria (e.g., at or above maximum clock frequency), analyses and measurements must be conducted to determine the minimum number of clock cycles that must occur between functions on the buss to ensure that invalid information is not picked up by the CPU. Analyses must also be performed to ensure that interfacing devices are capable of providing valid data within the required time frame for CPU access.

4.2.5 Read Only Memories

Where Read Only Memories (ROM) are used, positive measures must be taken to ensure that the data cannot be corrupted or destroyed.

4.3 Self-Check Design Requirements and Guidelines

The design requirements of this section provide for self-checking of the programs and computing system execution.

4.3.1 Watchdog Timers

Watchdog timers or similar devices must be provided to ensure that the microprocessor or computer is operating properly. The timer reset must be designed such that the software cannot enter an inner loop and reset the timer as part of that loop sequence. The design of the timer must ensure that failure of the primary CPU clock cannot compromise its function. The timer reset must be designed such that the system is returned to a known safe state and the operator alerted (as applicable).

4.3.2 Memory Checks

Periodic checks of memory, instruction, and data buss(es) must be performed. The design of the test sequence must ensure that single point or likely multiple failures are detected. Checksum of data transfers and Program Load Verification checks must be performed at load time and periodically thereafter to ensure the integrity of safety-related code.

4.3.3 Fault Detection

Fault detection and isolation programs must be written for safety-related subsystems of the computing system. The fault detection program must be designed to detect potential safety-related failures prior to the execution of the related safety-related function. Fault isolation programs must be designed to isolate the fault to the lowest level practical and provide this information to the operator or maintainer.

4.3.4 Operational Checks

Operational checks of testable safety-related system elements must be made immediately prior to performance of a related safety-related operation.

4.4 Safety-Related Events and Safety-Related Functions

A key aspect of the safety assessment of systems and software is the identification of safety-related events and the associated safety-related functions. Safety-related events are those that will cause hazards or mishaps should they occur inadvertently or when not desired (e.g., firing of a projectile is a safety-related event; inadvertent firing of a projectile is a hazard). Safety-related events at the system level always have a direct correlation in the Preliminary Hazards Analysis (PHA): if not, the PHA is incomplete. Safety-related events at a subsystem level will have a correlation in the Subsystem Hazards Analysis (SSHA) or both the PHA and the SSHA[11].

Safety-related functions are those functions that lead to or control safety-related events or generate or manipulate safety-related data. Safety-related functions may include hardware functions, software functions, and/or human actions. Identifying safety-related functions in the software provides a basis for identifying high-level software-related hazard causal factors thus allowing development of high-level safety design requirements to mitigate the identified hazard causal factors. Safety design requirements are often the inverse of the software-related hazard causal factor. Note that at this level of abstraction, the safety requirements are very broad and provide a baseline for further derivation of safety requirements.

During the early design phases, prior to finalizing the system architecture, System Safety Engineers use the identification of safety-related events and the associated functions to influence the architecture of the system with the objective of ensuring that the system architecture, and particularly the software architecture, supports desirable safety attributes. This same process applies at each level of refinement of the architecture including subsystems and the associated software. As the Systems Engineering team defines and refines the architecture, the System Safety team will develop and refine high-level safety requirements for implementation using the previously identified safety-related functions, associated hazards and safety design requirements. Complex systems require more iterations of this process than do simple systems.

[11] A hazard may be uncovered in the safety assessment of a subsystem that did not appear at the system level due to the lack of detail available during earlier analyses or it may not affect the overall system hazards such as a failure of one subsystem resulting in damage to another subsystem.

As the refinement of the system design proceeds, the Software Safety team will be able to refine the identification of software-related hazard causal factors and the development of safety design requirements or implementation recommendations to mitigate them. Tracing the functional thread through the software is one means of identifying potential software-related hazard causal factors at subsequent levels of refinement. However, the Software Safety team must also analyze other aspects of the software that can adversely influence the safety-related functional thread. Functionality within modules that form a part of the safety-related functional thread may impact the safe execution of that thread even though that functionality does not directly affect the functional thread. This may be due to errors in the implementation of the software or non-deterministic actions of the software caused by environmental factors[12]. Likewise, software that interfaces to the safety-related functional thread may cause the function to execute in an unsafe manner. The safety community often identifies software that directly affects the safety-related function as a first-level interface. A first-level interface function includes software whose failure directly results in a hazardous condition (i.e., hazard causal factor). Software interfaces that influence the execution of the safety-related functional thread but do not directly result in the associated software hazard causal factor are designated second-level interfaces. Second-level interfaces frequently include those software interfaces that, should they fail in combination with other failures of unexpected conditions, lead to the undesired hazard causal factor.

4.5 Safety-Related Computing System Functions Protection

The design requirements and guidelines of this section provide for protection of safety-related computing system functions and data.

4.5.1 Safety Degradation

Other interfacing automata and software must design the system such that automata and software must prevent degradation of safety.

4.5.2 Unauthorized Interaction

The software must be designed to prevent unauthorized system or subsystem interaction from initiating or sustaining a safety-related function sequence.

4.5.3 Unauthorized Access

The system design must prevent unauthorized or inadvertent access to or modification of the software (source or assembly) and object code. This includes preventing self-modification of the code.

4.5.4 Safety Kernel ROM

Safety kernels should be resident in non-volatile ROM or in protected memory that cannot be overwritten by the computing system.

4.5.5 Safety Kernel Independence

A safety kernel, if implemented, must be designed and implemented in such a manner that it cannot be corrupted, misdirected, delayed, or inhibited by any other program in the system.

[12] See previous discussion of the software environment.

4.5.6 Inadvertent Jumps

The system must detect inadvertent jumps within or into Safety Critical Computing System Functions (SCCSFs); return the system to a safe state, and, if practical, perform diagnostics and fault isolation to determine the cause of the inadvertent jump.

4.5.7 Load Data Integrity

The executive program or OS must ensure the integrity of data or programs loaded into memory prior to their execution.

4.5.8 Operational Reconfiguration Integrity

The executive program or OS must ensure the integrity of the data and programs during operational reconfiguration.

4.6 Interface Design Requirements

The design requirements of this section apply to the design of input/output interfaces.

4.6.1 Feedback Loops

Feedback loops from the system hardware must be designed such that the software cannot cause a runaway condition due to the failure of a feedback sensor. Known component failure modes must be considered in the design of the software and checks designed into the software to detect failures.

4.6.2 Interface Control

SCCSFs and their interfaces to safety-related hardware must be controlled at all times, i.e., the interface must be monitored to ensure that erroneous or spurious data does not adversely affect the system, that interface failures are detected, and that the state of the interface is safe during power-up, power fluctuations and interruptions, and in the event of system errors or hardware failures.

4.6.3 Decision Statements

Decision statements in safety-related computing system functions must not rely on inputs of all ones or all zeros, particularly when this information is obtained from external sensors.

4.6.4 Inter-CPU Communications

Inter-CPU communications must successfully pass verification checks in both CPUs prior to the transfer of safety-related data. Periodic checks must be performed to ensure the integrity of the interface. Detected errors must be logged. If the interface fails several consecutive transfers, the operator must be alerted and the transfer of safety-related data terminated until diagnostic checks can be performed.

4.6.5 Data Transfer Messages

Data transfer messages must be of a predetermined format and content. Each transfer must contain a word or character string indicating the message length (if variable), the type of data and content of the message. As a minimum, parity checks and checksums must be used for

verification of correct data transfer. CRCs must be used where practical. No information from data transfer messages must be used prior to verification of correct data transfer.

4.6.6 External Functions

External functions requiring two or more safety-related signals from the software (e.g., arming of an ignition safety device or arm fire device and release of an air launched weapon) must not receive all of the necessary signals from a single input/output register or buffer.

4.6.7 Input Reasonableness Checks

Limit and reasonableness checks, including time limits, dependencies, and reasonableness checks, must be performed on all analog and digital inputs and outputs prior to safety-related functions' execution based on those values. No safety-related functions must be executable based on safety-related analog or digital inputs that cannot be verified.

4.6.8 Full Scale Representations

The software must be designed such that the full scale and zero representations of the software are fully compatible with the scales of any digital-to-analog, analog-to-digital, digital-to-synchro, and/or synchro-to-digital converters.

4.7 Human Interface

The design requirements of this section apply to the design of the human interface to safety-related computing systems.

4.7.1 Operator/Computing System Interface

Computer/Human Interface (CHI) Issues

Displays

Duplicated where possible, SCCSF displays to be duplicated by non-software generated output, designed to reduce human errors, quality of display, clear and concise

Hazardous condition alarms/warnings

Easily distinguished between types of alerts/warning, corrective action required to clear

Process cancellation

Multiple operator actions to initiate hazardous function

Detection of improper operator entries

4.7.2 Computer/Human Interface Issues

CHI issues are not software issues per se - they are really a distinct specification and design issue for the system. However, many of the CHI functions will be implemented in software, and CHI issues frequently are treated at the same time as software in milestone reviews.

- Has the supplier explicitly addressed the safety-related aspects of the design of the CHI? Has this included analysis of anticipated single and multiple operator failures? What kind of human factors, ergonomic, and cognitive science analyses were done (e.g., of cognitive overload, ambiguity of display information)?

- Does the design ensure that invalid operator requests are flagged and identified as such to the operator (vs. simply ignoring them or mapping them silently to "correct" values)?
- Does the supplier ensure that the system always requires a minimum of two independent commands to perform safety-related function? Before initiating any critical sequence, does the design require an operator response or authorization?
- Does the supplier ensure that there are no "silent mode changes" that can put the system in a different safety-related state without operator awareness (i.e., does the design not allow critical mode transitions to happen with notification)?
- Does the supplier ensure that there is a positive reporting of changes of safety-related states?
- Does the system design provide for notification that a safety function has been executed, and is the operator notified of the cause?
- Are all critical inputs clearly distinguished? Are all such inputs checked for range and consistency validity?

4.7.3 Processing Cancellation

The software must be designed such that the operator may cancel current processing with a single action and have the system revert to a designed safe state. The system must be designed such that the operator may exit potentially unsafe states with a single action. This action must revert the system to a known safe state. (e.g., the operator must be able to terminate missile launch processing with a single action which must safe the missile.) The action may consist of pressing two keys, buttons, or switches at the same time. Where operator reaction time is not sufficient to prevent a mishap, the software must revert the system to a known safe state, report the failure, and report the system status to the operator.

4.7.4 Hazardous Function Initiation

Two or more unique operator actions must be required to initiate any potentially hazardous function or sequence of functions. The actions required must be designed to minimize the potential for inadvertent actuation, and must be checked for proper sequence.

4.7.5 Safety-related Displays

Safety-related operator displays, legends and other interface functions must be clear, concise, and unambiguous, and where possible, be duplicated using separate display devices.

4.7.6 Operator Entry Errors

The software must be capable of detecting improper operator entries or sequences of entries or operations and prevent execution of safety-related functions as a result. It must alert the operator to the erroneous entry or operation. Alerts must indicate the error and corrective action. The software must also provide positive confirmation of valid data entry or actions taken (i.e., the system must provide visual and/or aural feedback to the operator such that the operator knows that the system has accepted the action and is processing it). The system must also provide a real-time indication that it is functioning. Processing functions requiring several seconds or longer must provide a status indicator to the operator during processing.

4.7.7 Safety-related Alerts

Alerts must be designed such that routine alerts are readily distinguished from safety-related alerts. The operator must not be able to clear a safety-related alert without taking corrective action or performing subsequent actions required to complete the ongoing operation.

4.7.8 Unsafe Situation Alerts

Signals alerting the operator to unsafe situations must be directed as straightforward as practical to the operator interface.

4.7.9 Unsafe State Alerts

If an operator interface is provided and a potentially unsafe state has been detected, the system must alert the operator to the anomaly detected, the action taken, and the resulting system configuration and status.

4.8 Critical Timing And Interrupt Functions

The following design requirements and guidelines apply to safety-related timing functions and interrupts.

4.8.1 Safety-related Timing

Safety-related timing functions must be controlled by the computer and must not rely on human input. Safety-related timing values must not be modifiable by the operator from system consoles, unless specifically required by the system design. In these instances, the computer must determine the reasonableness timing values.

4.8.2 Valid Interrupts

The software must be capable of discriminating between valid and invalid (i.e., spurious) external and/or internal interrupts. Invalid interrupts must not be capable of creating hazardous conditions. Valid external and internal interrupts must be defined in system specifications. Internal software interrupts are not a preferred design as they reduce the analyzability of the system.

4.8.3 Recursive Loops

Recursive and iterative loop must have a maximum documented execution time. Reasonableness checks will be performed to prevent loops from exceeding the maximum execution time.

4.8.4 Time Dependency

The results of a program should not be dependent on the time taken to execute the program or the time at which execution is initiated. Safety-related routines in real-time programs must ensure that the data used is still valid (e.g., by using senescence checks).

4.9 Selection of Language

4.9.1 High-level Language Requirement

For the vast majority of software systems, the software shall be written in a high level language.

4.9.2 Implementation Language Characteristics

The implementation language or subset shall have the following characteristics:

- Formally-defined syntax
- Strongly typed
- Block structured
- Predictable program execution
- Avoidance of features that are ambiguous to a human
- Ability to impose naming conventions and format that aid readability

4.9.3 Compilers

The implementation of software compilers shall be validated to ensure that the compiled code is fully compatible with the target computing system and application (may be done once for a target computing system).

4.9.4 Automated and tool assisted processes

All tools used in software safety processes shall have sufficient safety assurance to ensure that they do not undermine the integrity of the assurance process. Each tool shall be evaluated to determine its role and significance within the software safety process.

4.10 Coding and Coding standards

4.10.1 Modular Code

Software design and code shall be modular. Modules shall have one entry and one exit point.

4.10.2 Number of Modules

The number of program modules containing safety-related functions shall be minimized where possible within the constraints of operational effectiveness, computer resources, and good software design practices.

4.10.3 Size of Modules

The size of a programming module should be no longer than a printed-page long. This enhances readability and reduces complexity of the code, allowing for easier testing.

4.10.4 Execution Path

Safety Critical Computing System Functions (SCCSFs) shall have one and only one possible path leading to their execution.

4.10.5 Halt Instructions

Halt, stop or wait instructions shall not be used in code for safety-related functions. Wait instructions may be used where necessary to synchronize input/output, etc. and when handshake signals are not available.

4.10.6 Single Purpose Files

Files used to store safety-related data shall be unique and shall have a single purpose. Scratch files, those used for temporary storage of data during or between processes, shall not be used for storing or transferring safety-related information, data, or control functions.

4.10.7 Unnecessary Features

The operational and support software shall contain only those features and capabilities required by the system. The programs shall not contain undocumented or unnecessary features. Every line of code shall map to a design requirement as well as a defined test procedure.

4.10.8 Indirect Addressing Methods

Indirect addressing methods shall be used only in well-controlled applications. When used, the address shall be verified as being within acceptable limits prior to execution of safety-related operations. Data written to arrays in safety-related applications shall have the address boundary checked by the compiled code.

4.10.9 Uninterruptible Code

Safety critical code shall not be interruptible. If interrupts are used, sections of the code which have been defined as uninterruptible shall have defined execution times monitored by an external timer.

4.10.10Safety-related Files

Files used to store or transfer safety-related information shall be initialized to a known state before and after use. Data transfers and data stores shall be audited where practical to allow traceability of system functioning and data integrity

4.10.11Unused Memory

All processor memory not used for or by the application program shall be initialized to a pattern that will cause the system to revert to a safe state if executed

4.10.12Overlays of Safety-related Software

Overlays shall not be used for safety related code.

4.10.13Operating System Functions

If an OS function is provided to accomplish a specific task, application programs shall mitigate possible errors by the OS through the use of wrappers, boundary conditions, and exception handlers.

4.10.14Flags and Variables

Flags and variable names shall be unique. Flags and variables shall have a single purpose and shall be defined and initialized prior to use.

4.10.15Loop Entry Point

Loops shall have one and only one entry point. Branches into loops shall not be used. Branches out of loops shall lead to a single exit point placed after the loop within the same module.

4.10.16Critical Variable Identification

Safety-related variables must be identified in such a manner that they can be readily distinguished from non-safety-related variables (e.g., all safety-related variables begin with a letter S).

4.10.17Variable Declaration

Variables or constants used by a safety-related function shall be declared/initialized at the module level

4.10.18Global Variables

Global variables shall not be used in safety related applications.

4.10.19Unused Executable Code

Operational program loads which contain unused executable code shall:

- Be identified as active code for safety analyses and V&V
    - Safety assessment should include system-level impacts if executed
    - Assessments shall be conducted at the highest level of rigor of the equipment under control
    - Report findings as a hazard causal factor in the Safety Case, and SwSSWG, and program reviews
- Be mitigated in accordance with the SSPP

4.10.20Unreferenced Variables

Operational program loads shall not contain unreferenced or unused variables or constants.

4.10.21Data Partitioning

Safety related data shall be partitioned away from other non-safety related data.

4.10.22Conditional Statements

Conditional statements shall have all possible conditions satisfied and under full software control.

4.10.23Strong Data Typing

Safety-related functions shall exhibit strong data typing.

4.10.24Annotation of Timer Values

Values for timers shall be annotated in the code.

4.11 Software Maintenance

The requirements and guidelines of this section are applicable to the maintenance of the software in safety-related computing system applications. The requirement applicable to the design and development phase as well as the software design and coding phase are also applicable to the maintenance of the computing system and software

4.11.1 Critical Function Changes

Changes to SCCSFs on deployed or fielded systems must be issued as a complete package for the modified unit or module and must not be patched.

4.11.2 Critical Firmware Changes

When not implemented at the depot level or in manufacturers' facilities under appropriate QC, firmware changes must be issued as a fully functional and tested circuit card. Design of the card and the installation procedures should minimize the potential for damage to the circuits due to mishandling, electrostatic discharge, or normal or abnormal storage environments, and must be accompanied with the proper installation procedure.

4.11.3 Software Change Medium

When not implemented at the depot level or in manufacturers' facilities under appropriate QC, software changes must be issued as a fully functional copy on the appropriate medium. The medium, its packaging, and the procedures for loading the program should minimize the potential damage to the medium due to mishandling, electrostatic discharge, potential magnetic fields, or normal or abnormal storage environments, and must be accompanied with the proper installation procedure.

4.11.4 Modification Configuration Control

All modifications and updates must be subject to strict configuration control. The use of automated CM tools is encouraged.

4.11.5 Version Identification

Modified software or firmware must be clearly identified with the version of the modification, including configuration control information. Both physical (e.g., external label) and electronic (i.e., internal digital identification) "fingerprinting" of the version must be used.

4.12 Software Analysis And Testing

The requirements and guidelines of this section are applicable to the software-testing phase.

4.12.1 General Testing Guidelines

Systematic and thorough testing is clearly required as evidence for critical software assurance; however, testing is "necessary but not sufficient." Testing is the chief way that evidence is provided about the actual behavior of the software produced, but the evidence it provides is always incomplete since testing for non-trivial systems is always a sampling of input states and not an exhaustive exercise of all possible system states. In addition, many of the testing and reliability estimation techniques developed for hardware components are not directly applicable to software; and care must, therefore, be taken when interpreting the implications of test results for operational reliability.

Testing to provide evidence for critical software assurance differs in emphasis from general software testing to demonstrate correct behavior. There should be a great deal of emphasis placed on demonstrating that even under stressful conditions, the software does not present a hazard; this means a considerable amount of testing for critical software will be fault injection, boundary condition and out-of-range testing, and exercising those portions of the input space that

are related to potential hazards (e.g., critical operator functions, or interactions with safety-related devices). Confidence in the results of testing is also increased when there is evidence that the assumptions made in designing and coding the system are not shared by the test suppliers (i.e., that some degree of independence between testers and suppliers has been maintained).

- Does the supplier provide evidence that for critical software testing has addressed not only nominal correctness (e.g., stimulus/response pairs to demonstrate satisfaction of functional requirements) but robustness in the face of stress? This includes a systematic plan for fault injection, testing boundary and out-of-range conditions, testing the behavior when capacities and rates are extreme (e.g., no input signals from a device for longer than operationally expected, more frequent input signals from a device than operationally expected), testing error handling (for internal faults), and the identification and demonstration of critical software's behavior in the face of the failure of various other components.
- Does the supplier provide evidence of the independence of test planning, execution, and review for critical software? Are unit tests developed, reviewed, executed, and/or interpreted by someone other than the individual supplier? Has some amount of independent test planning and execution been demonstrated at the integration test level?
- Has some amount of independent 'free play" testing been provided? If so, during this testing is there evidence that the critical software is robust in the face of "unexpected" scenarios and input behavior, or does this independent testing provide evidence that the critical software is "fragile"? (free play testing should place a high priority on exercising the critical aspects of the software and in presenting the system with the kinds of operational errors and stresses that the system will face in the field.)
- Does the supplier's software problem tracking system provide evidence that the rate and severity of errors exposed in testing is diminishing as the system approaches operational testing, or is there evidence of "thrashing" and increasing fragility in the critical software? Does the problem tracking system severity classification scheme reflect the potential hazard severity of an error, so that evidence of the hazard implications of current Problems can be reviewed?
- Has the supplier provided evidence that the tests that exercise the system represent a realistic sampling of expected operational inputs? Has some portion of testing been dedicated to randomly selected inputs reflecting the expected operational scenarios? (This is another way to provide evidence that implicit assumptions in the design do not represent hazards in critical software, since the random inputs will be not selectively "screened" by implicit assumptions.)

4.12.2 Trajectory Testing for Embedded Systems

There is a fundamental challenge to the amount of confidence that software testing can provide for certain classes of programs. Unlike "memory-less" batch programs that can be completely defined by a set of simple stimulus/response pairs, these programs "appear to run continuously...One cannot identify discrete runs, and the behavior at any point may depend on events arbitrarily far in the past." In many systems where there are major modes or distinct partitioning of the program behavior depending on state, there is mode-remembered data that is retained across mode-changes. The key issue for assurance is the extent to which these

characteristics have been reflected in the design and especially in the testing of the system. If these characteristics are ignored and the test set is limited to a simplistic set of stateless stimulus/response pairs, the extrapolation to the operational behavior of the system is seriously weakened.

- Has the supplier identified the sensitivities to persistent stale and the "input trajectory" the system has experienced? Is this reflected in the test plans and test descriptions?
- Are the supplier's assumptions about prohibited or "impossible" trajectories and mode changes explicit with respect to critical functions? "There is always the danger that the model used to determine impossible trajectories over looks the same situation overlooked by the programmer who introduced a serious bug. It is important that any model used to eliminate impossible trajectories be developed independently of the program. Most safety experts would feel more comfortable if some tests were conducted with "crazy" trajectories."

4.12.3 Formal Test Coverage

All software testing must be controlled by a formal test coverage analysis and document. Computer-based tools must be used to ensure that the coverage is as complete as possible.

4.12.4 Go/No-Go Path Testing

Software testing must include GO/NO-GO path testing.

4.12.5 Input Failure Modes

Software testing must include hardware and software input failure mode testing.

4.12.6 Boundary Test Conditions

Software testing must include boundary, out-of-bounds, and boundary crossing test conditions.

4.12.7 Input Data Rates

Software testing must include minimum and maximum input data rates in worst-case configurations to determine the system' capabilities and responses to these conditions.

4.12.8 Zero Value Testing

Software testing must include input values of zero, zero crossing, and approaching zero from either direction and similar values for trigonometric functions.

4.12.9 Regression Testing

SCCSFs in which changes have been made must be subjected to complete regression testing.

4.12.10Operator Interface Testing

Operator interface testing must include operator errors during safety-related operations to verify safe system response to these errors.

4.12.11Duration Stress Testing

Software testing must include duration stress testing. The stress test time must be continued for at least the maximum expected operating time for the system. Testing must be conducted under simulated operational environments. Additional stress duration testing should be conducted to identify potential critical functions (e.g., timing, data senescence, resource exhaustion, etc.) that are adversely affected as a result of operational duration. Software testing must include throughput stress testing (e.g., CPU, data bus, memory, input/output) under peak loading conditions.

5 Previously Developed Software

5.1 Definitions

Previously Developed Software (PDS) includes:

- Commercial off-the-shelf (COTS) software, including:
    - Non-Developmental Items (NDIs) (e.g. OSs and environments, communications handlers, interface handlers, network software, database managers, data reduction and analysis tools, and a variety of other software components that are functionally part of the system. Indirect applications of NDI include programming languages, compilers, software development (e.g., CASE) tools, and testing tools that directly or indirectly affect the applications software in the fielded system)
    - Commercially Developed Items (CDI) (e.g. specific application software packages)
    - GOTS (Government Off the Shelf) software. Software specifically developed for military purposes which may be functionally suitable for reuse in a new development
- Legacy Software (e.g. an existing (typically bespoke) software product that is to be upgraded)
- Reusable software includes libraries of software routines specifically designed and developed for reuse during system development by a system designer. Examples include interfacing software, math routines, protocol handlers, etc. These routines augment other non-developmental software.

5.2 Overview

While use of PDS is a goal to reduce cost, it is a high-risk process for safety. Safety-related PDS will be handled at the same level of rigor as other software handling those hazards. Therefore, there is a hazard action record on each piece of PDS in the safety thread. From the safety perspective PDS can be classified into Information Technology (IT) and equipment under control. Regardless of the PDS pedigree, safety analyses and Verification and Validation (V&V) are required if the PDS is safety related.

Different types of software may raise different issues. There are similar possibilities for assurance arguments and evidence that are discussed in this section. When proposing to make use of PDS the following points should be considered:

- Although there are potential advantages to the use of PDS, lessons learned have shown there may be considerable systems engineering and software safety risk
- The appropriate reuse of well proven software can be of substantial benefit to the integrity of software
- Use of an existing software system can have significant benefits for cost and schedule
- Risks associated with dormant of unused executable code need to be identified and mitigated

- If software is unreliable however, in the sense that it fails more frequently than the tolerable frequencies specified by safety integrity requirements, then no amount of activity in looking for assurance evidence will make it suitable for the safety related role

Virtually all PDS contains unused executable code that may not be discovered. Safety analysis and V&V should be performed consistent with the level of rigor to ensure no anomalous behavior is exhibited. Refer to section 4.10.19 for further requirements.

5.3 Points to Consider for COTS Software

The safety assessment of COTS software poses one of the greatest challenges to the safety certification of systems. COTS software is generally developed for a wide range of applications in the commercial market. Suppliers use an internal company or industry standard for software development. The languages used will generally match the nation and skills of the developing workforce. Since the supplier releases only compiled versions of the product, there is often no way to determine the programming language, the requirements to which the supplier designed the software or the rigor with which they tested the software. Because the PDS suppliers can only intelligently assess the applications for the product, they cannot address specific issues related to a particular application, however, they must supply a full list of features, options, and functions. The PDS supplier attempts to ensure that the product is compatible with many system and software configurations. This often results in additional unnecessary functionality that may introduce hazards.

5.3.1 Documentation

Generally, suppliers provide only user documentation from the PDS suppliers. The type of documentation necessary to conduct detailed analyses is usually not available and limits the software safety engineer's ability to identify system hazard causal factors related to the COTS software. Occasionally it may be possible to purchase the required data or documentation from the COTS supplier (at a premium price) concerning all necessary safety data. Examples of necessary data are a full list of features, functions, and options.

5.3.2 Testing

Testing of the COTS software in an application is very limited in its ability to provide evidence that the software cannot influence system hazards. The testing organization, like the safety organization, must still treat COTS software as a "black box," developing tests to measure the response of the software to input stimulus under (presumably) known system states. Hazards identified through "black box" testing are sometimes happenstance and difficult to duplicate. Timing issues and data senescence issues also are difficult to fully test in the laboratory environment even for software of a known design. Without the ability to analyze the code, determining potential timing, logic, or other problems in the code is difficult at best. Without detailed knowledge of the design of the software, the system safety and test groups can only develop limited testing to verify the safety and fail-safe features of the system. A well-known paradigm of Software Engineering is that it is impossible to completely test any except the most trivial of software programs.

5.3.3 Obsolescence

COTS products typically have a short market life compared with military projects. COTS suppliers exist in a marketplace that affects their supply chain and resources as well as their product decisions. This is particularly the case where the COTS software is delivered as a package with hardware, because hardware components bought in by the COTS supplier are likely to become obsolete. A COTS supplier, whose market strategy relies on continuously upgraded software, may be unwilling or unable to support a particular version of software beyond a short time after its replacement is available. However, a good COTS supplier will have a strategy for continuous evolution to avoid being trapped with an obsolete product. This evolutionary strategy may be incompatible with the needs of a safety related product, especially where the costs of reassurance are high. Fixing at one version, however, may mean that a later, 'big bang' upgrade is not supported by the COTS supplier.

5.3.4 Additional Features

Many COTS suppliers do not have the processes or resources to provide customized versions or features for their software. If the COTS supplier is willing to provide specific features then the supplier is now a sub-contractor and the software is no longer defined as COTS. Note that all sub-contracted software must comply with all aspects of the SSP the cost of adding features will typically be disproportionate to the cost of the baseline product. The additional features will also need to be maintained (at the customer's cost) for future upgrades of the product. Obviously this includes upgrades necessary to fix faults.

5.3.5 Compatibility of COTS Products Upgrades

Where a system is developed using more than one COTS products, each of the COTS products need to be compatible with the others. If there are upgrades, it may not be possible to find a set of compatible versions. This is particularly the case where COTS products are incorporated into a software system e.g. COTS compilation system, operating system, middleware, drivers, libraries etc.

5.4 Points to Consider for Legacy Software

5.4.1 Obsolete Tools & Methods

Legacy software may have been written using software development methods and tools that are obsolete. It may be difficult to obtain tools for old languages (such as compilers) that are supported on modern platforms. It may also be difficult to find competent software engineers willing or able to work with obsolete technology. Safety analysis will be difficult without these tools and will increase costs.

5.4.2 Non-maintained Documentation

Many legacy projects will not have maintained original development documentation when there have been updates. Software by its nature is flexible and the legacy software may have evolved substantially over time without the benefit of documentation. Therefore, safety will require additional time to generate system descriptions and identify and mitigate hazards.

5.4.3 Degradation Through Updates

Changing software over time tends to have the effect of degrading its quality. Updates, especially patches, affect the flow, cohesion and coupling of the software. This will impact documentation pedigree and safety analyses and V&V time and budget. Some code that is no longer required (as a result of changes) may have been left in (i.e. unused executable code). A thorough safety analysis and V&V may require reverse engineering.

5.4.4 Architecture

Changes in architecture have wide-ranging effects on safety, performance, maintainability, sustainability, and utility for use. . The legacy architecture and the architecture into which it is being inserted must be subject to a SHA and V&V.

5.5 Points to Consider for Reusable Software

Reusable software regardless of modifications or not, is subject to the SSP and must comply with safety requirements. Libraries of reusable software, especially commercially developed libraries, suffer from the same shortcomings as other PDS items. Safety analysts will not have the visibility into the source code, design documentation, or requirements documentation for the modules that comprise the resulting software.

All reusable software that is safety-related must be considered as new code. Any software hazard causal factors must be identified, analyzed, mitigated, and that mitigation subject to V&V per SSP. Failure to do so will increase system hazard risk.

5.6 Related Issues

5.6.1 Managing Change

The project must maintain a change configuration board, which notifies the SSP of pending PDS changes. The project must also establish and maintain processes for integrating the many suppliers' upgrades, enhancements, or error corrections to commercial software products on a routine basis. Changes to PDS may introduce new hazards, reopen closed hazards, undo mitigation of hazards and therefore must be subject to the SSP and V&V to manage potential and residual risk.

5.6.2 Configuration Management (CM)

PDS changes are subject to project CM, CCB reviews, and the SSP processes. Suppliers and their sub-contractors must obtain contractual agreements from PDS suppliers that they will be alerted of any changes made to PDS. This rule applies to software that is part of the system as well as software used to develop the system. Failure to do so can result in the release of an unsafe system to the user.

5.7 Assurance Issues

5.7.1 Strategy

The SSPP should include policy for the use of PDS detailing the criteria for acceptability (e.g. appropriate in-service data for the PDS). Where the PDS does not have sufficient assurance for

the application, it may be possible to develop bespoke safeguards to provide the additional assurance. These are often referred to as 'wrappers'.

The PDS assurance strategy must be included in the SSPP and all relevant project processes and documentation. It is acceptable to refer all other project documents to the SSPP. Failure to coordinate this strategy will result in increased project costs and may increase safety risk.

5.7.2 Safety Analysis

A systematic safety analysis should be conducted to determine the differences between the safety requirements required for the role the PDS in the safety related system and the functions and properties of the PDS. This may identify:

- Safety requirements that are met by the PDS
- Safety requirements that are not met by the PDS, for which alternative provision needs to be made
- Features and properties of the PDS that are not required (and may even be inimical to safety) in the safety related system

5.7.3 Operational or In-Service History

In-service history should only be taken into account as evidence of the integrity of legacy software or re-use of software where reliable data exists relating to the in-service user, usage, and failure rates in similar applications for similar periods of time. The operating environments of such PDS should be assessed to determine their relevance to the proposed use in the new application. Determine the acceptability of the PDS, taking into account:

- Length of service period
- The operational in-service hours within the service period, allowing for different operational modes and the numbers of copies in service
- Definition of what is counted as a fault/error/failure

The suitability of the quantified methods, assumptions, rationale and factors relating to the applicability of the data must be justified.

5.7.4 Versions, Variants and Problem Reporting

Configuration changes during the software's service life should be identified and assessed in order to determine the stability and maturity of the software and to determine the applicability of the entire service history data to the particular version to be incorporated in the software.

In order for in-service history of the PDS to provide useful assurance evidence, there will need to have been effective configuration management of the PDS and a demonstrably effective problem reporting system. In particular, both the software and the associated service history evidence should have been under configuration management throughout the software's service life.

5.7.5 Visibility of Process

Evidence developed during the development may be available for use in the assurance of the safety related system when PDS has been developed under verified and approved processes which meet the requirements of a recognized standard or best practices of the software supplier.

It is necessary to ensure that any differences between the original and new development environments and tool suites will not reduce the safety integrity of the software. Where differences exist, that are not covered already by assurance evidence, V&V activities should be repeated to a degree necessary to provide assurance in safety. For example, if a different compiler or different set of compiler options are used, resulting in different object code, it will be necessary to repeat all or part of the test and analysis activities that involve the object code, particularly with respect to the safety related parts of the code.

Reverse engineering may be an option where there are gaps in the documentation required for assurance of process (see the section on Reverse Engineering and Design Reviews below).

5.7.6 Upgrades

All changes to PDS made as part of its incorporation in the safety related software should be considered as new software and assurance evidence and arguments should be developed. For a legacy product that is to be upgraded, the legacy part will typically be argued as a grandfather rights/previous history case but the new upgrades should be performed to the appropriate safety integrity argument. However, if only black box information is known for the legacy part, the interaction of legacy and upgraded parts will also be reliant on empirical arguments thus reducing confidence in this aspect. Additional assurance activities, such as wrapping the legacy software, are therefore required to provide evidence that there are no adverse interactions.

Where there is white box visibility of the PDS, it should be checked for code that is unreachable in its new context. Unreachable code should be documented in the Software Design and a decision made as to whether it is safer to leave it in or remove it.[13] Unreachable code may be allowed to remain in the final application where it can be shown that the risks of leaving it in are less than the risks of modifying the code to remove it. Where there is unreachable code, additional safeguards may be necessary to mitigate the effects should it be inadvertently executed.

5.7.7 Wrappers

Software may need to be developed to interface and interact between the safety related system and the PDS. In such a case, the safe-by-design argument for the wrapper may be the primary argument for safety and could be combined with an argument for appropriateness of the PDS product. This wrapper could allow a lower safety integrity product to be used in a higher safety integrity environment. The requirements for the wrapper should be determined from the safety analysis for the use of the PDS.

5.7.8 Middleware

The use of middleware can be a fault-isolation design technique to provide a means of isolating (e.g., firewalls) the PDS from the safety-related functions. However, middleware also delays or defeats real-time safety monitoring and control of hazards. The middleware consists of two layers, an interfacing layer to the OS and an interfacing layer to the applications software. Features in the middleware can "capture" undesired events from the OS. A significant advantage to middleware is that it can minimize the impact of changes to the OS or other PDS. Only the

[13] That decision may depend heavily on the availability of qualified tools to make the modifications, especially for legacy software developed using outdated languages.

interfacing layer to the OS requires modification to accommodate the change. To minimize the safety impact of middleware, it must be tailored and designed to meet the time and sequence requirements for all safety-related threads. Failure to do so will result in unreported, unsafe performance and monitoring of safety mitigations.

5.7.9 Reverse Engineering and Design Reviews

One way to approach the assurance of PDS is via reverse engineering.[14] Access to the source code, design documentation and the software supplier is the least expensive way for this to be practicable.

The primary aims of the reverse engineering, from the assurance perspective, are to provide an unambiguous and complete definition of what the software does and to verify, to an appropriate level of confidence, that the software meets this definition. A secondary aim is to produce documentation that will assist the assessment of the PDS for use with the safety related software and to facilitate future maintenance.

5.7.10 Additional Verification and Validation

All PDS should be subject to at least validation to ensure that it performs acceptably in the system context. The amount of assurance evidence already available will dictate the amount of additional verification and validation necessary. This can provide demonstration evidence.

5.7.11 Eliminating OS Functionality

Eliminating unnecessary functionality from OSs and environments reduces the risk that these functions will corrupt safety-related functions. It may not be possible, and occasionally even risky, to eliminate functions from OSs or environments. Generally, one eliminates the functionality by preventing certain modules from loading. However, there may be interactions with other software modules in the system not obvious to the user. This interdependency, particularly between apparently unrelated system modules, may cause the software to execute unpredictably or to halt. Any changes to the OS will require a full system V&V plus full software safety regression testing to ensure the reduced risk has been realized.

[14] Ensure that any license agreements with the supplier permit reverse engineering.

6 Testing and Assessment Guidelines

Testing is an incremental risk reduction technique verifying requirements and specifications. Testing methodologies of software using either "white-box" or "black-box"technique are limited by the visibility that safety analysts have into the internal functionality of the software. Therefore, safety-specific testing, such as using hazard scenario data, failure mode, or fault insertion testing will be limited in its ability to provide evidence that the software cannot influence system hazards. The testing organization may need to employ specialized methods of testing, including mutation testing or perturbation testing, to provide sufficient evidence for safety certification. Hazards are not closed until a completely specified test suite has been successfully performed and safety results verified on the unit level, Computer Software Configuration Item (CSCI) level, and system level.

Software testing should include testing at each of the following levels during the development life cycle:

- Unit Level Testing – Testing done during the development of the software, but performed by someone other than the writer of that code. Software safety analyses data is used to check logic for monitoring and control of the hazard.
- CSCI Level Testing – Testing done to integrate certain code segments that will form a sub-system of the overall system, also performed by someone other than the writer of that code. For example, this could include the fire control subsystem of an overall system including four components; fire control, drive control, user interface and ballistic solution calculator. Software safety analyses data is used to check sharing of resources and communication of status, states, modes, and hazards
- System Level Testing – Testing done on the system as a whole by someone other than the writer of that code. This includes the integration of all subsystems and is representative of the final product to be fielded (hardware as well as software). Software safety analyses data is used to check interfaces and system level requirements for safety.
- Test Coverage Analysis – (Source; DO-178B 1 December 1992) A two-step process, involving requirements-based coverage analysis and structural coverage analysis. The first step analyzes the test cases in relation to the software requirements to confirm that the selected test cases satisfy the specified criteria. The second step confirms that the requirements-based test procedures exercised the code structure. Structural coverage analysis may not satisfy the specified criteria. Additional guidelines are provided for resolution of such situations as dead code (in DO-178B, subparagraph 6.4.4.3). Software safety will use test coverage analyses results to confirm system level safety requirement were tested.

The following are some of the types of tests that can be performed on software, and depending on the complexity and criticality of the code module you will need to include different types to rigorously test and evaluate the software. Types of software testing include, but may not be limited to:

- Functionality testing – This is the most common testing and includes such tests as FQT because it's requirements based testing. In this testing, the requirements and design are verified to operate correctly under normal operating procedures. Software safety requirements are a subset of the overall requirements being tested. Completion of

functional testing that validates safety requirements is indispensable in achieving hazard closure and risk reduction.

- Stress Testing – This type of testing includes any testing that represents a worst-case scenario in terms of such things as message traffic and expected inputs and outputs. This testing is used to verify that the system either acts correctly (as per the normal operating procedures) or fails safe during these extreme conditions. The value of this type of testing to software safety is that it is usually the first long-term continuous operation of the system software and validates stability.
- Boundary and Out of Bounds Testing – This refers to testing at, near, and across the boundaries and stress testing where the system receives an unexpected value forprocessing. For example, if the user can choose a distance from anywhere between 100 meters and 10 kilometers, and the tester enters 70 meters, how will the system handle that situation? The value of this type of testing to software safety is that it verifies that the system either acts correctly or fails safe (does not create a hazard).
- Fault Insertion Testing (FIT) – This testing can be considered a subset of stress testing, but relates to inserting faults directly into the system (possibly at the CSCI or unit level) in order to test the handling of such events by the system (as laid out in the requirements). FIT requires a software safety engineer to assist in the test design to insert faults that exacerbate or cause hazardous conditions. In all cases, the software safety team must have valid data from previous analyses to begin this task. This testing is used to verify that the system either acts correctly (as per the normal operating procedures) or fails safe during the operation of these test cases. This type of testing is the only purposeful way for software safety to verify exception handling for safety-related components.

6.1 Generic Test Requirements

The following requirements should be utilized when developing a testing strategy for any software program, especially those involving safety-related aspects. The application of these generic test requirements is dependent on the level of rigor and scale of the system/software under test. These requirements will increase the confidence in the implementation of safety functions in the system/software. Any deletions from or changes in the set of requirements below will be assessed for safety risk impact by the SwSSWG and SSWG, and logged into the Safety Case by the supplier. The minimum testing requirements should be confirmed by the review authority. Failure to do so will delay the approval process and complicate the project schedule.

All safety-related software testing shall be defined and controlled by a formal document. The means to assess and approve deviations to the test processes or formal document will be addressed in the SSPP.

- New or Modified Code effecting safety critical functions shall be tested to the same standards originally designated for the module.
- Tests shall exercise program logic (decision coverage) through nominal safety related paths (branch and path coverage) and through the combination of contingency and error paths (FIT, stress testing, boundary and out-of-bounds testing, and exception handling).
- Testing shall be used to verify correct implementation of safety-critical requirements under off-nominal and fault conditions.

- Any fault condition not able to be verified through the normal operation of the system shall be tested through the implementation of fault-insertion testing using validated simulators/stimulators, emulators, and test equipment.
- All system functions that may adversely affect the safety of equipment or personnel shall be tested as part of the system performance verification tests using validated simulators/stimulators or emulators, where possible, to protect equipment and personnel.
- All software tests which may adversely affect the safety of personnel or equipment shall be formally identified to the test IPT, SwSSWG and SSWG and logged into the Safety Case.
- Exit Criteria for safety shall be defined for each test objective that determines successful test requirements or execution of safety-related functions prior to test execution.
- All safety requirements shall be traceable to a specific test, or multiple tests, designed to investigate its implementation. *This will be found in the RTM.*
- All safety design elements shall be traceable to a specific test, or multiple tests, designed to investigate its implementation. *This will be found in the RTM.*

6.2 Test Recommendations

The following recommendations provide methods and prescriptions for executing a rigorous software safety-testing program.

- Testing should include minimum and maximum input data rates in worst-case configurations to determine the system's capabilities and responses to these conditions. Testing should also include presenting data at a rate greater than the maximum specified and less than the minimum specified to establish a baseline for the system behavior.
- Software testing should include GO/NO-GO path testing.

6.3 Test Execution

The following are guidelines on things to consider when actually conducting tests on software.

- Testing should be conducted utilizing an approved test facility where software execution and environment is controlled and monitored.
- Execution of tests shall be in accordance with the approved test procedures.
- All tests shall document test objectives prior to execution.
- No tests should be performed on code segments by individuals who took part in developing that specific code segment.

6.4 Results and Analysis

The following are ways in which to manage the data collected during the testing phases and how to properly manage the testing program and verify that all requirements have been met through the testing process.

- Safety related test failures occurring during system integration and test shall be formally documented and tracked.

- Any safety problems discovered during testing shall be analyzed and documented in discrepancy reports as well as test reports. Discrepancy reports shall contain recommended solutions for the identified safety problems.

6.5 Guidelines for Previously Developed Software (PDS) Testing

The following should be used for any safety-related PDS that has insufficient supporting requirements or design documentation available, that has been incorporated into the system being tested.

- A full software safety analysis shall be performed when source code is available.
- If source code is not available, special safety tests shall be performed that include load and stress testing, using the full range of inputs as well as out of range inputs, to elicit unsafe program behavior. The test cases should include exception handling of any erroneous I/O from the PDS piece of software.
- Testing shall ensure that COTS cannot cause a single point failure leading to a hazard or be the sole control (initiate or active decision maker) of a system hazard.
- The interfaces to the system shall be tested to ensure that they pass only required information to the rest of the system and react to unintended information in a safe manner. The PDS will be tested to ensure that the interface software provides all the information that the PDS software requires at all times.

6.6 Software Testing Process

The review authorities will be expecting the project to lead them through the following bullets for planning for and executing both developmental and Functional Qualification Testing (FQT). This process will help to describe the necessary steps and considerations in order to effectively, rigorously and efficiently test the software that is being developed. The formal names of the various groups may be different to each organization, but the process in planning, executing and evaluating the software through all testing phases is applicable to all software developments.

6.6.1 Testing During the Development Phase

- Software testing will be done at every logical step in the development process. This includes, but is not limited to, at the unit level, the sub-system level and the system level.
- During the requirements development phase, test engineers should be involved to ensure the testability of the requirements in order to both determine the proper method for testing and to ensure that the necessary resources (people, equipment and facilities) are available and planned to be ready when testing is scheduled.
- At each development phase, test equipment such as simulators, stimulators, emulators and test environments must be taken into account including the design, qualification and cost effectiveness. These components will be paramount in the verification that the unit and sub-system level code is truly meeting the requirements for the software and system in general.

- All testing performed should have detailed plans, procedures and reports generated each and every time testing is performed to not only report back to the development team, but also to determine how the development of the code is progressing and if the code is truly maturing and meeting the intended requirements.
- All unit level testing should be performed by individuals other than those who actually wrote the code. This will add a necessary independence to the verification of the code in that an individual with less knowledge about how the code was written, but adequate knowledge of how the requirements are being met by this code unit, can truly verify the intended requirements are met.
- Sub-system level testing, or integration testing, is the first check that the individually developed units of code not only meet their own specific requirements, but that they work in conjunction with the other units developed that they will need to interact with regularly during the performance of the system. With this integration, more robust testing techniques may be utilized to determine not only if the specific code units perform properly, but also to determine if the units pass the correct values to the other units that it is interacting with and to determine if any and all possible exceptions or erroneous inputs or outputs are handled properly by the receiving unit of code.
- Once the code has reached a point that it has been integrated into all of its subsystems and all of those subsystems meet the requirements and have passed all unit and sub-system level testing satisfactorily, a test readiness review should be conducted to insure that the system is ready for FQT. This also requires that all test plans, procedures, facilities, personnel and test equipment necessary to perform FQT is also in place and ready to begin the testing.

6.6.2 Functional Qualification Testing and Independent Verification and Validation

- Software testing will include hardware in the target environment. This is necessary, since the integration and interaction between hardware and software is the area that usually cause the most issues during testing. If testing is conducted on the actual hardware that it will ultimately be used upon as early in the development process as possible, then these issues can be resolved before they become more costly and potentially require "stepping back" in the life cycle in order to address design issues that could have been mitigated by finding them during unit level or subsystem level testing. This is much more amiable as opposed to the costlier predicament of having to redesign during FQT or FQT Regression Testing.
- Testing should include parameters to validate the interaction between the operator and the system. This could include exception handling if the operator tries to access programs that the system should not be allowing, or simply not allowing the operator to cause an unsafe condition by holding down one button or entering in improper information. Both the system and the operator must be able to check the validity of each other's inputs and outputs.
- Prior to executing the FQT, after the testers have written the FQT procedures, the rest of the team (including the software engineers, software quality engineers and software safety engineers) must evaluate the adequacy and completeness of the FQT procedures

and trace it back to the requirements verification methods, procedures and mitigation strategies included in the hazard analyses. All safety critical requirements are to be addressed in verification plans, which can be seen in an updated RTM to show the section of the test plan that relates to each requirement and design component. The SwSSWG must verify that all SSRs can be traced from system level specifications to design documentation /diagrams to test plans /procedures (for this, use a hazard tracking tool, e.g. RTM).

- The test plan, descriptions and procedures will be reviewed by the SwSSWG to ensure that they address all safety requirements adequately, as laid out in the hazard analyses. The test plans, descriptions, and procedures must also be checked by the software quality engineer to verify that they are consistent with one another. Safety tests will be performed by independent evaluators, meaning those who did not develop the code themselves. The software quality and software safety engineers shall take part in such activities as witnessing tests, reviewing test results and reporting any test incidents to the appropriate level.
- A Data Review Board (DRB) shall be initiated and chaired by the software quality engineer, inviting all stakeholders, usually limited to the SwSSWG, but can involve the test community, or program management based on the test issues that are to be discussed at each meeting, and the importance of any decisions that must be made during the DRB. A DRB is an Integrated Project Team meeting to review the Software Problem Reports (SPR) to qualify, rank and assign them for disposition.
- All SPR from testing should be resolved or have an assignee actively pursuing closure. Acceptable closure or mitigation of an SPR include design change, deferment to a future spiral development or block upgrade, or informational with no action required. All non-deferred and non-informational SPRs mitigations must be verified through testing, as defined in the hazard analyses through analysis of safety test results.
- If necessary, prepare a system level risk assessment to address those risks that have not been fully mitigated. This assessment shall define the constraints on the system due to the safety-critical failure(s) that will not be mitigated through design change, but rather through procedure augmenting.
- The SwSSWG will take the SPR list and other safety-critical testing results to determine the residual risk that could not be mitigated and is still inherent to the system. The system safety engineer and software safety engineer will then update any safety assessment documentation, which will then be reviewed by the rest of the SwSSWG.

Guidance on Software Safety Design and Assessment of Munition-Related Computing Systems

ANNEXES

AOP-52

Contents

Figures

A Acronyms and Terms of Reference

A.1 Acronyms

AECL	Atomic Energy of Canada Limited
ALARP	As Low As Reasonably Practicable
ARP	Aerospace Recommended Practice
ARTE	Ada Runtime Environment
CASE	Computer-Aided Software Engineering
CCB	Configuration Control Board
CDI	Commercially Developed Item
CDR	Critical Design Review
CHI	Computer/Human Interface
CI	Configuration Item
CM	Configuration Management
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CSCI	Computer Software Configuration Item
CSR	Component Safety Requirement
CTA	Critical Task Analysis
DAL	Development Assurance Level
DDA	Detailed Design Analysis
DFD	Data Flow Diagram
DOD	Department of Defense
DOT	Department of Transportation
DSMC	Defense Systems Management College
DU	Depleted Uranium
E/E/PES	Electrical/Electronic/Programmable Electronic Systems
EIA	Electronic Industries Association
EMP	Electro-Magnetic Pulse
EOD	Explosive Ordnance Disposal
ESH	Environmental Safety and Health
FAA	Federal Aviation Administration
FCA	Functional Configuration Audit
FFD	Functional Flow Diagram
FIT	Fault Insertion Testing

FQT	Functional Qualification Test
FTA	Fault Tree Analysis
GOTS	Government Off-The-Shelf
GSSRL	Generic Software Safety Requirements List
HAR	Hazard Action Record
HHA	Health Hazard Assessment
HMI	Human/Machine Interface
HRI	Hazard Risk Index
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronic Engineering
ILS	Integrated Logistics Support
IPD	Integrated Product Development
IPT	Integrated Product Team
IT	Information Technology
IV&V	Independent Verification & Validation
LOT	Level of Trust
MIL-STD	Military Standard
NASA	National Aeronautics and Space Administration
NDI	Non-Developmental Item
O&SHA	Operating and Support Hazard Analysis
OOA&D	Object Oriented Analysis & Design
OS	Operating System
PAF	Programmable logic device Analysis Folder
PDL	Program Design Language
PDS	Previously Developed Software
PFD	Process Flow Diagram
PFS	Principal for Safety
PHA	Preliminary Hazard Analysis
PHL	Preliminary Hazard List
PLD	Programmable Logic Devices

QA	Quality Assurance
QAP	Quality Assurance Plan
QC	Quality Control
RHA	Requirements Hazards Analysis
ROM	Read Only Memory
RTCA	RTCA, Inc.
RTM	Requirements Traceability Matrix
SAF	Software Analysis Folder
SAR	Safety Assessment Report
SCC	Software Control Category
SCCSF	Safety Critical Computing System Functions
SCLI	Software Criticality Level Index
SCLM	Software Criticality Level Matrix
SCM	Software Configuration Management
SDL	Safety Data Library
SDP	Software Development Plan
SDR	System Design Review
SEDS	Systems Engineering Detailed Schedule
SEE	Software Engineering Environment
SHA	System Hazard Analysis
SIL	Safety Integrity Level
SQA	Software Quality Assurance
SRA	Safety Review Authority
SRCA	Safety Requirements Criteria Analysis
SRCSF	Safety-related Computing System Functions
SRFL	Safety-related Functions List
SSE	Software Safety Engineer
SSG	System Safety Group
SSHA	Subsystem Hazard Analysis
SSMP	System Safety Management Plan
SSP	System Safety Program
SSPP	System Safety Program Plan
SSR	Software Safety Requirements
SSS	Software System Safety
SSWG	System Safety Working Group
STP	Software Test Plan
STR	Software Trouble Report
SwSE	Software Safety Engineer
SwSPP	Software Safety Program Plan
SwSSP	Software System Safety Program
SwSSWG	Software System Safety Working Group

V&V Verification and Validation

WBS Work Breakdown Structure

A.2 Terms of reference

NOTE: All definitions used in this AOP have either been extracted from various standards such as MIL-STD-882C/D, MIL-STD-498, AOP-15, Def-Stan 00-55/00-56, and Def(Aust) 5679. A [882], [498], [00-55], [00-56], [AOP-15], or [5679] references each definition's source.

Acceptance. An action by an authorized representative of the acquirer by which the acquirer assumes ownership of software products as partial or complete performance of a contract. [498]

Acquiring Agency. An organization that produces software products for itself or another organization. [498]

Architecture. The organizational structure of a system or CSCI, identifying its components, their interfaces, and concept of execution among them. [498]

Automata. A machine or controlling mechanism designed to follow a predetermined sequence of operations or respond to encoded instructions.

Battle Short. (Safety Arc) The capability to bypass certain safety features in a system to ensure completion of a mission without interruption due to the safety feature. Bypassed safety features include such items as circuit current overload protection, thermal protection, etc.

Behavioral Design. The design of how an overall system or CSCI will behave, from a user's point of view, in meeting its requirements, ignoring the internal implementation of the system or CSCI. This design contrasts with architectural design, which identifies the internal components of the system or CSCI, and with the detailed design of those components. [498]

Build. (1) A version of software that meets a specified subset of the requirements that the completed software will meet. (2) The period of time during which such a version is developed. [498]

Commercially Developed Items. Computer programs and/or hardware procured from commercial vendors. Commercially Developed Items are generally designed and sold for a broad range of applications. Examples include Operating Systems and environments, computer program development tools, compilers, microprocessors, etc.

Computer Hardware. Devices capable of accepting and storing computer data, executing a systematic sequence of operations on computer data, or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control, or other logical functions. [498]

Computer Program. A combination of computer instructions and data definitions that enables computer hardware to perform computational or control functions. [498]

Computer Software Configuration Item. An aggregation of software that satisfies an end-use function and is designated for separate configuration management by the acquirer. CSCIs are selected based on tradeoffs among software function, size, host or target computers, developer,

support concept, plans or reuse, criticality, interface considerations, need to be separately documented and controlled, and other factors. [498]

Computing System. A device and its associated interfaces capable of accepting and storing computer data, executing a systematic sequence of operations on computer data, or producing control outputs. Such devices can perform substantial interpretation, computation, communication, control, or other logical functions.

Condition. An existing or potential state such as exposure to harm, toxicity, energy source, activity, etc. [882]

Configuration Item. An aggregation of hardware, software, or both that satisfies an end use function and is designated for separate configuration management by the acquirer. [498]

Contractor. A private sector enterprise or the organizational element of DOD or any other government agency engaged to provide services or products within agreed limits specified by the MA. [882]

Data Type. A class of data characterized by the members of the class and operations that can be applied to them; for example, integer, real, or logical. [IEEE 729-1983]

Dead Code. Executable but not reachable code containing functionality not intended for the current environment.

Deliverable Software Product. A software product that is required by the contract to be delivered to the acquirer or other designated recipient. [498]

Derived Safety Requirement. A design requirement that:

- has a basis in a higher level safety requirement and is allocated to a subsystem or component of the system
- is developed during the hazard analysis process to mitigate specific hazard causal factors
- is developed as a result of the identification of safety related functions at the system integration, system, subsystem, or component level

Design. Those characteristics of a system or CSCI that are selected by the developer in response to the requirements. Some will match the requirements; others will be elaborations of requirements, such as definitions of all error messages; others will be implementation related, such as decisions, about what software units and logic to use to satisfy the requirements. [498]

Designed Safe State. A system state that provides the maximum degree of safety within the constraint of the current operational or logistic phase.

Dormant Code. Reachable and executable code containing functionality appropriate for any previous environments that is not intended for the current environment

Fail Safe. A design feature that ensures that the system remains safe or in the event of a failure will cause the system to revert to a state which will not cause a mishap. [882]

Firmware. The combination of a hardware device and computer instructions and/or computer data that reside as read-only software on the hardware device. [498]

Formal Methods. The application of a mathematical process for the verification of software design compliance with the specification.

Hazard. Any real or potential condition that can cause injury, illness, or death to personnel, damage to or loss of a system, equipment or property; or damage to the environment. [AOP-15]

Hazardous State. A computer program state that may lead to an unsafe state.

Independent Verification & Validation. Systematic evaluation of software products and activities by an agency that is not responsible for developing the product or performing the activity being evaluated. [498]

Managing Activity. The organizational element of DOD assigned acquisition management responsibility for the system, or prime or associate contractors or subcontractors who impose system safety tasks on their suppliers. [882]

Middleware. A computer program or series of programs that functionally isolate application computer programs from non-developmental computer programs.

Mishap. An unplanned event or series of events resulting in death, injury, occupational illness, or damage to or loss of equipment or property, or damage to the environment. [882 and AOP-15]

Mishap Probability. The aggregate probability of occurrence of the individual events that create a specific hazard. [882]

Mishap Risk. An expression of the impact and possibility of a mishap in terms of potential mishap severity and probability of occurrence. A measure of the likelihood of the hazardous event occurring with the consequences if it does occur. [882 and AOP-15]

Mishap Severity. An assessment of the consequences of the worst credible mishap that could be caused by a specific hazard. [882]

Non-Developmental Items. Items, (computer programs, hardware, subsystems, components, etc.), including commercially developed items, government development items, or items from other sources, employed in the design of a system with or without modification.

Patch. A modification to a computer program that is inserted into the program in machine (object) code.

Path. The logical sequential structure that the program must execute to obtain a specific output.

Peer Review. An overview of a computer program presented by the author to others working on similar programs in which the author must defend his implementation of the design.

Process. An organized set of activities performed for a given purpose. [498]

Qualification Test. Testing performed to demonstrate to the acquirer that a CSCI or a system meets its specified requirements. [498]

Reengineering. The process of examining and altering an existing system to reconstitute it in a new form. May include reverse engineering (analyzing a system and producing a representation at a higher level of abstraction, such as design from code), restructuring (transforming a system from one representation to another at the same level of abstraction), redocumentation (analyzing a system and producing user or support documentation), forward engineering (using software

products derived from an existing system, together with new requirements, to produce a new system), retargeting (transforming a system to install it on a different target system), and translation (transforming source code from one language to another, or from one version of a language to another). [498]

Requirement. (1) A characteristic that a system or CSCI must possess in order to be acceptable to the acquirer. (2) A mandatory statement in contractual binding document (i.e., standard, or contract). [498]

Reusable Software Products. A software product developed for one use but having other uses, or one developed specifically to be usable on multiple projects or in multiple roles on one project. Examples include, but are not limited to, commercial-off-the-shelf software products, acquirer-furnished software product, software products in reuse libraries, and pre-existing developer software products. Each use may include all or part of the software product and may involve its modification. [498]

Risk. An expression of the impact and possibility of a mishap in terms of potential mishap severity and probability of occurrence. [882 and AOP-15]

Risk Assessment. A comprehensive evaluation of the risk and its associated impact. [882]

Safety. Freedom from those conditions that can cause death, injury, occupational illness, damage to or loss of equipment or property, or damage to the environment. [882 and AOP-15]

Safety-Critical. A term applied to a condition, event, operation, process, or item of whose proper recognition, control, performance or tolerance is essential to safe system operation or support (e.g., safety-critical function, safety-critical path, safety-critical component. [882 and AOP-15]. In the context of this AOP, all software that affects the safety of the system is "safety related".

Safety Critical Computing System. A computing system containing at least one Safety Critical Function.

Safety Critical Function. A computer software function in which an error or failure can cause a catastrophic mishap.

Safety Kernel: An independent computer program that monitors the state of the system to determine when potentially unsafe system states may occur or when transitions to potentially unsafe system states may occur. The Safety Kernel is designed to prevent the system from entering the unsafe state and return it to a known safe state.

Safety-Related Computer Software Components. Those computer software components and units whose errors can result in a potential hazard, or loss of predictability or control of a system. [882] In the context of this AOP, all software that affects the safety of the system is safety related.

Safety-Significant Functions and Safety-Significant Hazard Causal Factors: In the context of this AOP, all software that is shown to have a "Medium" or "Serious" risk in the Software Safety Criticality Matrix is safety significant.

Software Development. A set of activities that results in software products. Software development may include new development, modification, reuse, reengineering, maintenance, or any other activities that result in software products. [498]

Software Engineering. In general usage, a synonym for software development. As used in MIL-STD 498, a subset of software development consisting of all activities except qualification testing. [498]

Software Failure. An unexpected or unintended action by the software. An error in the software that results in functioning not in accordance with the design requirements and specification.

Software System. A system consisting solely of software and possibly the computer equipment on which the software resides and operates. [498]

Subsystem. A grouping of items satisfying a logical group of functions within a particular system [882 and AOP-15]

System. An integrated composite of people, products, and processes that provide a capability to satisfy a stated need or objective. [882 and AOP-15]

System Safety. The application of engineering and management principles, criteria, and techniques to achieve mishap risk as low as reasonably practicable (ALARP), within the constraints of operational effectiveness and suitability, time and cost, throughout all phases of the life cycle. [AOP-15]

System Safety Engineer. An engineer who is qualified by training and/or experience to perform system safety engineering tasks. [882]

System Safety Engineering. An engineering discipline requiring specialized professional knowledge and skills in applying scientific and engineering principles, criteria, and techniques to identify and eliminate hazards, in order to reduce the associated risk. [882]

System Safety Group/Working Group. A formally chartered group of persons, representing organizations initiated during the system acquisition program, organized to assist the MA system PM in achieving the system safety objectives. Regulations of the military components define requirements, responsibilities, and memberships. [882]

System Safety Management. A management discipline that defines SSP requirements and ensures the planning, implementation and accomplishment of system safety tasks and activities consistent with the overall program requirements. [882]

System Safety Manager. A person responsible to program management for setting up and managing the SSP. [882]

System Safety Program. The combined tasks and activities of system safety management and system safety engineering implemented by acquisition project managers. [882]

System Safety Program Plan. A description of the planned tasks and activities to be used by the contractor to implement the required SSP. This description includes organizational responsibilities, resources, methods of accomplishment, milestones, depth of effort, and integration with other program engineering and management activities and related systems. [882]

Unsafe State. A system state that may result in a mishap.

Unused Executable Code. Dead or dormant code.

Watchdog Timer. An independent, external timer that ensures that the computer cannot enter an infinite loop. Watchdog timers are normally reset by the computer program. Expiration of the timer results in generation of an interrupt, program restart, or other function that terminates current program execution.

B References

B.1 Government References

Australian Defence Standard 5679

Defence Standard 00-54: Requirements for Safety Related Electronic Hardware in Defence Equipment, UK Ministry of Defence, April 1999.

Defence Standard 00-55: Requirements for Safety Related Software in Defence Equipment, UK Ministry of Defence, Issue 2, 1997

Defence Standard 00-56: Safety Management Requirements for Defence Systems, UK Ministry of Defence, Issue 2, 1996

DODD 5000.1, Defense Acquisition,

DOD 5000.2R, Mandatory Procedures for Major Defense Acquisition Programs and Major Automated Information Systems,

Military Standard 882B, System Safety Program Requirements, March 30, 1984

Military Standard 882C, System Safety Program Requirements, January 19, 1993

Military Standard 882D, Standard Practice for System Safety,

FAA Order 1810, Acquisition Policy

FAA Order 8000.70, FAA System Safety Program

RTCA-DO 178B, Software Considerations In Airborne Systems And Equipment Certification, December 1, 1992

COMDTINST M411502D, System Acquisition Manual, December 27, 1994

NSS 1740.13, Interim Software Safety Standard, June 1994

US Air Force, Software Technology Support Center, Guidelines for Successful Acquisition and Management of Software-Intensive Systems: Weapon Systems, Command and Control Systems, Management Information Systems, Version-2, June 1996, Volumes 1 and 2

US Air Force Inspection and Safety Center AFISC SSH 1-1, Software System Safety Handbook, September 5, 1985

B.2 Commercial References

EIA-6B, G-48, Electronic Industries Association, System Safety Engineering In Software Development1990

IEC 61508: International Electrotechnical Commission. Functional Safety of Electrical/Electronic/ Programmable Electronic Safety-Related Systems, December 1997.

IEEE STD 1228, Institute of Electrical and Electronics Engineers, Inc., Standard For Software Safety Plans, 1994

IEEE Standard 1498, Software Development and Documentation,

IEEE STD 829, Institute of Electrical and Electronics Engineers, Inc., Standard for Software Test Documentation, 1983

IEEE STD 830, Institute of Electrical and Electronics Engineers, Inc., Guide to Software Requirements Specification, 1984

IEEE STD 1012, Institute of Electrical and Electronics Engineers, Inc., Standard for Software Verification and Validation Plans, 1987

ISO 12207-1, International Standards Organization, Information Technology-Software, 1994

Society of Automotive Engineers, Aerospace Recommended Practice 4754: Certification Considerations for Highly Integrated or Complex Aircraft Systems, November 1996.

Society of Automotive Engineers, Aerospace Recommended Practice 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, December 1996.

B.3 Individual References

Bozarth, John D., Software Safety Requirement Derivation and Verification, Hazard Prevention, Q1, 1998

Bozarth, John D., The MK 53 Decoy Launching System: A "Hazard-Based" Analysis Success, Proceedings: Parari '99, Canberra, Australia

Brown, Michael, L., Software Systems Safety and Human Error, Proceedings: COMPASS 1988

Brown, Michael, L., What is Software Safety and Who's Fault Is It Anyway?, Proceedings: COMPASS 1987

Brown, Michael, L., Applications of Commercially Developed Software in Safety Critical Systems, Proceedings of Parari '99, November 1999

Card, D.N. and **Schultz**, D.J., Implementing a Software Safety Program, Proceedings: COMPASS 1987

Church, Richard, P., Proving A Safe Software System Using A Software Object Model, Proceedings:15th International System Safety Society Conference, 1997

Connolly, Brian, Software Safety Goal Verification Using Fault Tree Techniques: A Critically Ill Patient Monitor Example, Proceedings: COMPASS 1989

De Santo, Bob, A Methodology for Analyzing Avionics Software Safety, Proceedings: COMPASS 1988

Dunn, Robert and **Ullman,** Richard, Quality Assurance For Computer Software, McGraw Hill, 1982

Ericson, C.A., Anatomy of a Software Hazard, Briefing Slides, Boeing Computer Services, June 1983

Foley, Frank, History and Lessons Learned on the Northrop-Grumman B-2 Software Safety Program, Paper, Northrop-Grumman Military Aircraft Systems Division, 1996

Forrest, Maurice, and **McGoldrick,** Brendan, Realistic Attributes of Various Software Safety Methodologies, Proceedings: Ninth International System Safety Society, 1989

Gill, Janet A.,Safety Analysis of Heterogeneous-Multiprocessor Control System Software, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1990.

Hammer, William, R., Identifying Hazards In Weapon Systems – The Checklist Approach, Proceedings: Parari '97, Canberra, Australia

Kjos, Kathrin, Development of an Expert System for System Safety Analysis, Proceedings: Eighth International System Safety Conference, Volume II.

Lawrence, J.D., Design Factors for Safety-Critical Software, NUREG/CR-6294, Lawrence Livermore National Laboratory, November 1994

Lawrence, J.D., Survey of Industry Methods for Producing Highly Reliable Software, NUREG/CR-6278, Lawrence Livermore National Laboratory, November 1994.

Leveson, Nancy, G, SAFEWARE; System Safety and Computers, A Guide to Preventing Accidents and Losses Caused By Technology, Addison Wesley, 1995

Leveson, Nancy, G., Software Safety: Why, What, and How, Computing Surveys, Vol 18, No. 2, June 1986

Littlewood, Bev and **Strigini**, Lorenzo, The Risks of Software, Scientific American, November 1992

Mattern, S.F., Software Safety, Masters Thesis, Webster University, St. Louis, MO. 1988

Mattern, S.F. Capt., Defining Software Requirements for Safety-Critical Functions, Proceedings: Twelfth International System Safety Conference, 1994

Mills, Harland, D., Engineering Discipline For Software Procurement, Proceedings: COMPASS 1987

Moriarty, Brian and **Roland**, Harold, E., System Safety Engineering and Management, Second Edition, John Wiley & Sons, 1990

Russo, Leonard, Identification, Integration, and Tracking of Software System Safety Requirements, Proceedings: Twelfth International System Safety Conference, 1994

Unknown Author: Briefing on the Vertical Launch ASROC (VLA), Minutes, 2nd Software System Safety Working Group (SwSSWG), March 1984

B.4 Other References

DEF(AUST) 5679, Army Standardization (ASA), The Procurement Of Computer-Based Safety Critical Systems, May 1999

International Electrotechnical Commission, IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems, draft 61508-2 Ed 1.0., 1998

Document ID: CA38809-101, International Standards Survey and Comparison to Def(Aust) 5679, Issue: 1.1, Dated 12 May 1999

C Software Development Models

C.1 Software Development Models

Software Engineering uses several different lifecycle models to develop software. These models evolved over the years to meet different needs and expectations and to help cope with the increasing complexity of software used in modern systems. One generally accepted paradigm is that software engineering is not able to adequately develop software of the complexity of today's systems.

A software development process for a given system may use a single model or it may include several different models depending on the needs of the program. The software engineering team may choose the model or individual software developers may use a model they are experienced with to do the development.

Figure C-1 is a graphical representation of the relationship of the software development life cycle to the system/hardware development life cycle. The model is representative of the "Waterfall," or "Grand Design" life cycle. While this model is still used on numerous developments, other models are more representative of the current software development practices, such as the "Spiral", "Modified V", Rapid Prototyping, Object Oriented Analysis and Design, and Generative Programming techniques such as Aspect Oriented Programming.

An important consideration is that the software development lifecycle does not correlate exactly with the hardware or system development lifecycle. It often "lags" behind the hardware development at the beginning but may finish before the hardware development is complete. Another important consideration is that design reviews for hardware often lag behind those for software.

C.1.1 Grand Design, Waterfall Life Cycle Model[1]

The Waterfall software acquisition and development lifecycle model is one of the oldest "formal" models in use by software developers. This strategy "...was conceived during the early 1970s as a remedy to the *code-and-fix* method of software development." Grand Design places emphasis on up-front documentation during early development phases, but does not support modern development practices such as object oriented design, rapid prototyping, or automatic code generation. "With each activity as a prerequisite for succeeding activities, this strategy is a risky choice for unprecedented systems because it inhibits flexibility." Another limitation to the model is that after a single pass through the model, the system is complete. Therefore, identification of many integration problems occurs too late in the development process resulting in significant cost and schedule impacts. In terms of software safety, interface issues must be identified and rectified as early as possible in the development life cycle to allow for adequate correction and verification. Figure C-1 is a representation of the Grand Design, or Waterfall, life cycle model. The Waterfall model is not an effective development model for large, software-

[1] The following descriptions of the software acquisition life cycle models are either quoted or paraphrased from the Guidelines for Successful Acquisition and Management of Software Intensive Systems, Software Technology Support Center (STSC), September 1994, unless otherwise noted.

intensive, systems due to the limitations stated above and the inability to manage program risks during the software development process effectively. The Grand Design does, however, provide a structured and well-disciplined method for software development.

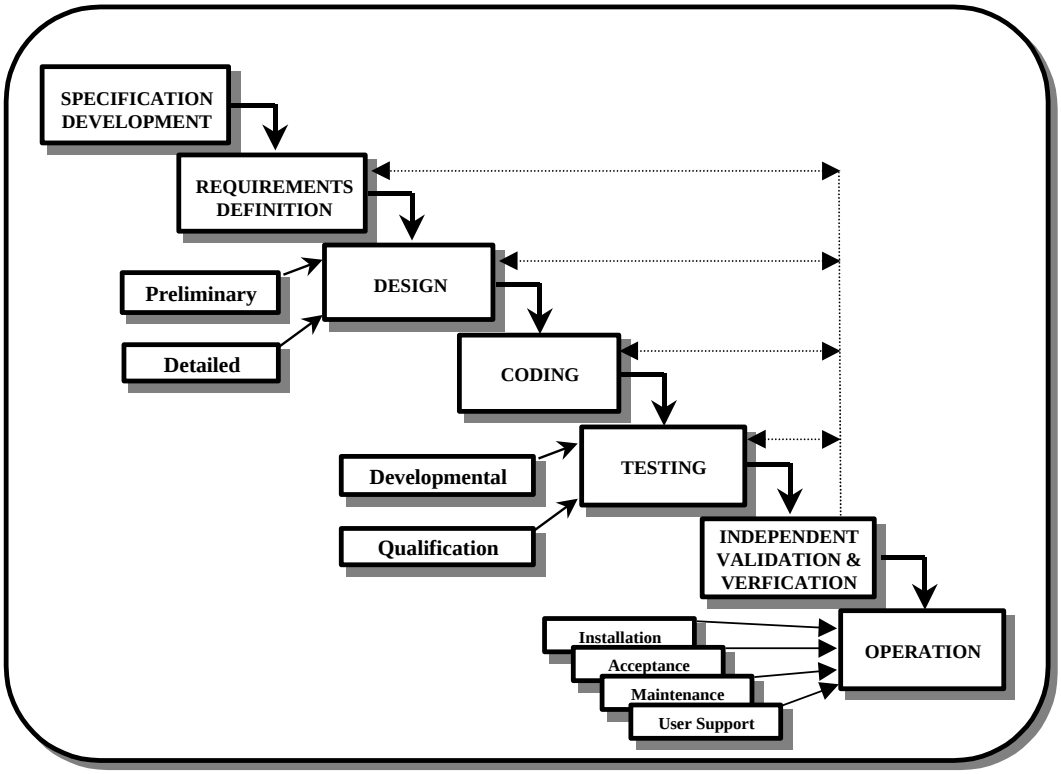


Figure C-1: Grand Design Waterfall Software Acquisition Life Cycle Model

C.1.2 Modified V Life Cycle Model

The Modified V software acquisition life cycle model depicted in Figure C-2 is another example of a defined method for software development. The model relies heavily on the ability to design, code, and test the software in increments of design maturity. "The left side of the figure identifies the specification, design, and coding activities for developing software. It also indicates when the test specification and test design activities can start. For example, the system/acceptance tests can be specified and designed as soon as software requirements are known. The integration tests can be specified and designed as soon as the software design structures are known. And, the unit tests can be specified and designed as soon as the code units are prepared."[2] The right side of the figure identifies when the evaluation activities occur that are involved with the execution and testing of the code at its various stages of evolution.

[2] Software Test Technologies Report, August 1994, STSC, Hill Air Force Base, UT 84056

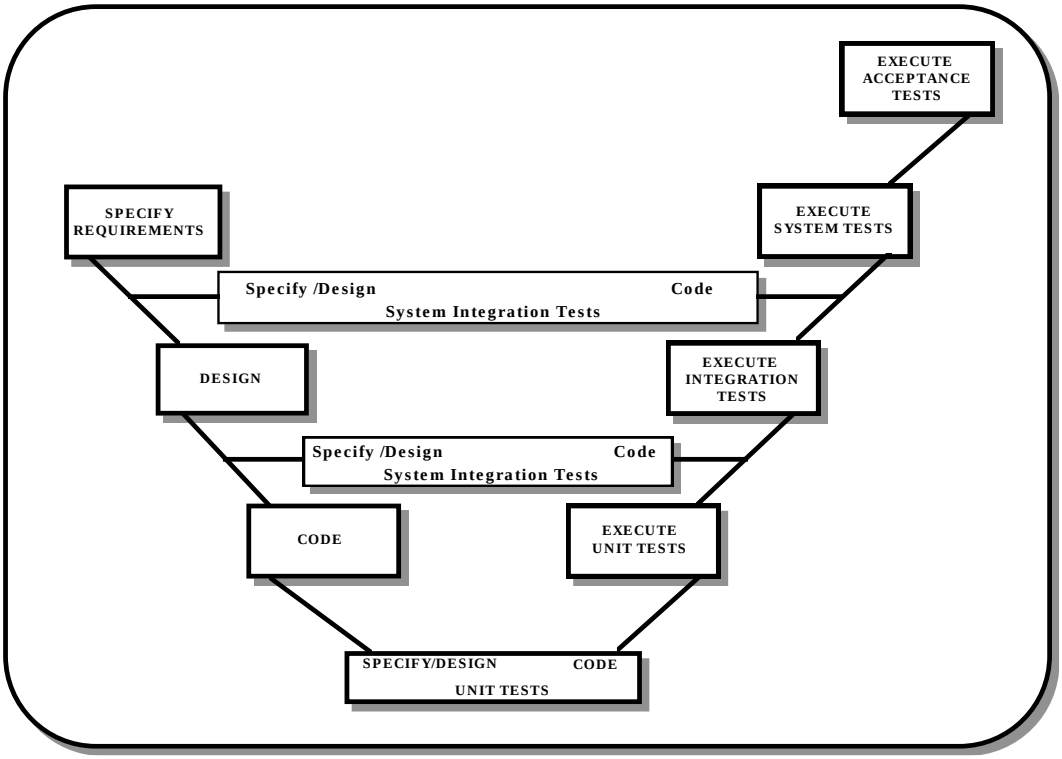


Figure C-2: Modified V Software Acquisition Life Cycle Model

C.1.3 Spiral Lifecycle Model

The Spiral development life cycle model provides a risk-reduction approach to the software development process. In the Spiral model, Figure C-3, the radial distance is a measure of effort expended, while the angular distance represents progress made. It combines features of the Waterfall and the incremental prototype approaches to software development. "Spiral development emphasizes evaluation of alternatives and risk assessment. These are addressed more thoroughly than with other strategies. A review at the end of each phase ensures commitment to the next phase or identifies the need to rework a phase if necessary. The advantages of Spiral development are its emphasis on procedures, such as risk analysis, and its adaptability to different development approaches. If Spiral development is employed with demonstrations and Baseline/Configuration Management (CM), you can get continuous user buy-in and a disciplined process."[3] From the system safety perspective, the spiral development model is ideal: it offers the opportunity to gradually develop, implement and verify safety requirement during each cycle.

[3] Guidelines for Successful Acquisition and Management of Software Intensive Systems, STSC, September 1994.

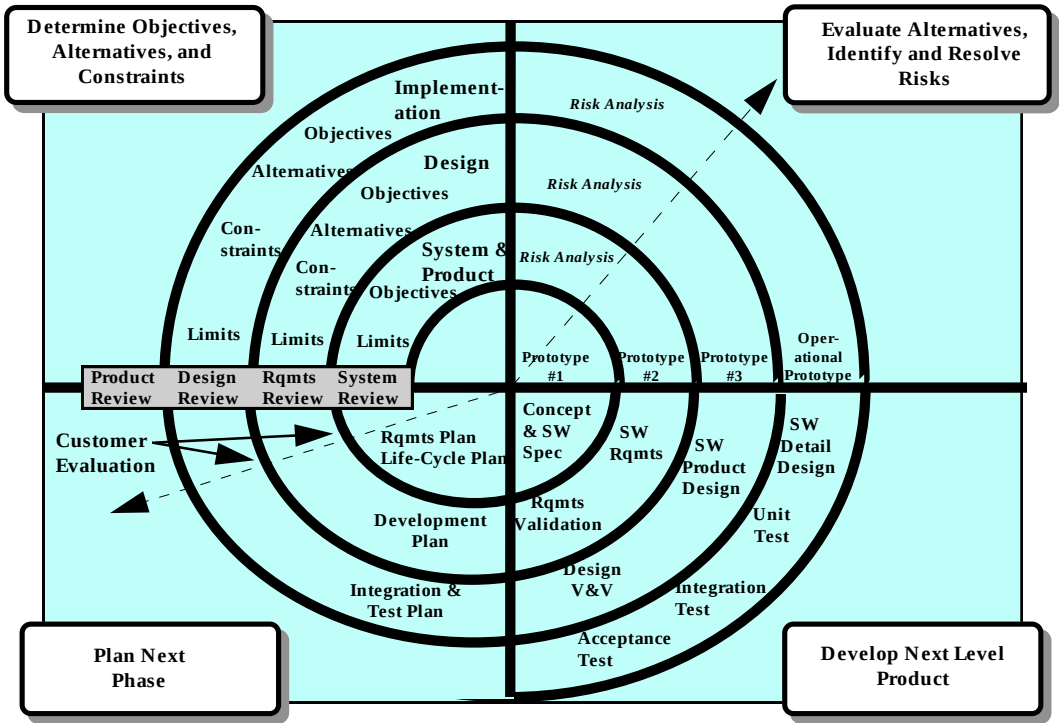


Figure C-3: Spiral Software Acquisition Life Cycle Model

C.1.4 Rapid Prototyping

Rapid prototyping involves the development of models of a system at high levels of abstraction and refining the models as development progresses. It is useful for develop large, complex system software, particularly software with significant user interaction. In such systems, developers frequently begin with prototypes of the user interface with rudimentary simulators to simulate system necessary functionality. After the user has an opportunity to test the interface and make comments, suggestions, and criticisms, the developers revise the interface and begin developing the functional software. Prototypes will undergo many revisions in the course of development and therefore, their design must support these changes.

Rapid prototyping may involve several teams who independently develop models of the system software. The teams or team leaders meet to discuss the models and determine which model or combination of models best exemplifies the desired capabilities of the system. A prototype must satisfy the system-level requirements however, the interpretation of those requirements may vary, especially if written in English language specifications. Through the independent study, teams often find that they've interpreted the high-level requirements differently. These meetings allow the teams to discuss the different interpretations and the rationale for those interpretations. This process often identifies missing or ambiguous requirements allowing for clarification from the procuring agency.

Rapid prototyping is not intended to produce robust software for production. Its primary purpose is to allow for reasoning about the system and its software and develop specifications that most accurately reflect the desired functional and non-functional capabilities of the system.

C.1.5 Object Oriented Analysis and Design

The classical software-development paradigms (e.g., waterfall and Grand V models) worked well for small to medium scale programs however, they were unable to scope up to the size of modern software products. Also, developers and users expected that these paradigms would reduce post-delivery maintenance, an expectation they never realized. One significant reason for the failure to meet expectations is that they are either operation oriented or data oriented but not both. The Object Oriented Analysis and Design (OOA&D) paradigm considers both attributes and operations to be equally important. An object is a unified software artifact that encompasses both attributes and operations. Safety requirements are typically specified as "contracts" on classes and subclasses of objects. These contracts translate into design requirements in the actual implementation. One of the negative features of OOA&D is the difficulty in testing the implementation of the safety design requirements. A key reason is that OOA&D minimizes the visibility into the internal functionality of the objects. Another negative feature of OOA&D is that the identification of classes and objects may make the integration of desirable safety features difficult if not impossible. Safety requirements are typically "cross-cutting" concerns in that they affect multiple objects (and their interactions). Classical OOA&D does not support the resolution of cross-cutting concerns very well.

C.1.6 Evolutionary Prototyping

Evolutionary prototyping requires the rapid realization and analysis of proposed system behavior. It blends the traditional waterfall model of requirements analysis, design, coding, and testing with rapid prototyping with requirements refinement occurring at each phase of development. As such, it eliminates many details of the system design and development from the designer's consideration during the early phases of development. The process uses an iterative approach to define requirements and design solutions, determining required interactions between the proposed system and its environment, identifies and determines constraints on the proposed system, and explores a range of possible solutions. Evolutionary prototyping provides executable models of the target systems, models hardware resources, software architecture, and supports evaluation of the system performance. This allows validation of specifications through prototype demonstrations. It also allows early identification of resource allocation and system performance issues early in the development. This reduces the system integration and overall testing effort.

C.1.7 Extreme Programming Model

The Extreme Programming (often called XP) model is also known as Pair-wise programming. In the XP model, the customer specifies a capability for the software system under development. The developer breaks the software system into component capabilities and parts and returns to the customer who chooses what features the initial delivery must provide. Programmers break the features down into stand-alone tasks and estimate the level of effort required to complete each task. The programmers work in pairs (hence the name "pair-wise") with one overlooking the efforts of the other. The programmers write unit tests, add features to modules to pass the unit tests, fix features and tests as necessary until all tests past. The programmers then integrate the code and conduct testing until they can release the code to the customer. The customer then runs acceptance tests on the code. Once the customer accepts the release, the developer releases a production version. Note that Extreme Programming is the least suitable methodology for developing safety-related software.

C.1.8 Generative Programming Techniques

Generative Programming techniques automate aspects of the software development process, primarily in the high-level design phase. The processes generally involve developing models from the system specifications which are refined to the point where tools can begin selecting reusable components to build the system software. Generative Programming techniques therefore rely on the availability of libraries of reusable components that the developers use to build the system.

C.2 Software Systems Safety and Software Development Models

The Software Systems Safety process remains relatively constant through many of the software development lifecycle models although there are variations required to accommodate fundamental differences in the models. The basic premise of the Software Systems Safety lifecycle process follows the Grand Design Model closely. However, it has many of the same difficulties associated with the Grand Design Model. More difficult models to apply the process for the same reasons are:

- Modified V Life Cycle Model
- Spiral Lifecycle Model
- Rapid Prototyping
- Object Oriented Analysis and Design
- Evolutionary Prototyping
- Extreme Programming Model
- Generative Programming Technique

D Guidance on System Design Requirements

Guidance within this Appendix refers to the Requirements set down in Chapter 4 where each paragraph relates to the same paragraph in this appendix providing the corresponding guidance. If these requirements and guidelines are properly implemented, they should reduce the risk of the computing system causing an unsafe condition, malfunction of a failsafe system, or non-operation of a safety function. These requirements and guidelines are not intended to be used as a checklist but, in conjunction with safety analyses performed in accordance with applicable standards and directives, they must be tailored to the system or system type under development. These requirements and guidelines must also be used in conjunction with accepted high quality software engineering practices including configuration control, reviews and audits, structured design, and related systems engineering practices.

D.1 General Principles

D.1.1 Two Person Rule

While this basic requirement is applicable to all systems, it is worth stressing the absolute necessity for at least two people to be totally familiar with the software components and their impact on safety

D.1.2 Program Patch Prohibition

The use of patches to 'fix' a problem may be acceptable in general software development, but must not be tolerated in the development of safety related software.

D.1.3 Designed Safe States

No further guidance.

D.1.4 Safe State Return

Any safety condition that may be overridden must be well documented, with appropriate warnings clearly annotated. It would be advisable that any safety violation is logged in order that any retrospective safety implications may be further analysed.

D.1.5 Circumvent Unsafe Conditions

Guidance: As for Safe State Return.

D.1.6 External Hardware Failures:

No further guidance.

D.1.7 Safety Kernel Failure:

No further guidance.

D.1.8 Fallback and Recovery

A common design idiom for critical software systems is that they are "self checking and self protecting." This means that software components "protect" themselves from invalid requests or

invalid input data by frequently checking for violations of assumptions or constraints. In addition, they check the results of service requests to other system components to make sure that they are behaving as expected. Finally, such systems typically provide for the checking of internal intermediate states to determine if the routine is itself working as expected. Violations of any of these kinds of checks can require transition to a safe state if the failure is serious or if the confidence in further correct execution has been seriously reduced. Failure to address this "defensive" approach can allow a wide variety of failures to propagate throughout the system in unexpected and unpredictable ways, potentially resulting in a hazard.

- Does the developer identify a distinct safe mode or set of safe modes? Has the analysis of these safe modes adequately considered the transition to these safe modes from potentially hazardous states (e.g., internal inconsistency)?
- Does the design include acceptable safety provisions upon detection of an unsafe state?
- Does the design include assertion checks or other mechanisms for the regular run-time calibration of internal logic consistency?
- Does the developer provide for an orderly system shutdown as a result of operator shutdown instructions, power failure, etc.?
- Does the developer explicitly define the protocols for any interactions between the system and the operational environment? If anything other than the expected sequences or interlocks is encountered, does the system design detect this and transition to a safe state?
- Does the developer account for all power-up self-test and handshaking with other components in the operational environment in order to ensure execution begins in a predicted and safe state?

D.1.9 Computing System Failure

No further guidance.

D.1.10 Maintenance Interlocks

Upon completion of tests and/or training wherein safety interlocks are removed, disabled or bypassed, restoration of those interlocks should all be verified by the software prior to being able to resume normal operation. While overridden, a display should be made on the operator's or test conductor's console of the status of the interlocks, if applicable.

D.1.11 Interlock Restoration

No further guidance.

D.1.12 Simulators

No further guidance.

D.1.13 Logging Safety Errors

No further guidance.

D.1.14 Positive Feedback Mechanisms:

Such feedback should be analysed to ensure that any failure to operate a safety function is investigated. Where possible, such feedback should also be included in the design.

D.1.15 Peak Load Conditions

Where possible the system should be fully tested under peak load conditions to confirm this requirement has been satisfied.

D.1.16 Ease of Maintenance

It should be remembered that personnel that are not associated with the original design team will carry out the maintenance. Good documentation is essential and should be developed with maintenance of the software in mind. Strict configuration control of the software during development and after deployment is required. The use of techniques for the decomposition of the software system for ease of maintenance is recommended.

D.1.17 Endurance Issues

Although software does not "wear out," the context in which a program executes can degrade with time. Systems that are expected to operate continuously are subjected to demands for endurance - the ability to execute for the required period of time without failure. As an example of this, the failure of a Patriot missile battery in Dhahran during the Persian Gulf War was traced to the continuous execution of tracking and guidance software for over 100 hours; the system was designed and tested against a 24-hour upper limit for continuous operation. Long-duration programs are exposed to a number of performance and reliability problems that are not always obvious and that are difficult to expose through testing. This makes a careful analysis of potential endurance-related defects an important risk-reduction activity for software to be used in continuous operation.

D.1.17.1 Identification of duration requirements

The following questions should be considered:

- Has the developer explicitly identified the duration requirements for the system? Has the developer analyzed the behavior of the design and implementation if these duration assumptions are violated? Are any of these violations a potential hazard?
- Has the developer identified potential exposure to the exhaustion of finite resources over time, and are adequate detection and recovery mechanisms in place to handle these? Examples are as follows:
    - Memory (e.g., heap leaks from incomplete software storage reclamation)
    - File handles, Transmission Control Protocol ports, etc. (e.g., if not closed under error conditions)
    - Counter overflow (e.g., 8-bit counter and > 255 events was a factor in the failure of Theriac-25 radiation treatment machines).

D.1.17.2 **Performance Degradation**

The following should be considered:

- Has the developer identified potential exposure to performance degradation over time, and are adequate deduction and recovery mechanisms in place to handle these? Examples are memory and disk fragmentation that can result in increased latency.
- Has the developer analyzed increased exposure to cumulative effects over time, and are adequate detection and recovery mechanisms in place to handle these so that they do not present any hazards? Examples include cumulative drift in clocks, cumulative jitter in scheduling operations, and cumulative rounding error in floating point and fixed-point operations.

D.1.18 **Error Handling**

Causal analyses of software defects frequently identify error handling as a problem area. For example, one industry study observed that a common defect encountered was "failure to consider all error conditions or error paths." A published case study of a fault tolerant switching system indicated that approximately two thirds of the system failures that were traceable to design faults were due to faults in the portion of the system that was responsible for detecting and responding to error conditions. The results of a Missile Test and Readiness Equipment (MlTRE) internal research project on Error Handling in Large Software Systems also indicate that error handling is a problematic issue for many software systems. In many cases, the problems exposed were the result of oversight or simple logic errors. A key point is that these kinds of errors have been encountered in some software that is far along in the development process and/or under careful scrutiny because it is mission critical software. The presence of simple logic errors such as these illustrates the fact that error handling is typically not as carefully inspected and tested as other aspects of system design. It is important that the program office gain adequate insight into the developer's treatment of error handling in critical systems. The basic considerations are:

- Has the developer clearly identified an overall policy for error handling?
- Have the specific error detection and recovery situations been adequately analyzed?
- Has the developer defined their relationship between exceptions, faults, and "unexpected" results?

These considerations lead to a number of more technical issues, such as:

- Are different mechanisms used to convey this status of computations? What are they? [e.g., Ada exceptions, OS signals, return codes, messages]. If return codes and exceptions are both used, are there guidelines for when each is to be used? What are these guidelines and the rationale for them, and how are they enforced? Are return codes and exceptions used in distinct "layers of abstraction" (e.g., return codes only in calls to COTS OS services) or freely intermixed throughout the application? How are return codes and exceptions mapped to each other? In this mapping, what is done if an unexpected return code is returned, or an unexpected exception is encountered?
- Has the developer determined the costs of using exceptions for their compiler(s)? What is the space and runtime overhead of having one or more exception handlers in a

subprogram and a block statement, and is the overhead fixed or a function of the number of handlers? How expensive is propagation, both explicit and implicit?

- Are derived types used? If so, are there any guidelines regarding the exceptions that can be raised by the derived operations associated with the derived types? How are they enforced?
- Are there guidelines regarding exceptions that can be propagated during task rendezvous? How are they reinforced and tested?
- Is program suppression ever used? If so, what are the restrictions on its use, and how are they enforced? What is the rationale for using/not-using program suppression? If it is used, are there any guidelines for explicit checking that must be in the code for critical constraints in lieu of the implicit constraint checks? If not, how is the reliability of the code ensured?
- Are there any restrictions on the use of tasks in declarative regions of subprograms (i.e., subprograms with dependent tasks)? If so, how are they enforced? How are dependent tasks terminated when the master subprogram is terminating with an exception, and how is the suspense of exception propagation until dependent task termination handled?
- What process enforcement mechanisms are used to ensure global consistency among error handling components? (e.g., we have seen examples of systems where various subcontractors were under constrained; they each make locally plausible design decisions regarding error handling policy, but when these components were integrated they were discovered to be globally inconsistent.)
- Are there guidelines on when exceptions are masked (i.e., a handler for an exception does not in turn propagate an exception), mapped (i.e., a handler for an exception propagates a different exception), or propagated? If so, how are they enforced? Are there any restrictions on the use of the "others" handlers? If so, how are they enforced?
- How does the developer ensure that return codes or status parameters are checked after every subroutine call, or ensure that failure to check them does not present a hazard?

Are there any restrictions on the use of exceptions during elaboration? (e.g., checking data passed to a generic package during installation). Is exception handling during elaboration a possibility due to initialization functions in declarative regions? If so, how is this handling tested, and are there design guidelines for exception handling during elaboration? If not, how are they assured that this does not present a hazard?

D.1.19 Standalone Processors

It is important that the opportunity for non-safety related software components to react with safety related components is minimised. There should be partitioning to separate the different functions, which will ideally involve employing separate processors, protected memory etc.

The practical limits on resources for critical software assurance are consistent with the consensus in the software development community that a major design goal for critical software is to keep the critical portions small and isolated from the rest of the system. The program office can evaluate evidence provided by the developer that indicates the extent to which this isolation has

been a design goal and the extent to which the implementation has successfully realized this goal. Confidence that unanticipated events or latent defects in the rest of the software will not introduce an operational hazard is in part correlated with the confidence that such isolation has been achieved.

- Does the developer's design provide explicit evidence of an analysis of the criticality of the components and functions (i.e., does the design reflect an analysis of which functions can introduce a hazard)?
- Does the developer's design and implementation provide evidence that in critical portions of the software, coupling has been kept to a minimum (e.g., are there restrictions on shared variables and side-effects for procedures and functions)?
- Does the developer's design include attention to the implementation of "firewalls" in the software - boundaries where propagation of erroneous values is explicitly checked and contained? Do critical portions of code perform consistency checking of data values provided to them both by "clients" (i.e., software using the critical software as a service) and by the software services the critical software calls (e.g., OS services)?
- Does the critical software design and implementation include explicit checks of intermediate states during computation, in order to detect possible corruption of the computing environment (e.g., range checking for an intermediate product in an algorithm)?
- Does the developer provide the criteria for determining what software is critical, and is there evidence that these criteria were applied to the entire software system? How does the developer provide evidence that the portions considered non-critical in fact will not introduce a hazard?

D.1.20 Input/output Registers

Whenever it is impossible to segregate safety and non-safety functions, as discussed above, the rigor associated with the design and development should be commensurate with the highest safety level determined.

D.1.21 Power-Up Initialization

The requirements apply to the design of the power subsystem, power control, and power-on initialization for safety-related applications of computing systems.

D.1.22 Power-Down Transition

This is a key area, which could easily be overlooked in the test programme.

D.1.23 Power Faults

No further guidance.

D.1.24 System-Level Check

No further guidance.

D.1.25 Redundancy Management

In order to reduce the vulnerability of a software system to a single mechanical or logic failure, redundancy is frequently employed. However, the added complexity of managing the redundancy in fault-tolerant systems may make them vulnerable to additional failure modes that must be accounted for by the developer. For example, the first shuttle flight and the 44th flight of NASA's Advanced Fighter Technology Integration (AFTI)-F16 software both exhibited problems associated with redundancy management. The first shuttle flight was stopped 20 minutes before scheduled launch because of a race condition between the two versions of the software. The AFTI-F16 had problems related to sensor skew and control law gain causing the system to fail when each channel declared the others had failed; the analog backup was not selected, because the simultaneous failure of two channels was not anticipated.

If the developer's design includes redundancy (e.g., duplicate independent hardware, or "N version programming"), have the additional potential failure modes from the redundancy scheme been identified and mitigated? Examples include sensor skew, multiple inconsistent states, and common mode failures.

D.2 Computing System Environment Requirements and Guidelines

Good practice should be laid down in the development standards. This should include compliance with all limits specified for the overall size and complexity of the software and for the size and complexity of its constituent parts; Non-functional properties, such as fault-tolerance, resource management and timing should be implemented in a consistent way throughout the design process. The method of implementation should be defined as a policy in the development standards. These should address how architectural features such as interrupts and scheduling, memory management, global data, defensive coding and exception handling are implemented.

Design methods

Software design methods comprise a disciplined approach that facilitates the partitioning of a software design into a number of manageable stages through decomposition of data and function. They are usually tool supported. Design methods include object-oriented methods and older, structured design methods.

A range of design methods and variations exist, with some general purpose and some intended for specific application areas, for example real-time, process control, data-processing and transaction processing. Many are graphical in format, providing useful visibility of the structure of a specification or design, and hence facilitating visual checking.

Many design methods possess their own supporting notation and rules that in some cases may make use of mathematical notations thereby providing scope for automatic processing. Automatic code generation is also possible with some (tool supported) methods.

There is controversy about the use of object-oriented methods in high integrity software. The use of object-oriented methods implies that object-oriented languages will be used to implement the

design and raises issues of insecurities. On the positive side object oriented methods facilitate encapsulation, modularity and information hiding, all of which can contribute to better designs.

Alternatively, object oriented methods may be used for design, but with restrictions enabling an object based approach for implementation. An object based approach would typically be consistent with safe language subsets (e.g. of Ada).

D.3 Coding and Coding Standards

The coding standards should define rules that lead to clear, unambiguous source code that is easy to review, test and maintain, amenable to static analysis (where used) and whose traceability to software design is clear.

A coding standard should be defined as part of the development standard and justification for the choice of language and coding standard features should be provided.

The implementation language and coding standards used should conform to the development standards. Justification should be provided to show that the code is not affected by any known errors in the compilation system or target hardware.

To be effective, a coding standard must be enforced. It is easier to enforce a coding standard if the coding standard requirements are checked by a tool. It is not always possible to have a tool that checks all coding standard requirements however, so a subset of the requirements may need to be checked by review (e.g. naming conventions, explanatory comments and layout). A checklist may be useful where this is the case.

D.3.1 Modular Code

No further guidance.

D.3.2 Number of Modules

No further guidance.

D.3.3 Size of Modules

No further guidance.

D.3.4 Execution Path

No further guidance.

D.3.5 Halt Instructions

No further guidance.

D.3.6 Single Purpose Files

No further guidance.

D.3.7 Unnecessary Features

No further guidance.

D.3.8 Indirect Addressing Modes

No further guidance.

D.3.9 Uninterruptible Code

No further guidance.

D.3.10 Safety Related Files

No further guidance.

D.3.11 Unused Memory

It should not be filled with random numbers, halt, stop, wait, or no-operation instructions. Data or code from previous overlays or loads must not be allowed to remain. (Examples: If the processor architecture halts upon receipt of non-executable code, a watchdog timer shall be provided with an interrupt routine to revert the system to a safe state. If the processor flags non-executable code as an error, an error handling routine must be developed to revert the system to a safe state and terminate processing.) Information must be provided to the operator to alert him to the failure or fault observed and to inform him of the resultant safe state to which the system was reverted.

D.3.12 Overlays of Safety Related Software

Where less memory is required for a particular function, the remainder must be filled with a pattern that will cause the system to revert to a safe state if executed. It must not be filled with random numbers, halt, stop, no-op, or wait instructions or data or code from previous overlays.

D.3.13 Operating System Functions

No further guidance.

D.3.14 Flags and Variables

No further guidance.

D.3.15 Loop Entry Point

No further guidance.

D.3.16 Critical Variable Identification

No further guidance.

D.3.17 Variable Declaration

Care should be taken to use meaningful variable names. Duplication of variable name should be avoided, even though their scope may be independent.

D.3.18 Global Variables

No further guidance.

D.3.19 Unused Executable Code

No further guidance.

D.3.20 Unreferenced Variables

No further guidance.

D.3.21 Data Partitioning

No further guidance.

D.3.22 Conditional Statements

There should be no potentially unresolved input to the conditional statement.

Conditional statements must be analyzed to ensure that the conditions are reasonable for the task and that all potential conditions are satisfied and not left to a default condition. All condition statements must be annotated with their purpose and expected outcome for given conditions.

D.3.23 Strong Data Typing

Safety-related functions must not employ a logic "1" and "0" to denote the safe and armed (potentially hazardous) states. The armed and safe state for munitions must be represented by at least a unique, four-bit pattern. The safe state must be a pattern that cannot, as a result of a one-, two-, or three-bit error, represent the armed pattern. The armed pattern must also not be the inverse of the safe pattern. If a pattern other than these two unique codes is detected, the software must flag the error, revert to a safe state, and notify the operator, if appropriate.

D.3.24 Annotation of Timer Values

Comments shall include a description of the timer function, its value and the rationale or a reference to the documentation explaining the rationale for the timer value. These values shall be verified and shall be examined for reasonableness for the intended function.

D.4 Selection of Languages

D.4.1 High-level language requirement.

There are a few exceptions to the use of high level languages i.e. for very small units of software (or logic) where the use of assembler or a simple logic device avoids the complexity of assuring a compilation system. For safety related systems, the full set of features of a standard language is often inappropriate and a subset should be defined.

D.4.2 Use of assembler

Experience shows that programming in assembler language is more error-prone than programming in a high-level language. There are circumstances, however, in which the practicalities of real time operation are such that the use of assembler is necessary in order to meet performance requirements. Furthermore, the use of a high-level language introduces some

additional risks compared with the use of assembler. Specifically, high-level language compilation systems are more prone to error and less easy to verify than assemblers. There is therefore a need to trade off the risks of assembler programming against the risks of the use of a high-level language compiler, linker and run-time system.

In general it is considered that only for very small amounts of assembler do the safety benefits of not having a compilation system outweigh the increased risk of programming error. Examples where the use of assembler is justified include:

Sections of the software where close interaction with hardware is required that cannot easily be accommodated with a high-level language.

Sections of the software where performance constraints cannot be met by a high-level language.

Very small applications where the use of a high-level language, compilation system and more powerful processor would increase, rather than reduce, safety risk.

If programming in assembler, it is still important to have coding standards. There should be standards for block structuring, naming identifiers, layout and comments. Even with assembler, there may be features of the assembly language that should not be used (e.g. self modifying code!). Programming rules for assembler language programming should be more strict than those for high-level languages:

A possible alternative to small units of software programmed in assembler is the use of custom hardware, firmware and logic devices. These may be particularly appropriate for simple interlocks. These types of device may have features of both software development and hardware design. A technology should be selected that is robust enough for the environment (e.g. thermal range, EMC, shock resistance) and reliable. The logic may need to be developed and assured like a software process (particularly if the implementation is via a software like language such as VHDL and is compiled onto the hardware). If programmed in a similar way to software, coding standards should be used.

D.4.3 Characteristics and Selection of Languages

The syntax of a language is its vocabulary and grammar rules. A formally-defined syntax is one that is represented in an appropriate formal or mathematical language. A suitable formal language for representing programming language syntax is Backus-Naur Form (BNF). The semantics of the language are its meaning. The semantics define what the compiler will do when it translates the syntax into an executable form. In order for program execution to be predictable, the semantics of the language need to be well defined, either formally or informally.

Definitions of the language characteristic terms ‘strongly typed’ and ‘block structured’ are provided in Appendix A of this guidance.

D.4.4 Language Sub-Sets

Currently most commercially supported languages have features which are undefined, poorly defined or implementation-defined. Furthermore, most languages have constructs that are difficult or impossible to analyse. There are however, some widely accepted safe subsets of languages including Ada and C. Coding in a subset designed for use in safety related systems ensures that the program execution will be both predictable and verifiable.

Both static and dynamic checks are required to ensure that the requirements of the sub-et of the language hold. Syntax checks should be performed by the compiler (e.g. remove code that causes the compiler to generate warnings) and by additional syntax tools. Unfortunately it is rarely possible to define a subset that can be enforced only by syntax checks. Semantic checking of the subset requires more sophisticated static analysis tools. Some checking may need to be undertaken by reviewers. Checking of the dynamic properties may be possible by sophisticated static analysis tools (a few can show absence of run-time errors), otherwise they should be performed by analysis tools that operate during testing.

The safety claims for the software should include justification that the language checks performed by the compiler and by other means are adequate to ensure predictable program execution and that the code is unambiguous for human readers including reviewers, testers, analysts and future maintainers.

D.4.5 Object oriented languages

An object based approach uses objects in the structure of the programme. An object ***encapsulates*** data and operations together. Object based languages provide mechanisms so that objects have a public interface that provides services to other objects and the detail of how the object achieves these services is hidden within the object (***information hiding***). Many of the subsets of object oriented languages (e.g. of Ada) are object based. Object based languages have some, but not all of the advantages of object oriented languages in terms of productivity. Encapsulation and information hiding generally aid good program structure, limit the propagation of errors and assist in making the software easy to understand.

Inheritance permits one object to be defined from the properties of another. A general purpose object could be defined (e.g. I/O device) and then several different more specific objects (e.g. USB port, parallel port) could be defined by inheritance from the general purpose object. The specific objects are ***instantiations*** of the general object. Object based languages typically permit inheritance in a limited way.

Fully object oriented languages have all of the features of object based languages, but go further by permitting late or ***dynamic binding***. With dynamic binding, the instantiation and interfaces between objects are not statically determinable, because the decisions are left until run-time. Traditional approaches to the definition of a safe-subset are difficult to apply to object oriented languages because of this lack of static predictability.

To mitigate this, if an object oriented language is used, the design should impose strict control on interfaces between objects. Formal specification of contracts (including pre-conditions, post-conditions and invariants) should be considered. For object oriented languages, additional justification should be provided that development standards have controlled within safe bounds:

- Memory usage (e.g. by appropriate controls on dynamic allocation);
- Predictable execution (e.g. by removing reliance on garbage collection);
- Predictable implementation of dynamically defined aspects (e.g. by controls on inheritance and dynamic binding).

D.4.6 Language Issues

D.4.6.1 Ada

The Ada programming language provides considerable support for preventing many causes of unpredictable behavior allowed in other languages. For example, unless pragma Suppress or unchecked conversion (and certain situations with pragma Interface) are used, implicit constraint checks prevent the classic "C" programming bug of writing a value into the 11th element of a 10-element array (thus overwriting and corrupting an undetermined region of memory, with unknown results that can be catastrophic). However, the Ada language definition identifies specific rules to be obeyed by Ada programs but which no compile-time or run-time check is required to enforce. If a program violates one of these rules, the program is said to be erroneous. According to the language definition, the results of executing an erroneous program are undefined and unpredictable. For example, there is no requirement for a compiler to detect the reading of uninitialized variables or for this error to be detected at run-time. If a program does execute such a use of uninitialized variables, the effects are undefined: the program might raise an exception (e.g., Program_Error, Constraint_Error), or simply halt, or some random value may be found in the variable, or the compiler may have a pre-defined value for references to uninitialized variables (e.g., 0). For obvious reasons, the overall confidence that the program office has in the predictable behavior of the software will be seriously undermined if there are shown to be instances of "erroneous" Ada programs for which no evidence is provided that they do not present a hazard. There are several other aspects of the use of Ada that can introduce unpredictable behavior, timing, or resource usage, while not strictly erroneous.

- Are all constraints static? If not, how are the following sources of unpredictable behavior shown to prevent a hazard: Constraint_Error raised?
- Use of unpredictable memory due to elaboration of non-static declarative items
- For Ada floating point values, are the relational operators "<", '>", ""=, and "/=" precluded? Because of the way floating point comparisons are defined in Ada the values of the listed operators depend on the implementation. "<=" and ">=" do not depend on the implementation, however. Note that for Ada floating point it is not guaranteed that, for example, "X <= Y" is the same as "not (X>Y)". How are floating point, operations ensured to be predictable or how is the lack of predictability shown to not represent a hazard by the developer?
- Does the developer use address clauses? If so, what restrictions are enforced on the address clauses to prevent attempts to the overlay of data, which results in an erroneous program?
- If Ada access types are used, has the developer identified all potential problems that can result with access types (unpredictable memory use, erroneous programs if Unchecked_Deallocation is used and there are references to a deallocated object, aliasing, unpredictable timing for allocation, constraint checks) and provided evidence that these do not represent hazards?
- If pragma Interface is used, does the developer ensure that no assumptions about data values are violated in the foreign language code that might not be detected upon returning to the Ada code (e.g., passing a variable address to a C routine that violates a range

constraint - this may not be detected upon return to Ada code, enabling the error to propagate before detection)?

- Does the developer ensure that all out and in out mode parameters are set before returning from a procedure or entry call unless an exception is propagated, or provide evidence that there is no case where returning with an unset parameter (and therefore creating an erroneous program) could introduce a hazard?
- Since Ada supports recursion, has the developer identified restrictions on the use of recursion or otherwise presented evidence that recursion will not introduce a hazard (e.g., through exhaustion of the stack, or unpredictable storage timing behavior)?
- Are any steps taken to prevent the accidental reading of an uninitialized variable in the program [through coding standards (defect prevention) and code review or static analysis (defect removal)]? Does the developer know what the selected compiler's behavior is when uninitialized variables are referenced? Has the developer provided evidence that there are no instances of reading uninitialized variables that introduce a hazard, as such a reference results in an erroneous program?
- If the pre-defined Ada generic function Unchecked_Conversion is used, does the developer ensure that such conversions do not violate constraints of objects of the result type, as such a conversion results in an erroneous program?
- In Ada, certain record types and private types have discriminants whose values distinguish alternative forms of values of one of these types. Certain assignments and parameter bindings for discriminants result in an erroneous program. If the developer uses discriminants, how does he ensure that such erroneous uses do not present a hazard?

D.4.7 Compilers

Depending on the integrity requirements of the software (i.e. for lower integrity systems), it may be possible to argue that the correct operation of the compiler is demonstrated by the testing of the compiled code. For higher integrity systems (e.g. safety critical software), specific object code verification will be necessary

It is important that the operation of the compiler is repeatable, otherwise, the code that is tested may be different to the code that is put into service, even if there has been no change to the source code. It may be necessary to have different settings during testing compared with settings used for compiling the operational code (e.g. if debug facilities are needed during testing). If this is the case, then tests should be rerun using code compiled with the operational settings. Some compilers change their optimisation strategies depending on the memory available during compilation. If this is the case, and it is not possible to control the compiler through switches, it may be necessary to have a standard, consistent environment purely for compilation, to ensure a repeatable result.

Conditional compilation may be used e.g. to allow for variants of the software and to distinguish additional features as part of software for use on test rigs from operational software. Conditional compilation has the advantage that unnecessary executable code is not actually present in the operational software and therefore can not be accidentally triggered. Conditional compilation can be used as a defence against any failures in configuration management that allows test code

to be included in operational builds (e.g. by including conditionally compiled code that produces compilation errors if the compilation switches are set for operational code).

If conditional compilation is used the coding standards should require that conditionally compiled code is immediately obvious to reviewers and programmers.

Where the software is required to be of the highest integrity, there will be a requirement for high integrity performance from the compilation system. As compilers are not developed specifically for safety critical applications, in practice a combination of design, compiler assurance and object code verification should be used to maximize confidence in the correctness of the compilation process.

Methods of providing assurance of the correctness of a compiler include:

- Compiler validation: An internationally agreed black box method of testing compilers, designed to demonstrate that a compiler conforms to the appropriate international language standard. A validation certificate indicates that a compiler has successfully passed all the tests in the appropriate validation suite. This is not the same as a assurance that the compiler will always compile the safe subset correctly.
- Evaluation and testing: The conduct of evaluation and testing beyond that provided by validation. Such evaluation and testing may examine the compiler in more depth than validation and is also able to assess the compiler in the particular configuration in which it is to be used on the safety related software, which may not be the same configuration in which the compiler was validated. Such evaluations may be available from a third party.
- Experience in use. For lower integrity systems, the correctness of the compilation system may be demonstrated by the evidence that the executable code passed its tests.
- Object Code Verification. For high integrity systems, object code verification should be carried out. The process of object code verification will provide evidence for the correctness of the compilation system.
- Previous use: Evidence provided by successful previous use can be used as an argument for increasing the confidence in the correctness of the compiler. The requirements for configuration management, problem reporting and usage environments should be the same as for previously developed software.

The use of many programming languages will involve the use of an existing run-time system. This should be considered as use of previously developed software.

D.4.8 Automated and Tools assisted processes

The evaluation of automated tools should include:

- The role of the tool in assuring the safety of the software
- Whether the tool could introduce a safety significant fault
- Whether the tool could fail to detect a safety significant fault
- How failures of the tool could be detected and corrected by human supervision and by other tools and processes

The requirements for each tool should be documented and in addition to functionality and specific integrity properties, these should address:

- Usability
- Interoperability
- Stability
- Commercial availability
- Maintenance support
- Familiarity to software safety personnel

D.4.9 Selection of Tools

Tools are essential in an industrial scale project as they allow processes to be performed which otherwise would be impracticable or inefficient and they allow the developer to have a higher degree of confidence in the correctness of a process than might otherwise be the case. In addition to any tools that are needed for the specific methods to be used, tools to support common functions such as configuration management, checking of specification and design documents, static analysis, dynamic testing and subsequent manipulation of object code may be required.

Tools need to be assessed to ensure that they have sufficient safety assurance to ensure that they do not jeopardise the safety integrity of the software. The safety assurance requirements for the tools used to support the various functions of the development process are dependent on the contribution of the tools to the achievement of a safe product. In the same way that software should be considered in a systems context, the use of the tools should be considered in the context of the overall development process. The safety assurance requirements for a particular tool in a particular application can be deduced by undertaking a risk based assessment of the development process. The risk based assessment of the development process should consider each tool in the context of its use in the software development.

In many cases, the risk based assessment of the process may show that there are adequate safeguards within the process (e.g. potential faults introduced by one tool, will be detected by another). Alternatively, the analysis may show that the integrity of a tool can be adequately demonstrated by its use within the project. For example, where a tool is extensively used on a project and no problems found with the tool, such as in the case of a test tool that correctly detects test failures.

Commercial pressures mean that few tools, if any, are developed to meet high integrity requirements, and hence diverse checks within the development lifecycle are almost always be required to reduce the safety assurance requirements for the tools to a manageable level. The documentation should record the assessment of each tool to determine conformance with the required safety assurance requirements for the tool's proposed use in the development process.

Where the safety analysis and tool evaluation indicates that a single tool does not have the required level of safety assurance, combinations of tools or additional safeguards, such as diverse checks and reviews, should be used. The independence and diversity of the methods or tools and

their effectiveness in achieving the required level of safety assurance should be addressed in the safety analysis of the development process.

The Software Designer/developer has the overall responsibility for selecting tools and should address the matching of tools to the experience of members of the Design Team as this has an important effect on productivity and quality, and hence on safety assurance. It should be appreciated that compromises are inevitable in assembling a coherent tool set from a limited set of candidates. As part of the determination of adequate safety assurance, consideration should also be given to the interaction between the Design Team members and the tool, and the level of safety assurance that can be given to this interface.

The needs of support and maintenance activities during the in-service phase should be taken into account when selecting tools In order to minimize support requirements, consideration should be given to the selection of a limited number of broad spectrum tools in preference to the selection of a larger number of specialized tools. Tools which are unlikely to have continued support through the life of the equipment should not be employed in the software development.

In general, tools should be used as follows:

- Whenever practicable a tool of the required level of safety assurance should be used.
- Where it is not practicable to use a tool of the required level of safety assurance, a combination of tools should be used to provide the necessary added safety assurance.
- Where it is not practicable to use a combination of tools, the use of a tool should be combined with an appropriate manual activity
- Only if there are no suitable tools available should a process be performed manually

E Lessons Learned

E.1 Therac® Radiation Therapy Machine Fatalities

E.1.1 Summary

Eleven Therac-25 therapy machines were installed, five in the US and six in Canada. The Canadian Crown (government owned) company Atomic Energy of Canada Limited (AECL) manufactured them. The -25 model was an advanced model over earlier models (-6 and -20 models, corresponding to energy delivery capacity) with more energy and automation features. Although all models had some software control, the -25 model had many new features and had replaced most of the hardware interlocks with software versions. There was no record of any malfunctions resulting in patient injury from any of the earlier model Theracs (earlier than the -25). The software control was implemented in a DEC model PDP 11 processor using a custom executive and assembly language. A single programmer implemented virtually all of the software. He had an uncertain level of formal education and produced very little, if any documentation on the software.

Between June 1985 and January 1987 there were six known accidents involving massive radiation overdoses by the Therac-25; three of the six resulted in fatalities. The company did not respond effectively to early reports citing the belief that the software could not be a source of failure. Records show that software was deliberately left out of an otherwise thorough safety analysis performed in 1983, which used fault-tree methods. Software was excluded because "software errors have been eliminated because of extensive simulation and field testing. (Also) software does not degrade due to wear, fatigue or reproduction process." Other types of software failures were assigned very low failure rates with no apparent justification. After a large number of lawsuits and extensive negative publicity, the company decided to withdraw from the medical instrument business and concentrate on its main business of nuclear reactor control systems.

The accidents were due to many design deficiencies involving a combination of software design defects and system operational interaction errors. There were no apparent review mechanisms for software design or QC. The continuing recurrence of the accidents before effective corrective action resulted was a result of management's view. This view had faith in the correctness of the software without any apparent evidence to support it. The errors were not discovered; because the policy was to fix the symptoms without investigating the underlying causes, of which there were many.

E.1.2 Key Facts

- The software was assumed to be fail-safe and was excluded from normal safety analysis review.
- The software design and implementation had no effective review or QC practices.
- The software testing at all levels were obviously insufficient, given the results.
- Hardware interlocks were replaced by software without supporting safety analysis.
- There was no effective reporting mechanism for field problems involving software.

- Software design practices (contributing to the accidents) did not include basic, shared-data, and contention management mechanisms normal in multi-tasking software. The necessary conclusion is that the programmer was not fully qualified for the task.
- The design was unnecessarily complex for the problem. For instance, there were more parallel tasks than necessary. This was a direct cause of some of the accidents.

E.1.3 Lessons Learned

1 Changeover from hardware to a software implementation must include a review of assumptions, physics and rules.

2 Testing should include possible abuse or bypassing of expected procedures.

3 Design and implementation of software must be subject to the same safety analysis, review and QC as other parts of the system.

4 Hardware interlocks should not be completely eliminated when incorporating software interlocks.

5 Programmer qualifications are as important as qualifications for any other member of the engineering team.

E.2 Missile Launch Timing Causes Hangfire

E.2.1 Summary

An aircraft was modified from a hardware-controlled missile launcher to a software-controlled launcher. The aircraft was properly modified according to standards, and the software was fully tested at all levels before delivery to operational test. The normal weapons rack interface and safety overrides were fully tested and documented. The aircraft was loaded with a live missile (with an inert warhead) and sent out onto the range for a test firing.

The aircraft was commanded to fire the weapon, whereupon it did as designed. Unfortunately, the design did not specify the amount of time to unlock the holdback and was coded to the assumption of the programmer. In this case, the assumed time for unlock was insufficient and the holdback locked before the weapon left the rack. As the weapon was powered, the engine drove the weapon while attached to the aircraft. This resulted in a loss of altitude and a wild ride, but the aircraft landed safely with a burned out weapon.

E.2.2 Key Facts

- Proper process and procedures were followed as far as specified.
- The product specification was re-used without considering differences in the software implementation, i.e., the timing issues. Hence, the initiating event was a specification error.
- While the acquirer and user had experience in the weapons system, neither had experience in software. Also, the programmer did not have experience in the details of the weapons system. The result was that the interaction between the two parts of the system was not understood by any of the parties.

E.2.3 Lessons Learned

- Because the software-controlled implementation was not fully understood, the result was flawed specifications and incomplete tests. Therefore, even though the software and subsystem were thoroughly tested against the specifications, the system design was in error, and a mishap occurred.
- Changeover from hardware to software requires a review of design assumptions by all relevant specialists acting jointly. This joint review must include all product specifications, interface documentation, and testing.
- The test, verification and review processes must each include end-to-end event review and test.

E.3 Reused Software Causes Flight Controls to Shut Down

E.3.1 Summary

A research vehicle was designed with fly-by-wire digital control and, for research and weight considerations, had no hardware backup systems installed. The normal safety and testing practices were minimized or eliminated by citing many arguments. These arguments cited use of experienced test pilots, limited flight and exposure times, minimum number of flights, controlled airspace, use of monitors and telemetry, etc. Also, the argument justified the action as safer; because the system reused software from similar vehicles currently operational.

The aircraft flight controls went through every level of test, including "iron bird" laboratory tests that allow direct measurement of the response of the flight components. The failure occurred on the flight line the day before actual flight was to begin after the system had successfully completed all testing. The flight computer was operating for the first time unrestricted by test routines and controls. A reused portion of the software was inhibited during earlier testing as it conflicted with certain computer functions. This was part of the reused software taken from a proven and safe platform because of its functional similarity. This portion was now enabled and running in the background.

Unfortunately, the reused software shared computer data locations with certain safety-related functions; and it was not partitioned nor checked for valid memory address ranges. The result was that as the flight computer functioned for the first time, it used data locations where this reused software had stored out-of-range data on top of safety-related parameters. The flight computer then performed according to its design when detecting invalid data and reset itself. This happened sequentially in each of the available flight control channels until there were no functioning flight controls. Since the system had no hardware backup system, the aircraft would have stopped flying if it were airborne. The software was quickly corrected and was fully operational in the following flights.

E.3.2 Key facts

- Proper process and procedures were minimized for apparently valid reasons; i.e., the (offending) software was proven by its use in other similar systems.

- Reuse of the software components did not include review and testing of the integrated components in the new operating environment. In particular, memory addressing was not validated with the new programs that shared the computer resources.

E.3.3 Lessons Learned

- Safety-related, real-time flight controls must include full integration testing of end-to-end events. In this case, the reused software should have been functioning within the full software system.
- Arguments to bypass software safety, especially in software containing functions capable of a Kill/Catastrophic event, must be reviewed at each phase. Several of the arguments to minimize software safety provisions were compromised before the detection of the defect.

E.4 Flight Controls Fail at Supersonic Transition

E.4.1 Summary

A front line aircraft was rigorously developed, thoroughly tested by the manufacturer, and again exhaustively tested by the Government and finally by the using service. Dozens of aircraft had been accepted and were operational worldwide when the service asked for an upgrade to the weapons systems. One particular weapon test required significant telemetry. The aircraft change was again developed and tested to the same high standards including nuclear weapons carriage clearance. This additional testing data uncovered a detail missed in all of the previous testing.

The telemetry showed that the aircraft computers all failed—ceased to function and then restarted—at specific airspeed (Mach 1). The aircraft had sufficient momentum and mechanical control of other systems so that it effectively "coasted" through this anomaly, and the pilot did not notice.

The cause of this failure originated in the complex equations from the aerodynamicist. His specialty assumes the knowledge that this particular equation will asymptotically approach infinity at Mach 1. The software engineer does not inherently understand the physical science involved in the transition to supersonic speed at Mach 1. The system engineer who interfaced between these two engineering specialists was not aware of this assumption and, after receiving the aerodynamicist's equation for flight, forwarded the equation to software engineering for coding. The software engineer did not plot the equation and merely encoded it in the flight control program.

E.4.2 Key Facts

- Proper process and procedures were followed to the stated requirements.
- The software specification did not include the limitations of the equation describing a physical science event.
- The computer hardware accuracy was not considered in the limitations of the equation.
- The various levels of testing did not validate the computational results for the Mach 1 portion of the flight envelope.

E.4.3 Lessons Learned

- Specified equations describing physical world phenomenon must be thoroughly defined, with assumptions as to accuracy, ranges, use, environment, and limitations of the computation.
- When dealing with requirements that interface between disciplines, it must be assumed that each discipline knows little or nothing about the other and, therefore, must include basic assumptions.
- Boundary assumptions should be used to generate test cases as the more subtle failures caused by assumptions are not usually covered by ordinary test cases (division by zero, boundary crossing, singularities, etc.).

E.5 Incorrect Missile Firing from Invalid Setup Sequence

E.5.1 Summary

A battle command center with a network controlling several missile batteries was operating in a field game exercise. As the game advanced, an order to reposition the battery was issued to an active missile battery. This missile battery disconnected from the network, broke-down their equipment and repositioned to a new location in the grid.

The repositioned missile battery arrived at the new location and commenced setting up. A final step was connecting the battery into the network. This was allowed in any order. The battery personnel were still occupying the erector/launcher when the connection that attached the battery into the network, was made elsewhere on the site. This cable connection immediately allowed communication between the battery and the battle command center.

The battle command center, meanwhile, had prosecuted an incoming "hostile" and designated the battery to "fire," but targeted to use the old location of the battery. As the battery was off-line, the message was buffered. Once the battery crew connected the cabling, the battle command center computer sent the last valid commands from the buffer; and the command was immediately executed. Personnel on the erector/launcher were thrown clear as the erector/launcher activated on the old slew and acquire command. Personnel injury was slight as no one was pinned or impaled when the erector/launcher slewed.

E.5.2 Key Facts

- Proper process and procedures were followed as specified.
- Subsystems were developed separately with ICDs.
- Messages containing safety-related commands were not "aged" and reassessed once buffered.
- Battery activation was not inhibited until personnel had completed the set-up procedure.

E.5.3 Lessons Learned

- System engineering must define the sequencing of the various states (dismantling, reactivating, shutdown, etc.) of all subsystems with human confirmations and re-initialization of state variables (e.g., site location) at critical points.
- System integration testing should include buffering messages (particularly safety-related) and demonstration of disconnect and restart of individual subsystems to verify that the system always transitions between states safely.
- Operating procedures must clearly describe (and require) a safe and comprehensive sequence in dismantling and reactivating the battery subsystems with particular attention to the interaction with the network.

E.6 Operator's Choice of Weapon Release Overridden by Software

E.6.1 Summary

During field practice exercises, a missile weapon system was carrying both practice and live missiles to a remote site and was using the transit time for slewing practice. Practice and live missiles were located on opposite sides of the vehicle. The acquisition and tracking radar was located between the two sides causing a known obstruction to the missiles' field of view.

While correctly following command-approved procedures, the operator acquired the willing target, tracked it through various maneuvers, and pressed the weapons release button to simulate firing the practice missile. Without the knowledge of the operator, the software was programmed to override his missile selection in order to present the best target to the best weapon. The software noted that the current maneuver placed the radar obstruction in front of the practice missile seeker while the live missile had acquired a positive lock on the target and was unobstructed. The software, therefore, optimized the problem and deselected the practice missile and selected the live missile. When the release command was sent, it went to the live missile; and "missile away" was observed from the active missile side of the vehicle when no launch was expected.

The "friendly" target had been observing the maneuvers of the incident vehicle and noted the unexpected live launch. Fortunately, the target pilot was experienced and began evasive maneuvers, but the missile tracked and still detonated in close proximity.

E.6.2 Key Facts

- Proper procedures were followed as specified, and all operations were authorized.

- All operators were thoroughly trained in the latest versions of software.
- The software had been given authority to select “best” weapon, but this characteristic was not communicated to the operator as part of the training.
- The indication that another weapon had been substituted (live vs. practice) by the software was displayed in a manner not easily noticed among other dynamic displays.

E.6.3 Lessons Learned

- The versatility (and resulting complexity) demanded by the requirement was provided exactly as specified. This complexity, combined with the possibility that the vehicle would employ a mix of practice and live missiles was not considered. This mix of missiles is a common practice and system testing must include known scenarios such as this example to find operationally based hazards.
- Training must describe the safety-related software functions such as the possibility of software overrides to operator commands. This must also be included in operating procedures available to all users of the system.

F Process Charts

The process charts in Appendix H graphically depict the process described in this document. These charts show the process from a high, system-level safety assessment process to details regarding individual steps and tasks in the process. It is not the intent of these charts to supplant the content of AOP-15: the system-level process both provides the basis for the Software Systems Safety Process and shows the inherent cohesiveness of the Software Systems Safety Process with the System Safety Process. Therefore, the system-level charts are essential to the overall process description. Following the process charts, the reader can gain an understanding of the process and the various steps involved in each task within the process. In its electronic version, hyperlinks guide the reader.

Each process chart presented in Appendix F contains the following:

- Primary task description
- Process inputs and outputs
- Entry and exit criteria
- Personnel involved
- Primary sub-processes or tasks
- Critical interfaces

Not all of the items listed in the process chart apply in every instance of a software safety program. Rather, they provide a comprehensive list of items that may apply. The user will need to tailor the process charts to the development program. For example, small development efforts will not have the breadth of documentation available in larger programs. Therefore, one system-level document may provide both a system specification and a software requirements specification. Conversely, complex programs may have several subsystem specifications which may have one or more software requirements specifications. Likewise, one individual may fulfill several engineering and/ or management roles in a small development program.

System Definition & Safety Planning
Intermediate Flow Diagram

Preceding Process
•None

System Definition & Safety Planning
(Click here for Low Level Process Flow)

Next Process
•System Functional Hazard Analysis

Inputs:

- Contract Documents (procurement environment)
  - CM, QA, & Safety Requirements and System Specification
  - System capabilities specification
- Concept of operations Program funding and schedule (including WBS)
- Lessons learned
- System architecture concepts
- Functional allocation
- Software Development Plan

Entry Criteria:

- Contract Start

Purpose:

Establish the System / Software Safety Program and define program safety scope

Processes:

- Develop and/or tailor SSMP, including Software Safety consideration
  - Establish interface with stakeholders (e.g., security, user, survivability, etc.)
- Develop and/or tailor SSPP, including Software Safety section
  - Schedule
  - SSWG charter
  - Identify resources
  - Plan safety for warranty, maintenance and upgrades
- Conduct system functional allocation analysis for safety impact
- Conduct system architecture analysis for safety considerations
- Couple SWS with software development, CM, QA and test planning processes
- Support PESHE development (safety summary rollup)
- Develop and document safety programmatic risks (i.e., schedule, funding)

Players:

- Project manager (Identify safety manager & Allocate Safety Funding)
- Safety Manager (Establish Plans)
- SSE (Support safety program)
- SWSE (Support safety program)
- Sys Eng (Develop SS)
- S/W Eng (Conduct Studies)
- CM (Develop CM Plan)
- QA (Develop QA & SQA Plans)
- Domain experts (Support Sys Eng)
- User (Support Sys Eng)
- Test (Develop T&E Plans)

Outputs:

- SSMP
- SSPP, Including Software Safety Plan
- Safety programmatic risks (ie, schedule, funding)
- Safe coding practices
- Safety aspects of software development, QA & CM plans
- System safety input to related documents (e.g., environmental regulation and compliance

Exit Objective(s):

- Scope safety program, safety funding, & safety schedule
- Establish safety involvement within project and across all project interfaces
- Support system architecture and recommend preferred system architecture for safety

1

System Functional Hazard Analysis
Intermediate Flow Diagram

Preceding Process
•System Definition & Safety Planning

System Functional Hazard Analysis
(Click here for Low Level Process Flow)

Next Process
•System Software Requirements Hazard Analysis

Purpose:
Conduct Functional Allocation of Safety Related system elements & gain Review Authority Concurrence of System / Software Plans.

Inputs:
- System/Subsystem Specification
- Test and Evaluation Master Plan
- Mishap records for lessons learned from similar systems
- Generic safety requirement
  - Regulatory requirements
  - SOW clauses
  - Review Authority requirements
- Established SSWG
- System Architecture
- CONOPS

Processes:
- Identify system level hazard and potential mishaps
- Identify safety critical functions and/or requirements
- Conduct Safety Requirements Criteria Analysis (SRCA)
- Analyze architectures
- Allocate safety critical functions to system/software architectures
- Develop safety recommendations
- Define safety testing approach
- Establish working interface with review authority
- Conduct initial safety risk assessment
- Brief planned safety program to safety review authority
- Evaluate safety review authority recommendations

Outputs:
- Preliminary Hazard List
- Initial Safety Assessment
- Inputs to requirements traceability tool and requirements documents
- Safety requirements criteria analysis
- Introductory briefing for Safety Review Authority

Entry Criteria:
- Established System Safety Program

Players:
- SSWG (Review/Adjudicate Analyses)
- Safety Manager (Execute Plans)
- SSE (Conduct Safety Analysis)
- SWSE (Conduct S/W Safety Analysis)
- Sys Eng (Develop SSS)
- S/W Eng (Define Architecture)
- Domain experts (Support Sys Eng)
- User (Support Sys Eng)
- Test (Develop TEMP)
- Review Authority (Provide concurrence with Safety Plans

Exit Objective(s):
- Identify system level hazards and potential mishaps
- Identify safety critical functions and/or requirements
- Allocate safety critical functions to system and software architecture
  - Optimize the allocation of safety critical software in the system
  - Consider software, hardware, and Human Systems Integration (HSI)
- Gain Safety Review Authority concurrence regarding safety approach

2

System Software Requirements Hazard Analysis
Intermediate Flow Diagram

Preceding Process
•System Functional Hazard Analysis

System Software Requirements Hazard Analysis
(Click here for Low Level Process Flow)

Next Process
•Design & Implementation Hazard Analysis

Inputs:

- Software Criticality Matrix (SCM)
- Software Requirements Specification (SRS)
- Interface Requirements Specification (IRS)
- System Architecture with software visible
- Software Architecture

Entry Criteria:

- Defined allocation of System Functions to Software
- Defined Software and Interface Requirements

Purpose:

Identify and derive Software Safety Requirements and link Hazards to requirements to support Preliminary Design.

Processes:

- Analyze software functions to refine safety related functions
  - ➢ Identify additional safety related functions
  - ➢ Validate existing list of safety related functions
  - ➢ Identify interfaces to safety related functions
- Identify hazard software causal factors
  - ➢ Prioritize and categorize causal factors
- Tag or mark safety requirements in the SRS and IRS
- Derive safety requirements to mitigate identified hazards/causal factors
- Requirements walkthroughs to verify correctness and completeness of safety related requirements
- Confirm level of rigor equates to cumulative assigned risk index
- Update hazard analyses
- Update Safety Requirements Criteria Analysis (SRCA)

Players:

- ➢ Project manager (Support Meetings)
- ➢ Safety manager (Execute Plans)
- ➢ SSE (Conduct Analyses)
- ➢ SWSE (Determine SHRI)
- ➢ Sys Eng (Develop SRS/IRS)
- ➢ S/W Eng (Refine Architecture)
- ➢ CM (Maintain Requirements)
- ➢ QA (Support Requirement Walk-troughs)
- ➢ Domain experts (Support Sys Eng)
- ➢ User (Support Sys Eng)
- ➢ Test (Refine Test Plans)

Outputs:

- Hazard analysis update
- Inputs to hazard tracking system
- Safety Assessment for Review Authority
- Update to requirements traceability

Exit Objective(s):

- Link hazards to requirements and vice versa
  - ➢ Verify completeness of hazards and requirements
  - ➢ Develop requirements to mitigate hazards
- Identify software safety requirements
- Obtain Review Authority concurrence on progress

3

COTS Interoperability Hazard Analysis
Intermediate Flow Diagram

Preceding Process
•**Parallel Process** with System Software Requirements Hazard Analysis

COTS Interoperability Hazard Analysis
(Click here for Low Level Process Flow)

Next Process
•Design & Implementation Hazard Analysis

Purpose:
Identify and derive Software Safety Requirements for proposed COTS to be integrated into the system Preliminary Design. Always include unused functionalities.

Inputs:
- Software Criticality Matrix (SCM)
- Software Requirements Specification (SRS)
- Interface Requirements Specification (IRS)
- COTS Application History
- Supplier Data "Cut" Sheets
- System architecture showing COTS locations
- Software architecture showing COTS/NDI/GOTS locations and functions

Entry Criteria:
- Defined allocation of System Functions to Software
- Defined Software and Interface Requirements

Processes:
- Analyze COTS Requirements and Functions against System Requirements
  - Review COTS product History, if available, for similar application
  - Identify historical SHRI, if available, for used and unused functions
  - Identify interfaces to and threads from safety related functions
- Identify, Categorize, & Prioritize COTS Requirements & Functions
  - Prioritize and categorize causal factors for used and unused functions
- Match COTS to Safety Critical Functions, add new hazards for COTS unused functions
- Establish COTS level of rigor, unused functions inherit highest level of rigor
- Tag or mark safety requirements in the SRS, IRS, & Supplier Cut Sheets
- Analyze dead, unused, or inactivated options in COTS, and monitor/mitigation designs
- Analyze Functional & Behavioral characteristics of COTS
- Develop Safety Mitigation Requirements for COTS used and unused functions
- Develop Safety Regression Test criteria and procedures for used & inactivated COTS functions
- Update hazard analyses & Safety Requirements Criteria Analysis (SRCA)

Players:
- Project manager (Support Meetings)
- Safety Manager (Execute Plans)
- SSE (Conduct Analyses)
- SWSE (Determine SHRI)
- Sys Eng (Develop SRS/IRS)
- S/W Eng (Refine Architecture)
- COTS manager (buy, warranty)
- CM (Maintain Requirements)
- QA (Support Requirement Walk-troughs)
- Domain experts (Support Sys Eng)
- User (Support Sys Eng)
- Test (Refine Test Plans)

Outputs:
- Hazard analysis update
- Inputs to hazard tracking system
- Inputs to Safety Assessment for Review Authority
- Update to requirements traceability
- COTS used and unused functions list to input to buy and tech refresh criteria
- COTS safety criticality Indexes

Exit Objective(s):
- Link hazards to COTS requirements and vice versa
  - Verify completeness of hazards and requirements
  - Develop requirements to mitigate COTS hazards
- Identify COTS software safety requirements
- Obtain Review Authority concurrence on progress

4

COTS = commercial-off-the-shelf software, GOTS, NDI, tools, compilers, debuggers, and most open source software

Design & Implementation Hazard Analysis
Intermediate Flow Diagram (Page 1 of 2)

Preceding Process
- System Software Requirements Hazard Analysis

Design & Implementation Hazard Analysis
(Click here for Low Level Process Flow)

Next Process
- Software Test & Validation

Inputs:
- Baseline System/Subsystem Specification
- Baseline Software Requirement Specification (SRS) and Interface Requirement Specification (IRS)
- Software Design (SDD) and Interface Design (IDD) Documents
- COTS selections as completed
- Operating and Support Documents

Entry Criteria:
- Defined allocation of System Functions to Design
- Defined Software and Interface Design Requirements

Purpose:
Identify, track, & resolve hazards and software causal factors through detailed analysis and test.

Processes:
- Conduct SSHA
  - Hardware, software and human
  - Establish SHRI and conduct analysis according to SHRI level of rigor
- Incorporate and analyze COTS functionality/interactions/threads/interfaces
  - Match existing hazards to COTS (both used and unused functions)
  - Analyze COTS for introduction of new hazards, mitigate unused functions
  - Establish SHRI for new and/or modified requirements and conduct analysis according to SHRI level of rigor
- Conduct SHA
  - Include System of Systems Hazards, human computer interface, etc.
  - Establish SHRI for new and/or modified requirements and conduct analysis according to SHRI level of rigor
- Incorporate and analyze all system interfaces, esp. COTS, and tag for safety risk
- Conduct O&SHA
  - Develop Safety Notes, Warnings, Cautions, training and workarounds
  - Recommend updates to user manuals and training
- Develop new or modified Requirements
- Develop Software Safety Test Cases
- Coordinate with Reviewing Authority
- Produce software CDR Safety Assessment (SAR)
- Obtain Review Authority concurrence with software design
- Populate Hazard Tracking System
  - Develop Test Readiness evaluation material(s)
  - For each build, assess residual safety risk of partial-functionality and "left-out functionality" of build(s) to-date

NOTE: Applicable in research, spiral and rapid development incremental releases

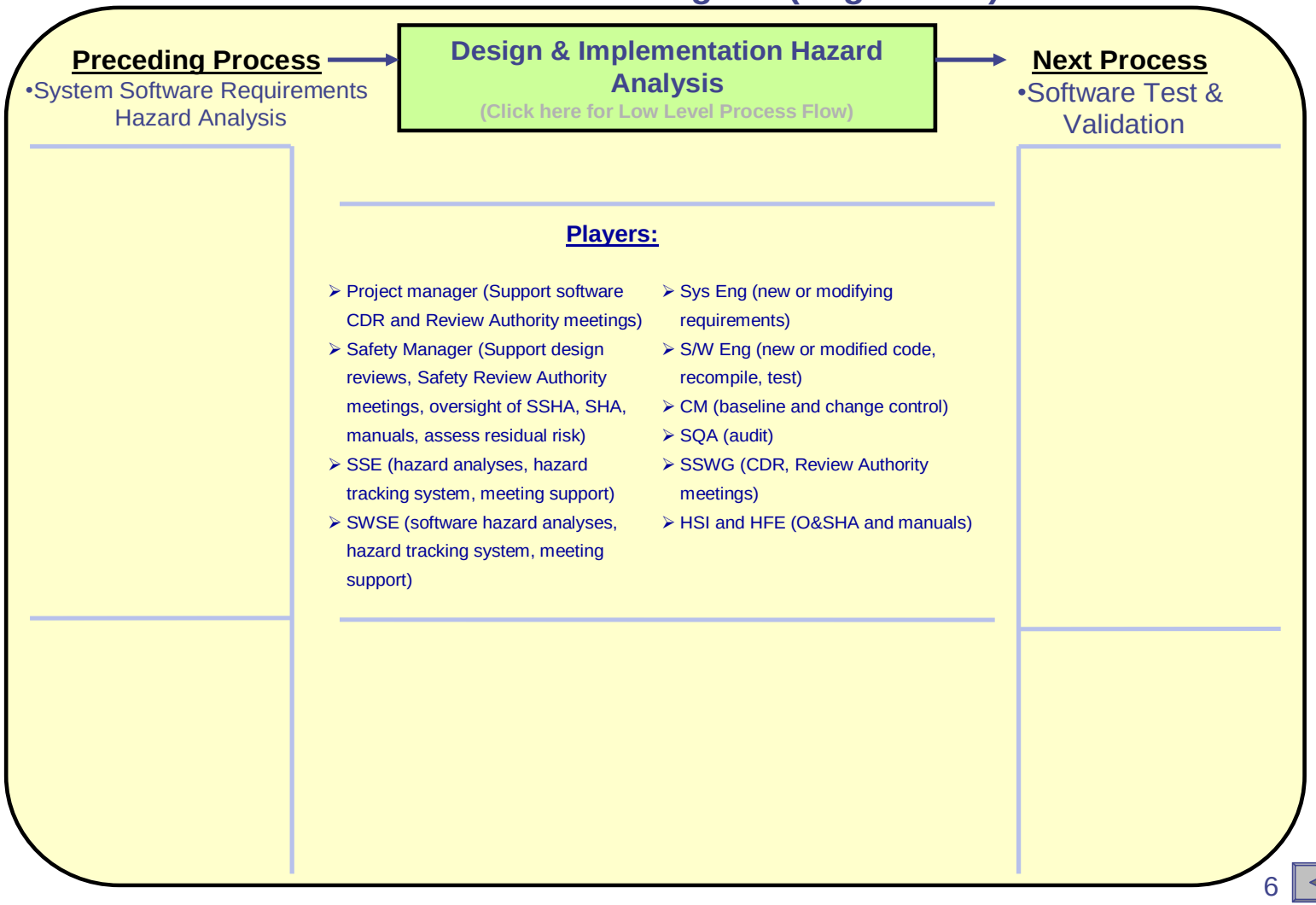
Players: See Next Page

Outputs:
- Updated Hazard Tracking System
- Updated and new Hazard Analyses
  - Updated SRCA
  - Updated software CDR SAR
  - Review Authority data package
- Residual safety risk
- New and/or modified requirements and associated test cases
- Recommend updates to user manuals and training
- Special COTS implementation safety requirements and COTS SHRI

Exit Objective(s):
- Identify, track, resolve hazards and causal factors
- Create and coordinate mitigations and test cases
- Analyze implementation of safety related functions and safety requirements to ensure they satisfy their intent

5

Design & Implementation Hazard Analysis
Intermediate Flow Diagram (Page 2 of 2)



Software Test & Validation Intermediate Flow Diagram

Preceding Process
•Design & Implementation Hazard Analysis

Software Test & Validation
(Click here for Low Level Process Flow)

Next Process
•Certification

Inputs:

- Test plans, descriptions and procedures
- Test results
- Software test anomalies (STRs, SPRs, TORs, etc.)
- Test environment and configuration definitions
- Build definitions
- SCNs, ECPs, etc.
- Regression suite(s), especially safety, and process triggers that initiate full sets of regression.
- COTS test requirements

Entry Criteria:

- Defined Software Safety Requirements

Purpose:

Verify Software Safety Requirements are validated through software developmental and System Integration Testing.

Processes:

- Identify residual risk
- Support software and integration testing
- Analyze test results
  - Identify safety related anomalies
  - Categorize safety related anomalies
  - Verify resolution of anomalies
- Update safety documentation (e.g., SAR, Hazard Tracking System, etc)
- Provide safety evidence to safety review authority prior to government integration testing, if required
- If IV&V is conducted, then provide safety support and review results

Players:

- Project manager(Support TRR & RA Meetings)
- Safety Manager (Execute Plans)
- SSE (Analyze Test Results)
- SWSE (Analyze Test Results)
- Sys Eng (Analyze Test Results)
- S/W Eng (Analyze Test Results)
- HSI & HFE Eng (Review Test Results)
- SSWG (Concur with Safety test procedures & test approach)
- CM (Maintain Requirements Test Matrix)
- QA (Support Test Procedure Walk-troughs)
- S/W & System Test Engineers (Develop & execute test procedures)
- IV&V personnel if required (Execute Test Procedures)
- Safety Review Authority (Concur with Test Event

Outputs:

- Updated safety documentation (e.g., SAR, Hazard Tracking System, etc)
- Safety review authority TDP
- Risk acceptance recommendations

Exit Objective(s):

- Verify the implementation of the intent of safety requirements
- Verify the implementation of safety related functions
- Identify and categorize safety anomalies
- Develop evidence supporting safety certification
- Document residual risk for government test events

7

Defect Resolution
Intermediate Flow Diagram (1 of 2)

Preceding Process
•Multiple Entry Points

Defect Resolution
(Click here for Low Level Process Flow)

Next Process
•Multiple Exit Points

Inputs:

- Anomaly report(s)
- Engineering change proposal(s)
- Hazard Tracking System
- Baselined Requirements
- Test results and observations
- Planned test procedures

Entry Criteria:

- Defect identified and communicated to software developers or In-Service Engineering Agent

Purpose:

Assure that software defects identified during development and program lifecycle are assessed for safety, risk identified, tracked, and resolved.

Processes:

- Identify anomalies (review STRs, etc.) passed through for safety impact(s)
- Create new entry in Hazard Tracking System
- Investigate possible causal factors
  - Tool suite, compiler error
  - Build, link list error (CM)
  - Code Implementation
  - Integration error
  - Design error
  - Requirement error
  - Misinterpreted requirement, interface, hazard, use case, etc…
  - COTS unused function
  - Legacy functionality not regressed
- Determine safety risk of each causal factor
- Update previous safety analysis(es) for anomaly(ies)
- Develop mitigation(s):
  - Rebuilt Code
  - Patch(es) [when absolutely necessary!] followed by regression tests
  - And also analyze for new hazards created by mitigation
    - If new hazard, then return to 2-1 on slide 5
- Check mitigations for new or changes to Requirements
- Coordinate with:
  - If software, then coordinate with Software Engineer
  - If hardware, then coordinate with System Engineer
  - If human, coordinate with HSI and HFE, etc.

Players: See Next Page

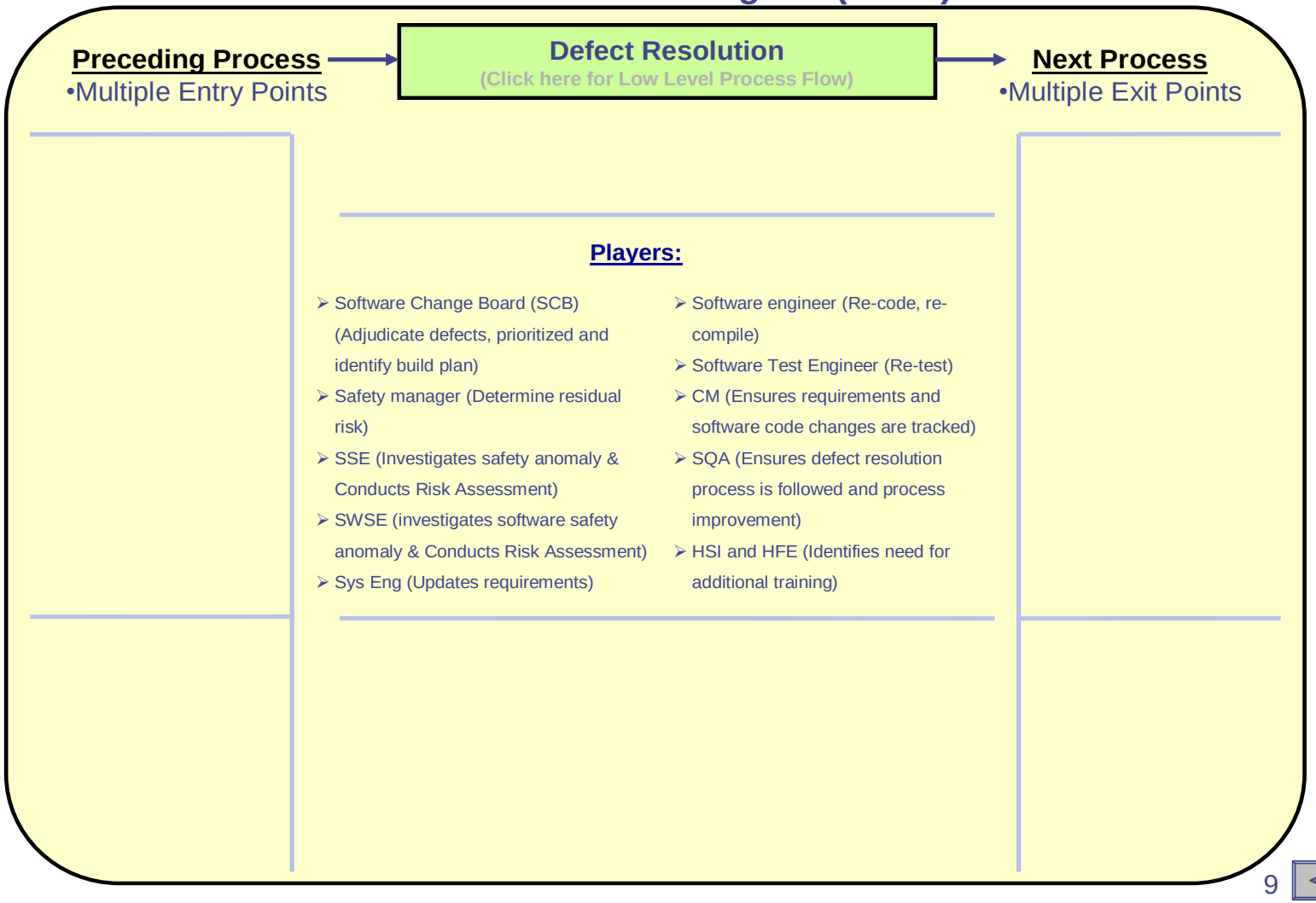
Outputs:

- Update to Baselined Requirements
- Engineering Change Proposal(s)
  - Software Change Board review
- Updated Hazard Tracking System
- Updated Hazard Analysis(es)
  - Possibly, new hazard(s)
- Recommended Test Case(s)
- Residual Safety Risk

Exit Objective(s):

- Resolve defects identified through software test and validation

Defect Resolution
Intermediate Flow Diagram (2 of 2)



9

Regression Testing
Intermediate Flow Diagram (1 of 2)

Preceding Process
•Multiple Entry Points

→ **Regression Testing**
(Click here for Low Level Process Flow) →

Next Process
•Multiple Exit Points

Inputs:
- Baselined, ready to release compiled code (anomalies removed)
- Engineering Change Proposals
- Software Trouble Reports
- Residual safety risk
- Existing Test cases for required functions to maintain through any change
- Existing Test cases for required safety functions to maintain through any change
- Existing Test cases for required requirements to maintain through any change
- Proposed new test cases for change
- Baseline safety documentation
- Qualified test tools and test suite

Entry Criteria:
- Ready to Test Software
- Qualified Test Environment

Purpose:
Assure that software modified through defect resolution does not effect other aspects of software performance, especially safety related functions.

Processes:
- Analyze functionality to be maintained through any change
- Analyze safety functionality and safety data and structure to be maintained through any change
- Analyze planned change(s) made to functionality and requirements
- Analyze new functionality provided by change
- Analyze change(s) to safety functionality and safety data and structure
- Develop new, use or modify existing, test(s) to verify and validate existing system functionality to be maintained in regardless change
- Develop new, use or modify existing, test(s) to verify and validate existing safety functionality and safety data and structure
- Develop new, use or modify existing, test(s), if required, to verify and validate new safety functionality and safety data and structure
- Develop, or modify existing, test(s) to verify and validate new system functionality and safety data and structure
- Execute combined test(s)

NOTE 1: May be a stand-alone (in Sustained Engineering) or embedded process (in Software V&V)

NOTE 2: During certain fix-fly and Sustained Engineering this and V&V may iterate in smaller tighter loops than shown in the full process flow.

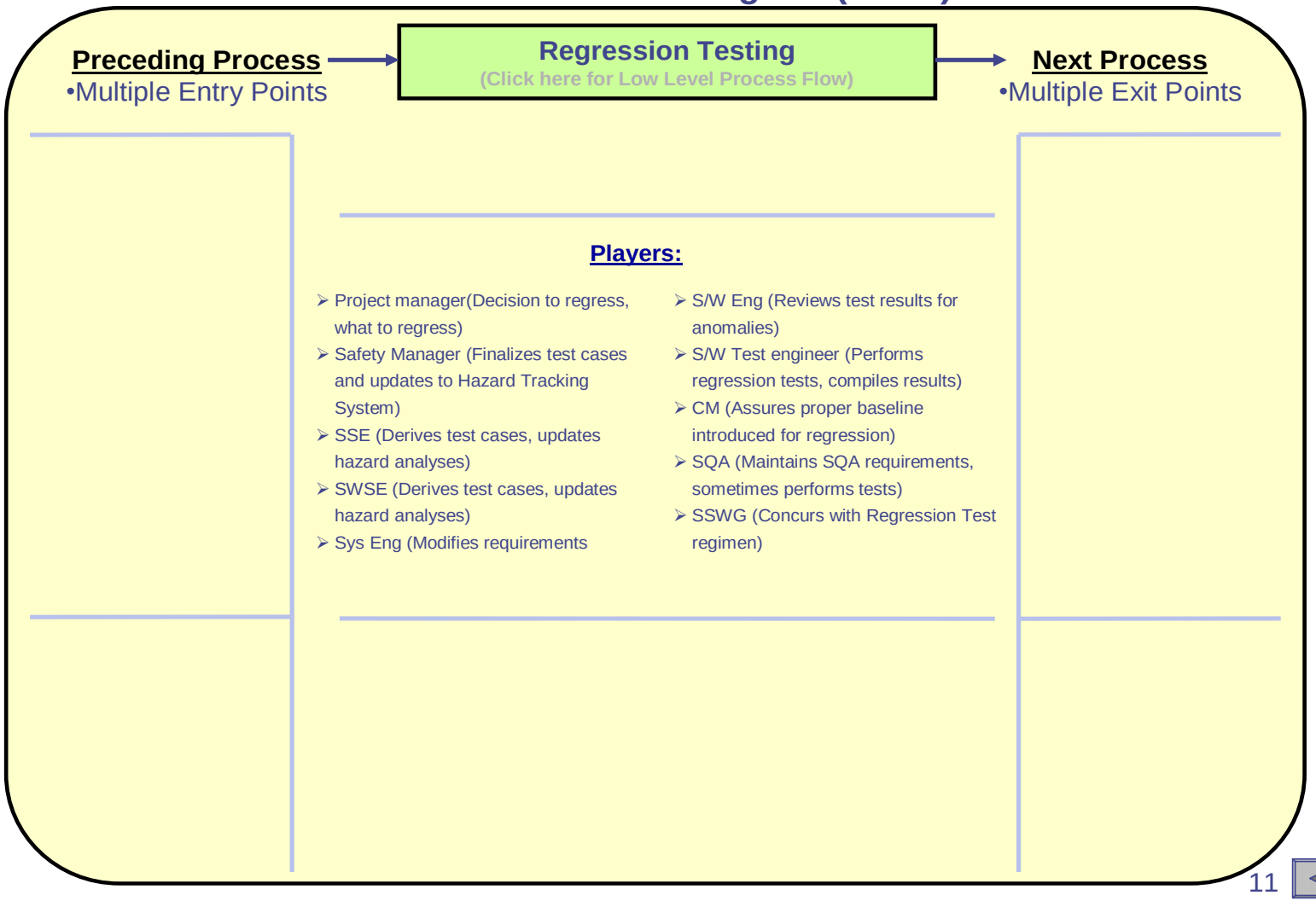
Players: See Next Page

Outputs:
- Verified and Validated change(s)
- Possible software anomalies to be rendered in return to Software V&V
- Recommendations for safety assessment
- Changes to requirements trace (SRCA)

Exit Objective(s):
- Assurance that changes or modifications in one area do not affect critical functions and safety functions in another portion of the system.
- Assurance that changes or modifications have their intended effect.
- New residual risk.

10

Regression Testing
Intermediate Flow Diagram (2 of 2)



11

Certification
Intermediate Flow Diagram

Preceding Process
•Software Test & Validation

Certification
(Click here for Low Level Process Flow)

Next Process
•Sustained Engineering

Inputs:

- Customer integration test plans, descriptions and procedures
- Customer integration and operational evaluation test results
- Customer integration test anomalies and resolutions (STRs, SPRs, TORs, etc.)
- Customer integration test environment and configuration definitions
- Build definitions

Entry Criteria:

- Completion of Factory Qualification Test

Purpose:

Provide substantiated evidence to demonstrate software is safe in the system context prior to introduction (use).

Processes:

- Develop evidence to support residual risk assessment
- Document residual risk for acceptance
- Support customer integration and certification testing
- Analyze test results supporting certification
  - Identify safety related anomalies
  - Categorize safety related anomalies
- Update safety documentation (e.g., SAR, Hazard Tracking System, etc)
- Compile safety evidence for safety review authority to support request for operational use
- Support test readiness evaluations and reviews
- Support mission control panels (i.e., review board) and readiness reviews
- Respond to and resolve safety review authority action items

Players:

- Project manager (Support Certification & RA Meetings)
- Safety Manager (Assemble Evidence, Gain Risk Acceptance for residual risk)
- SSE (Conduct Analyses / Gather Data)
- SWSE (Conduct Analyses / Gather Data)
- Sys Eng (Gather Data)
- S/W Eng (Gather Data)
- SSWG (Adjudicate final residual risk)
- CM (Verify Compliance with Plans)
- QA (Verify Compliance with Plans)
- Domain Experts (Support Sys Eng)
- User (Support Sys Eng)
- Customer integration test team (Conduct Integration Testing)
- Safety Review Authority (Weigh safety evidence & concur or assign action
- Certification authorities (Weigh evidence & concur or assign action)

Outputs:

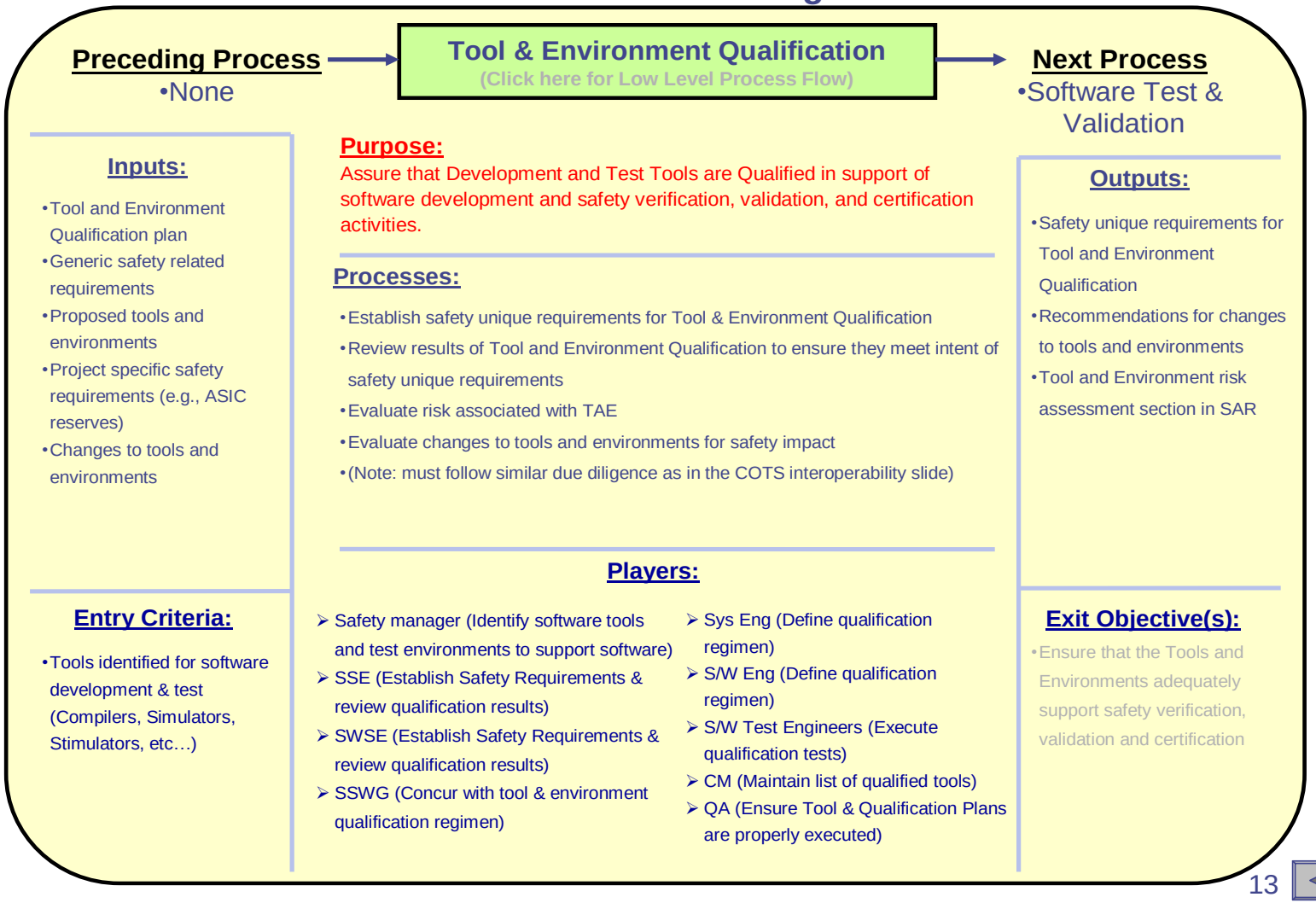
- Completed safety documentation (e.g., SAR, Hazard Tracking System, etc)
- Safety review authority TDP
- Formal acceptance of residual risk

Exit Objective(s):

- Provide evidence to certify software as safe in the system and operational environment context
- Provide evidence to accept residual risk
- Provide evidence to obtain safety review authority concurrence with operational use

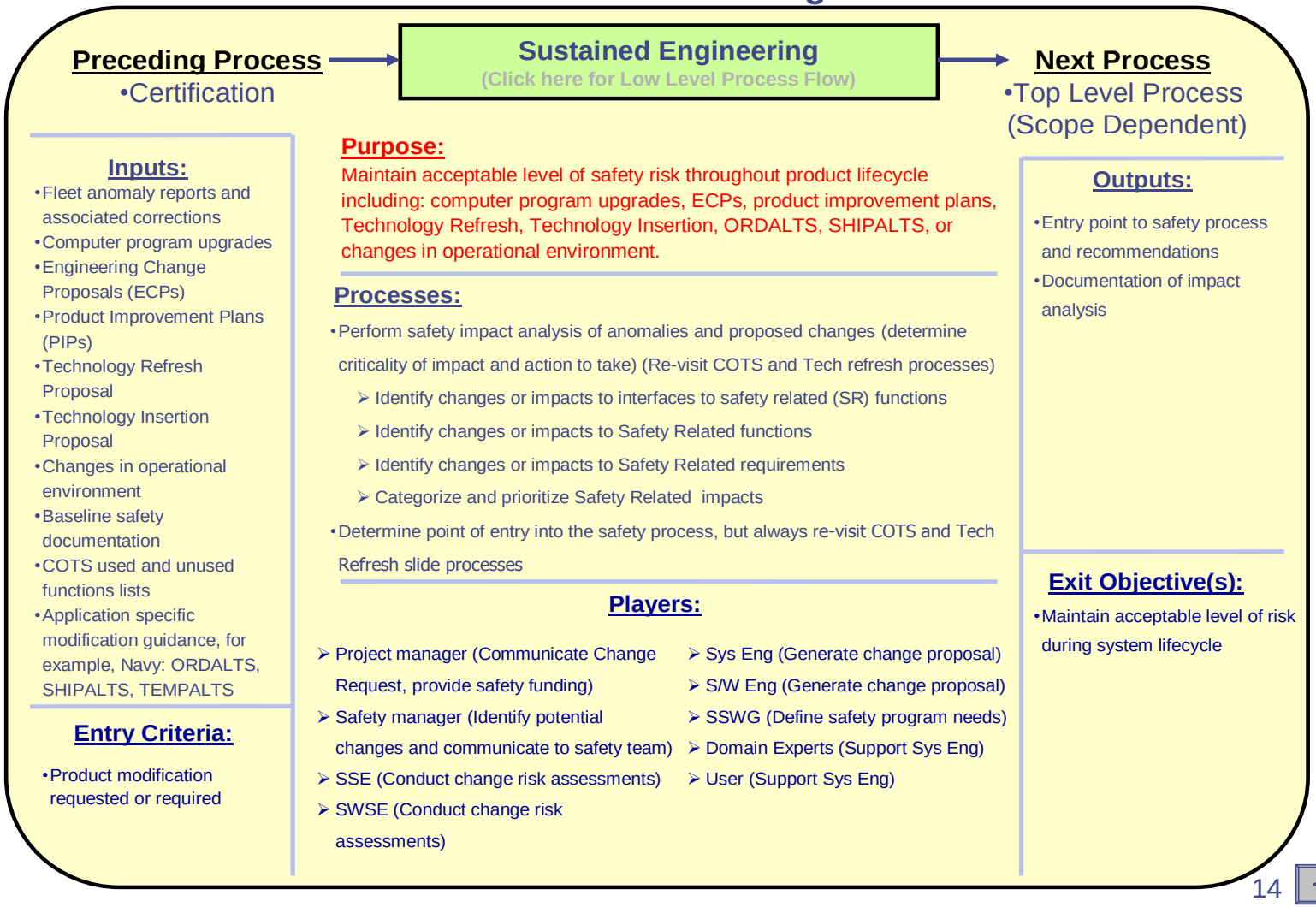
12

Tool & Environment Qualification
Intermediate Flow Diagram



13

Sustained Engineering
Intermediate Flow Diagram



14

Technology Insertion / Refresh Intermediate Flow Diagram

Preceding Process
•Sustained Engineering

Technology Insertion / Refresh
(Click here for Low Level Process Flow)

Next Process
•Top Level Process (Scope Dependent)

Inputs:

- Technology Insertion Proposal
- Baselined version(s) of hardware, software and tools suite(s)
- Supplier Data Sheets describing proposed technology (processor, memory, chip, etc.)
- Tools suite or Compiler Pre-Qualification or Accreditation
- Hazard Tracking System and associated safety analyses
- Existing SHRI
- Existing Software and Hardware engineering analyses
- Existing Regression Test suite
- Previous, if any, test results, STRs, etc.
- Previous Review Authority comments and lessons-learned

Entry Criteria:

- Decision to insert or refresh technology into existing baseline

Purpose:

Assure that the introduction or update to existing technology does not present unacceptable risk to the new system baseline.

Processes:

- Conduct Change Hazard/impact analysis
  - ➢ New Functionality or Capability
  - ➢ Hardware
    - Timing margin
    - Processor
    - Memory
    - Bus or Controller
    - PLC, EPLD, FPGA, ASIC, EEPROM, etc.
  - ➢ Software
    - COTS (always include existing used/unused functions comparison analyses to new COTS used/unused functions), mitigation update costs
    - Tools or Compiler
    - New Language
- Determine process entry point

Players:

- ➢ Project manager (Approves and budgets technology proposal)
- ➢ Safety Manager (Determines Recommendation(s) for further Safety Analysis and Test, allocates resources)
- ➢ SSE (Re-works assigned analyses)
- ➢ SWSE (Re-works assigned analyses)
- ➢ S/W Eng (Re-qualifies tools, re-times and re-sizes, re-codes, re-compiles, re-tests)
- ➢ CM (Updates baseline and configuration)
- ➢ QA (Process improvement and audits changes)
- ➢ HSI and HFE (Updates user manuals and training)

Outputs:

- Recommendation(s) for further Safety Analysis and Test (insertion point on which sheet)
  - ➢ Test recommendations
  - ➢ Updated Hazard Tracking System
  - ➢ Updated Hazard Analysis(es)
  - ➢ Updated baseline or configuration
  - ➢ Updated tools suite or compiler certifications/qualifications
  - ➢ Updated Requirements
  - ➢ Updated User Manuals and Training
  - ➢ Changed Residual Safety Risk

Exit Objective(s):

- Ensure acceptable level of risk in the introduction or insertion of new technologies into an existing baseline or system.

15

Changes In Operational Environment Intermediate Flow Diagram

Preceding Process
•Sustained Engineering

Changes In Operational Environment
(Click here for Low Level Process Flow)

Next Process
•Top Level Process (Scope Dependent)

Purpose:
Assure that a change in the operational environment of a system does not present unacceptable risk in the new environment.

Inputs:
- Documentation of Safety Impact Analysis
- Description of New Environment or new use
- Hazard Tracking System and associated safety analyses
- Existing SHRI
- Existing Software and Hardware engineering analyses
- Previous test results, STRs, etc.
- Previous Review Authority comments and lessons-learned

Entry Criteria:
- Decision to subject system / software to a change from approved environment

Processes:
- Conduct Change Hazard/impact analysis
  - ➢ Integration into new Systems of Systems Context
    - Identify safety related interfaces, existing risk index versus new index, and any constraints
    - Identify new interoperability and stakeholder safety requirements
  - ➢ New Operational Environment
    - Major Change (e.g., aircraft application to shipboard application, new interfaces)
    - Minor Change (e.g., Open Seas to Littoral Warfare and does not directly affect any safety functionality except boundaries)
      - Identify safety related environmental constraints and design assumptions, compare all old constraints and sequences to new
  - ➢ New Operational use or application
    - Major or Minor
      - Identify old system level hazards with new interfaces or bounds, and new interfaces with any safety implications
- Integration test planning with new system

Players:
- ➢ Project manager (Approves and budgets technology proposal)
- ➢ Safety Manager (Determines Recommendation(s) for further Safety Analysis and Test, allocates resources)
- ➢ SWSE (Re-works assigned analyses)
- ➢ SSWG (Define safety program needs)
- ➢ Domain Experts (Support Sys Eng)
- ➢ User (Support Sys Eng)
- ➢ SSE (Re-works assigned analyses)

Outputs:
- Recommendation(s) for further Safety Analysis and Test (insertion point on which sheet)
  - ➢ Test recommendations
  - ➢ Updated Hazard Tracking System
  - ➢ Updated Hazard Analysis(es)
  - ➢ Updated baseline or configuration
  - ➢ Updated Requirements
  - ➢ Updated User Manuals and Training
  - ➢ Changed Residual Safety Risk

Exit Objective(s):
- Ensure acceptable level of risk in the introduction or insertion of existing technologies into a new environment.

16

Fleet Anomaly Reporting
Intermediate Flow Diagram

Preceding Process
•Sustained Engineering

Fleet Anomaly Reporting
(Click here for Low Level Process Flow)

Next Process
•Defect Resolution

Inputs:

- Fleet anomaly reports
- Changes in operational environment
- Baseline safety documentation
- Casualty reports
- Commanding Officer Narrative Reports (CONAR)
- Airborne Weapon Corrective Action Reports (AWCAPS)
- Safety Center data

Purpose:
Evaluate defects discovered in the fleet, identify safety risk, and propose near and long term mitigations.

Processes:

- Perform safety impact analysis of anomalies and proposed changes (determine criticality of impact and action to take)
  - Identify interfaces to safety related functions
  - Identify changes or impacts to safety related functions
  - Identify changes or impacts to safety related requirements
  - Categorize and prioritize safety related impacts
- Determine point of entry into the safety process

Outputs:

- Risk assessment
- Fleet notification
  - Interim mitigation

Entry Criteria:

- User defect identified and communicated to the In-Service Engineering Agent

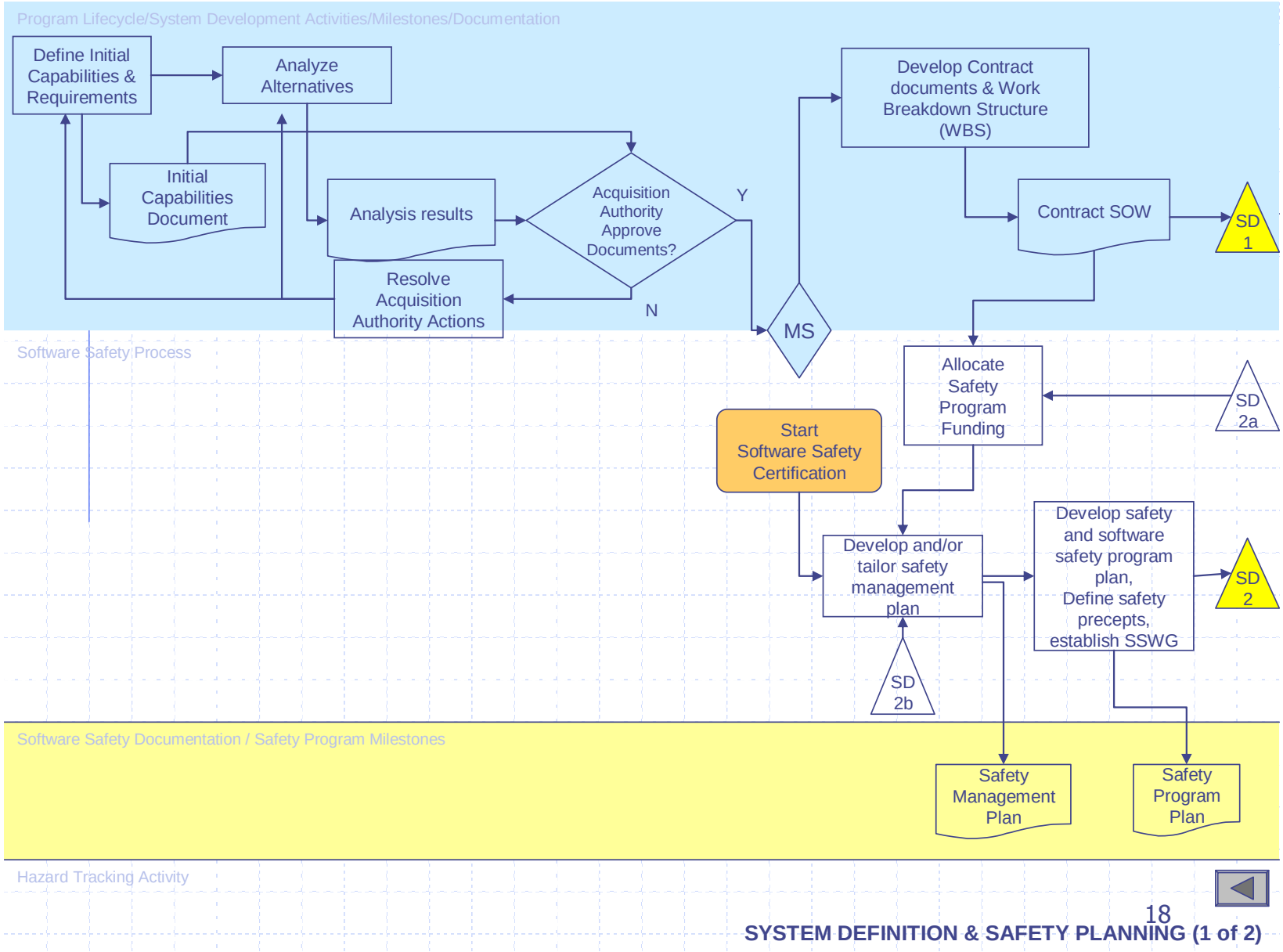
Players:

- Safety Manager (Identify potential changes and communicate to safety team)
- SSE (Conduct risk assessments)
- SWSE (Conduct risk assessments)
- Sys Eng (Identify root cause)
- S/W Eng (Identify root cause)
- Domain Experts (Support Sys Eng)
- User (Document identified deficiency)

Exit Objective(s):

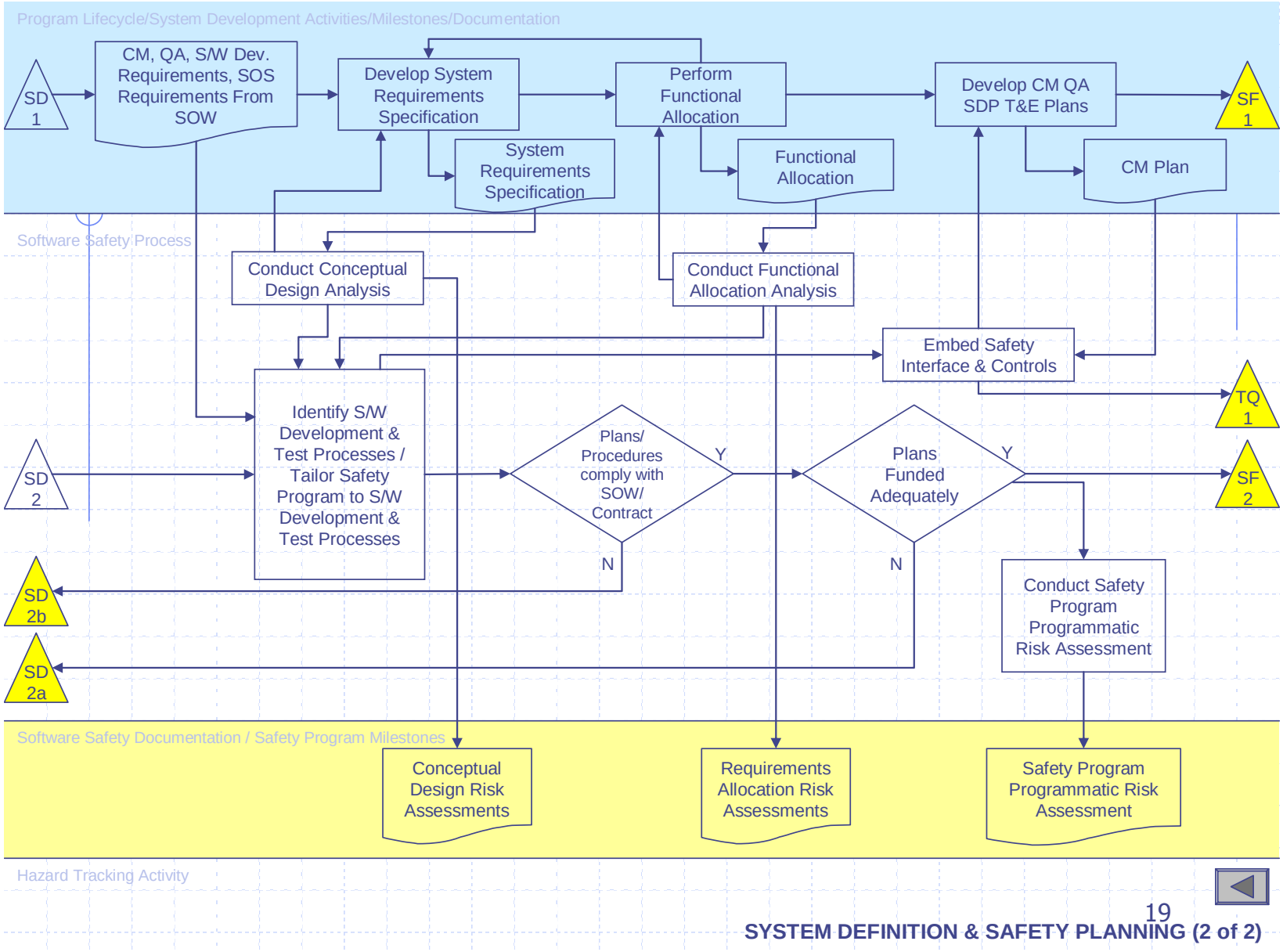
- Evaluate anomalies discovered during system lifecycle

17



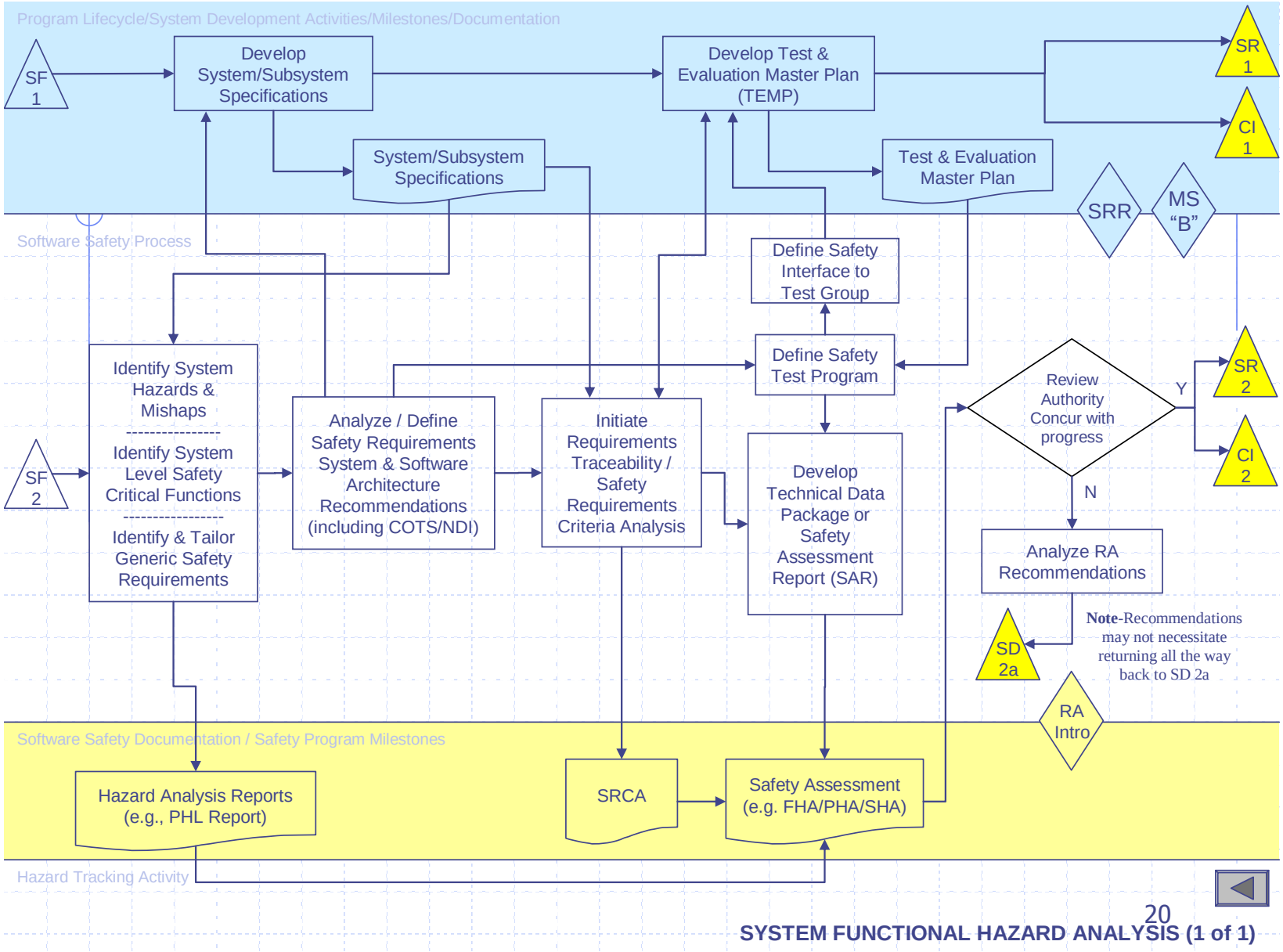
18

SYSTEM DEFINITION & SAFETY PLANNING (1 of 2)



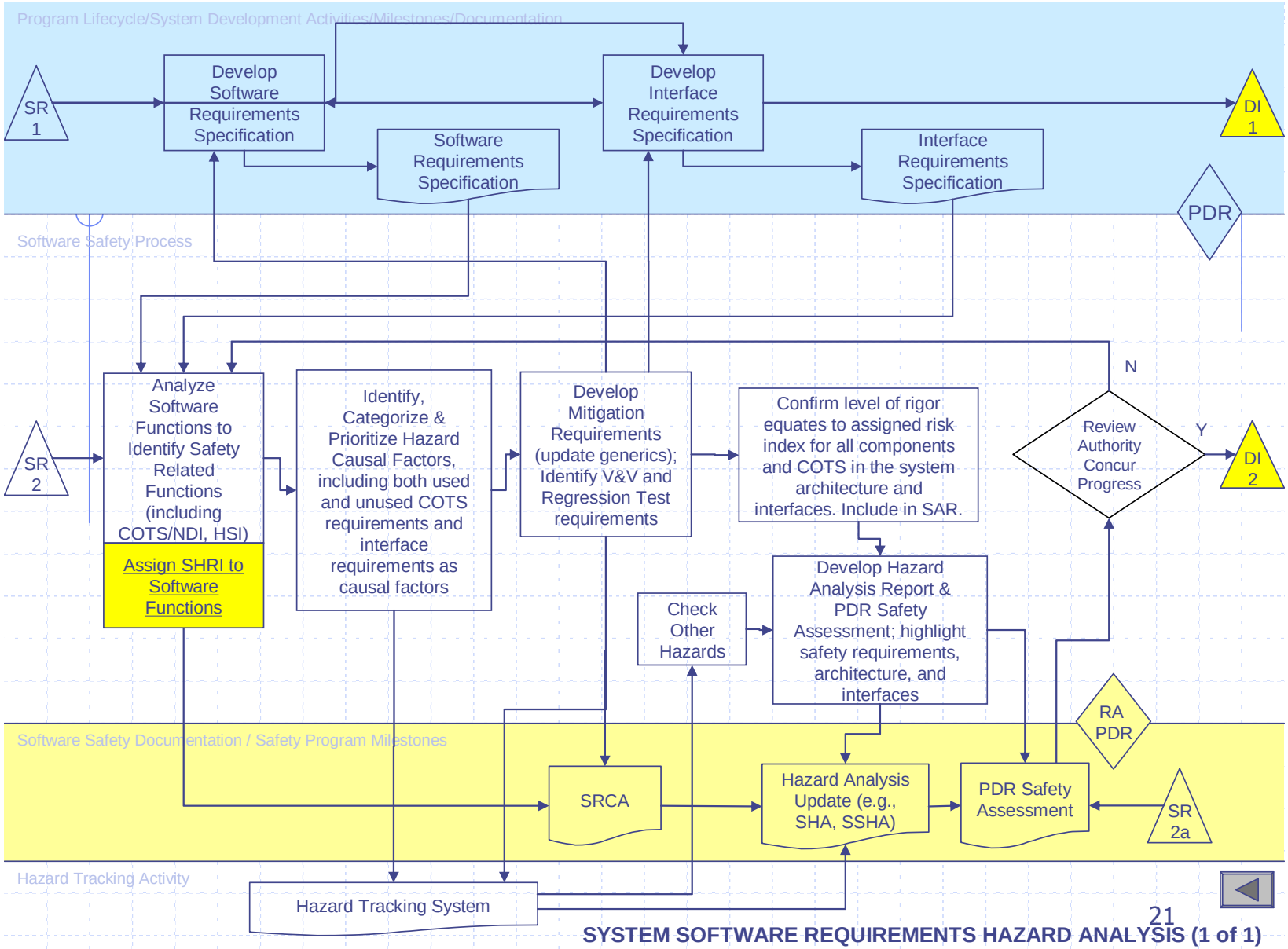
19

SYSTEM DEFINITION & SAFETY PLANNING (2 of 2)



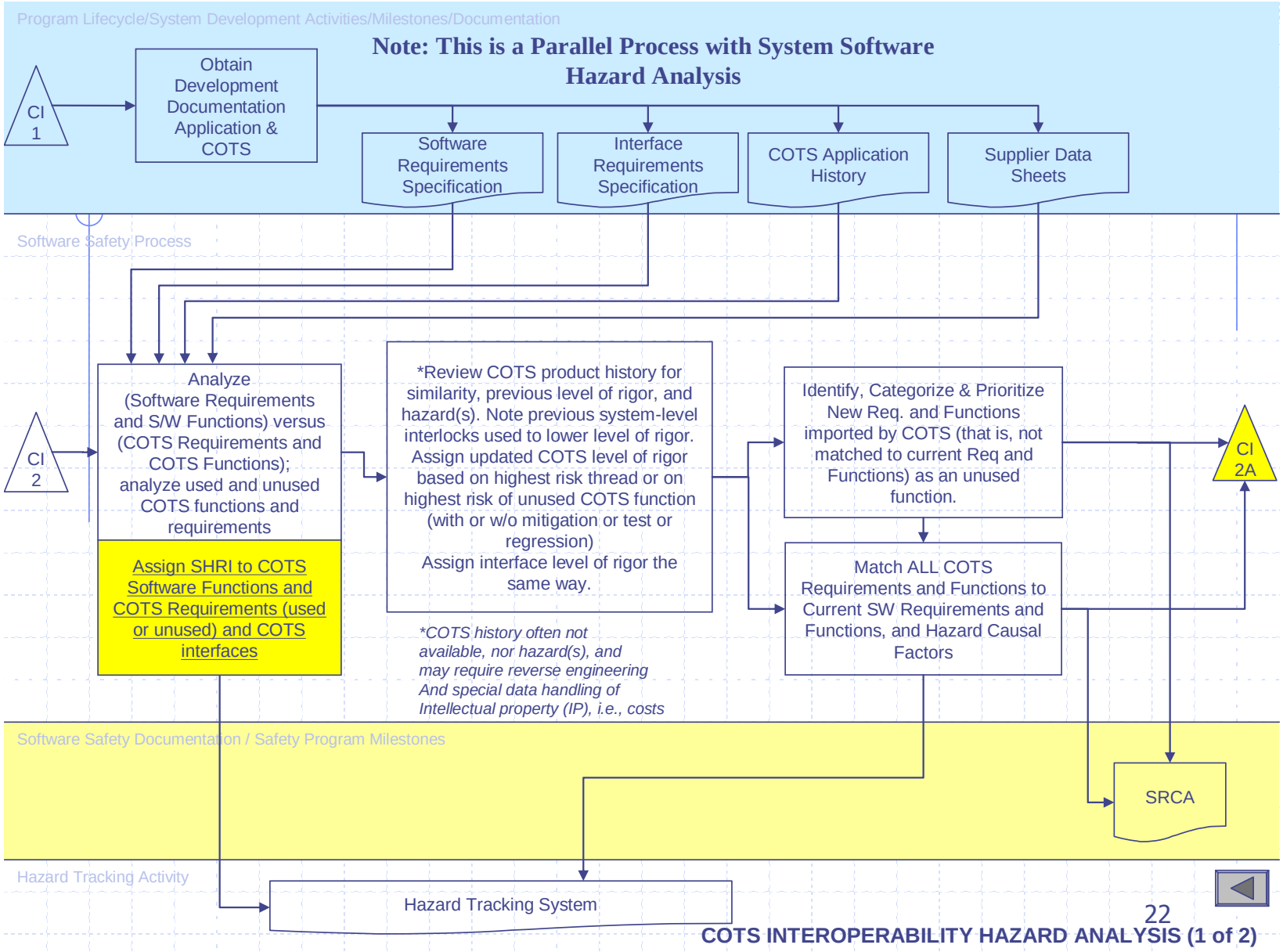
20

SYSTEM FUNCTIONAL HAZARD ANALYSIS (1 of 1)



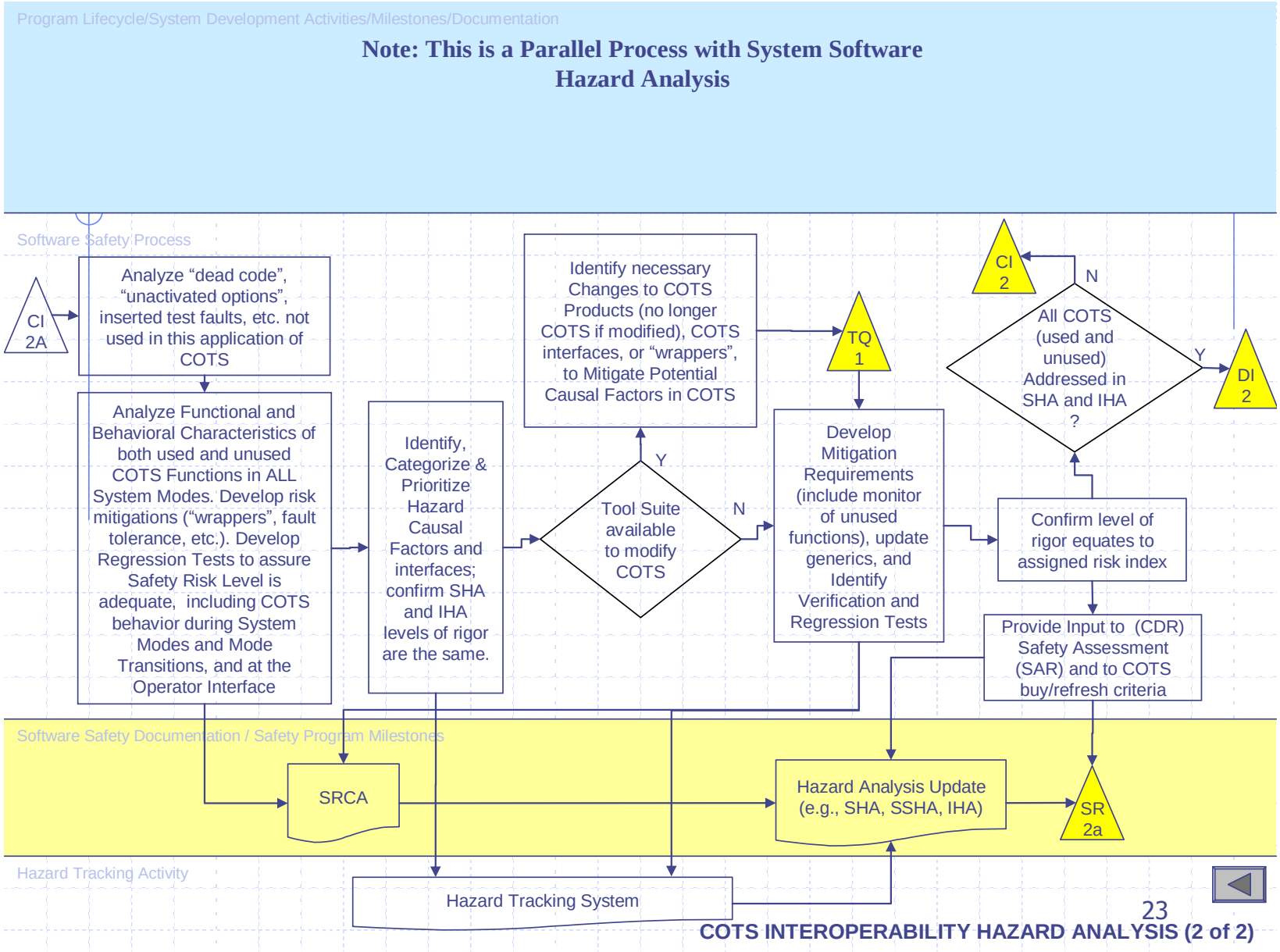
21

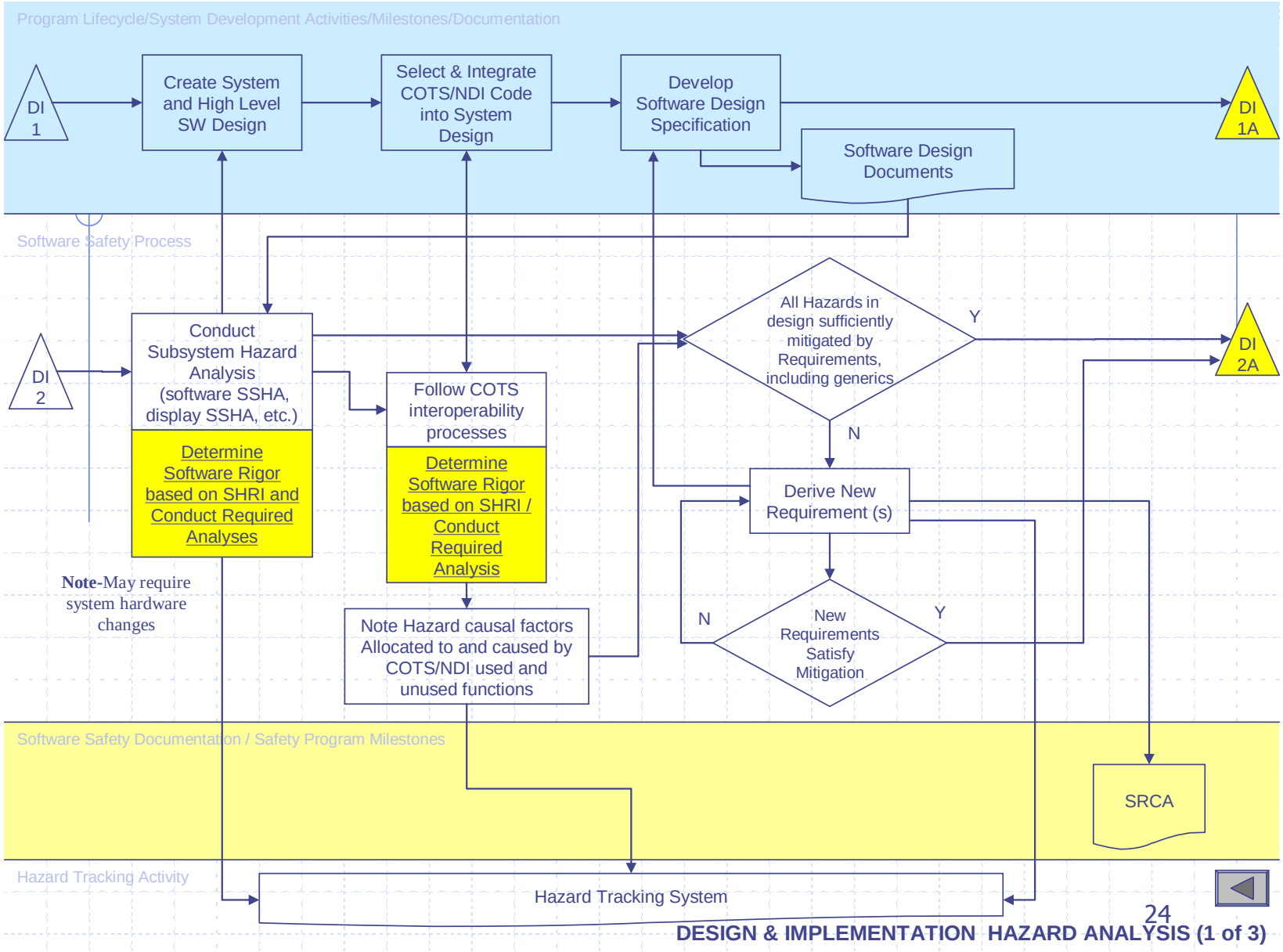
SYSTEM SOFTWARE REQUIREMENTS HAZARD ANALYSIS (1 of 1)

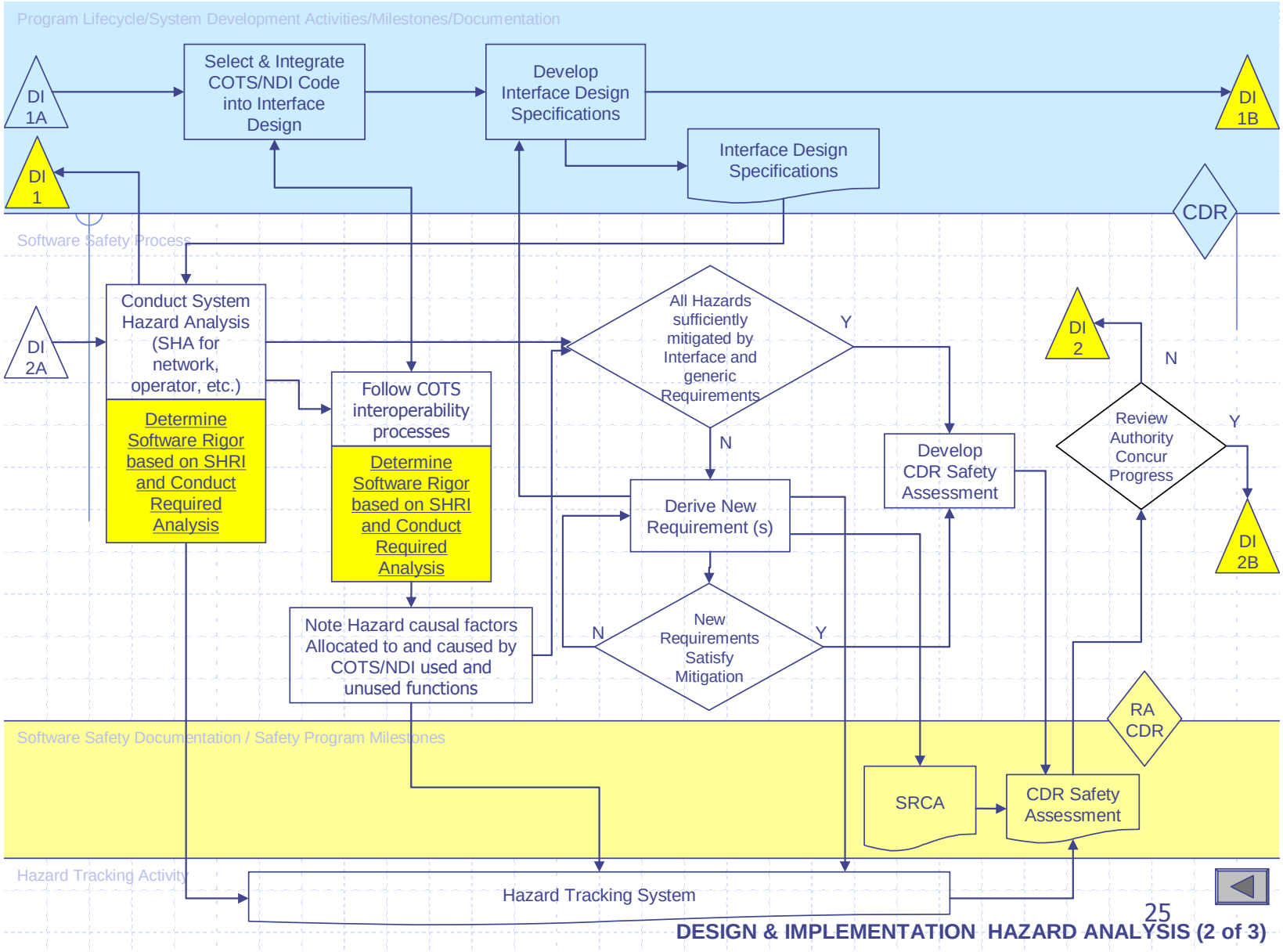


22

COTS INTEROPERABILITY HAZARD ANALYSIS (1 of 2)

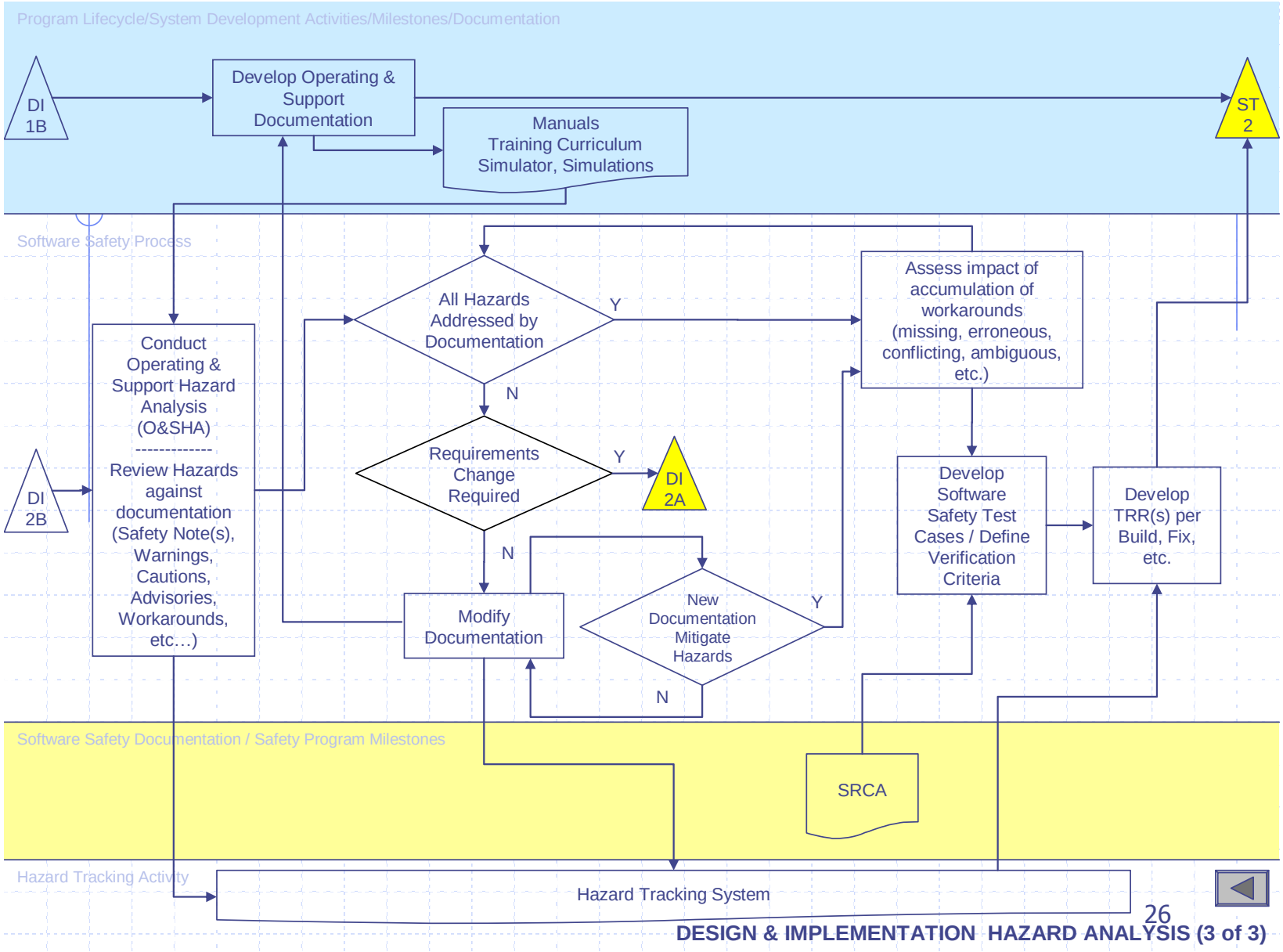

Program Lifecycle/System Development Activities/Milestones/Documentation
Note: This is a Parallel Process with System Software Hazard Analysis
Software Safety Process
CI 2A
Analyze "dead code", "unactivated options", inserted test faults, etc. not used in this application of COTS
Analyze Functional and Behavioral Characteristics of both used and unused COTS Functions in ALL System Modes. Develop risk mitigations ("wrappers", fault tolerance, etc.). Develop Regression Tests to assure Safety Risk Level is adequate, including COTS behavior during System Modes and Mode Transitions, and at the Operator Interface
Identify, Categorize & Prioritize Hazard Causal Factors and interfaces; confirm SHA and IHA levels of rigor are the same.
Identify necessary Changes to COTS Products (no longer COTS if modified), COTS interfaces, or "wrappers", to Mitigate Potential Causal Factors in COTS
Tool Suite available to modify COTS
Y
N
TQ 1
Develop Mitigation Requirements (include monitor of unused functions), update generics, and Identify Verification and Regression Tests
CI 2
N
All COTS (used and unused) Addressed in SHA and IHA ?
Y
DI 2
Confirm level of rigor equates to assigned risk index
Provide Input to (CDR) Safety Assessment (SAR) and to COTS buy/refresh criteria
Software Safety Documentation / Safety Program Milestones
SRCA
Hazard Analysis Update (e.g., SHA, SSHA, IHA)
SR 2a
Hazard Tracking Activity
Hazard Tracking System
23
COTS INTEROPERABILITY HAZARD ANALYSIS (2 of 2)


Program Lifecycle/System Development Activities/Milestones/Documentation
DI 1
Create System and High Level SW Design
Select & Integrate COTS/NDI Code into System Design
Develop Software Design Specification
DI 1A
Software Design Documents
Software Safety Process
DI 2
Conduct Subsystem Hazard Analysis (software SSHA, display SSHA, etc.)
Determine Software Rigor based on SHRI and Conduct Required Analyses
Follow COTS interoperability processes
Determine Software Rigor based on SHRI / Conduct Required Analysis
All Hazards in design sufficiently mitigated by Requirements, including generics
Y
N
DI 2A
Derive New Requirement (s)
New Requirements Satisfy Mitigation
N
Y
Note-May require system hardware changes
Note Hazard causal factors Allocated to and caused by COTS/NDI used and unused functions
Software Safety Documentation / Safety Program Milestones
SRCA
Hazard Tracking Activity
Hazard Tracking System
24
DESIGN & IMPLEMENTATION HAZARD ANALYSIS (1 of 3)



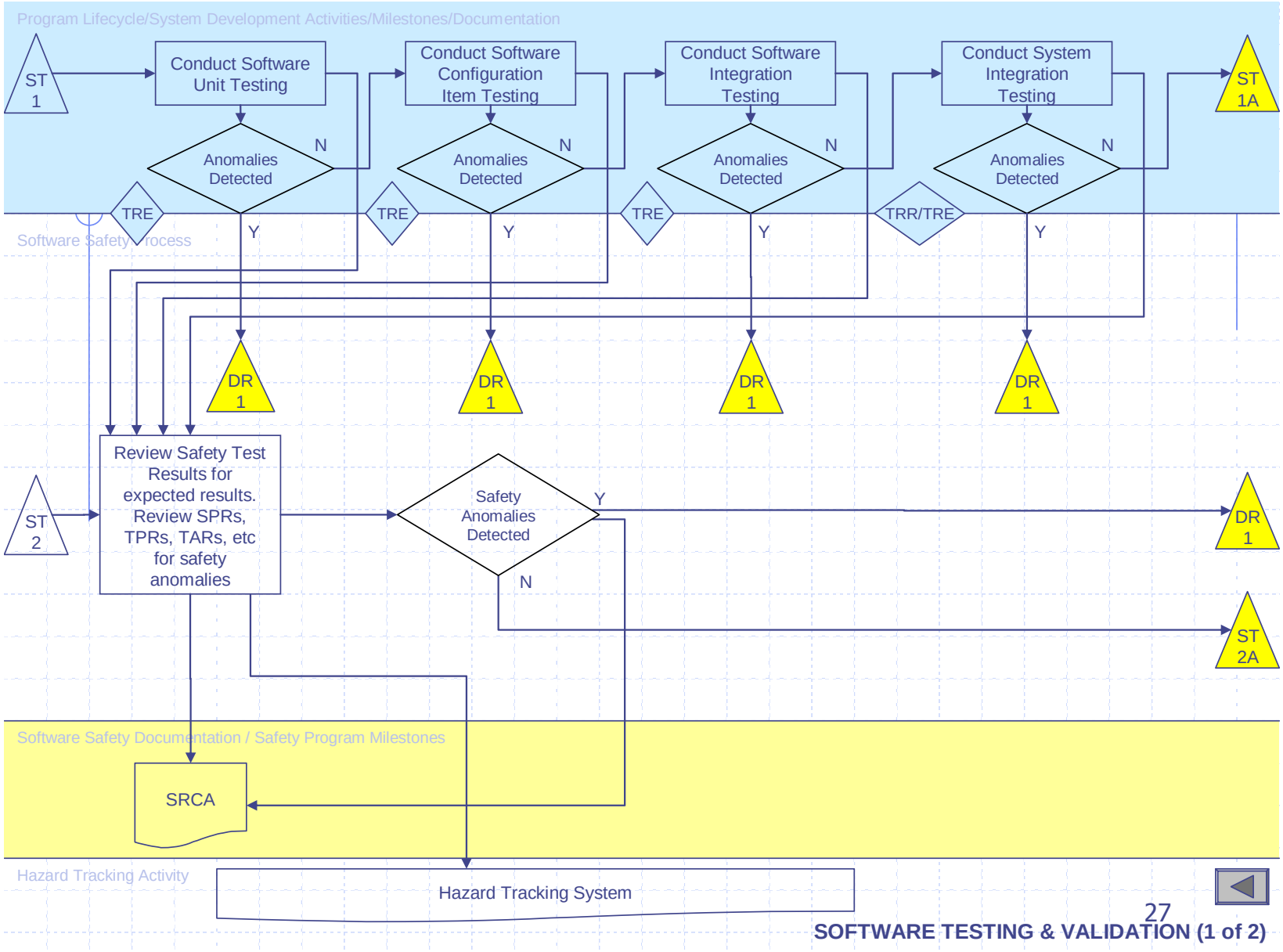
25

DESIGN & IMPLEMENTATION HAZARD ANALYSIS (2 of 3)



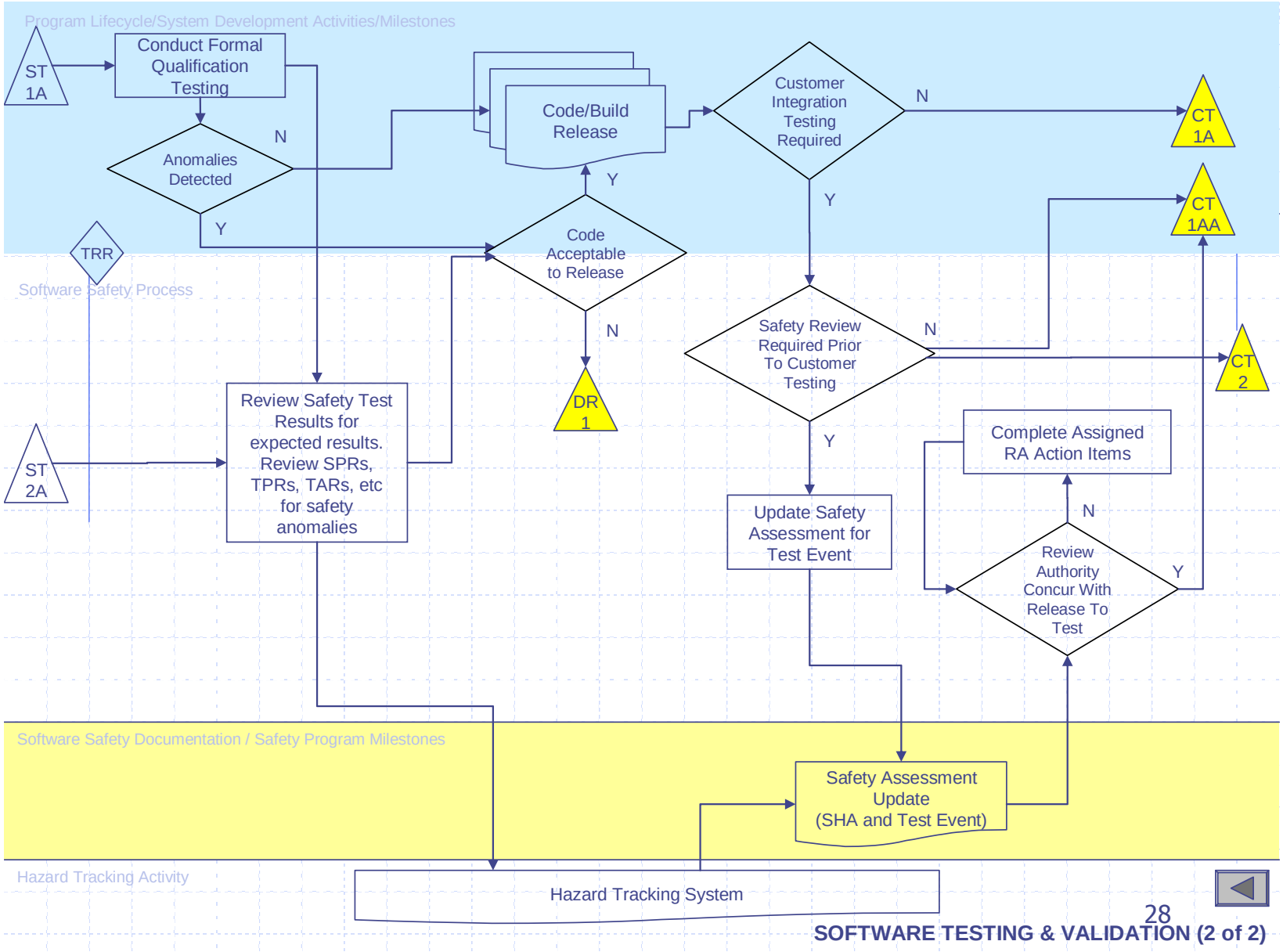
26

DESIGN & IMPLEMENTATION HAZARD ANALYSIS (3 of 3)



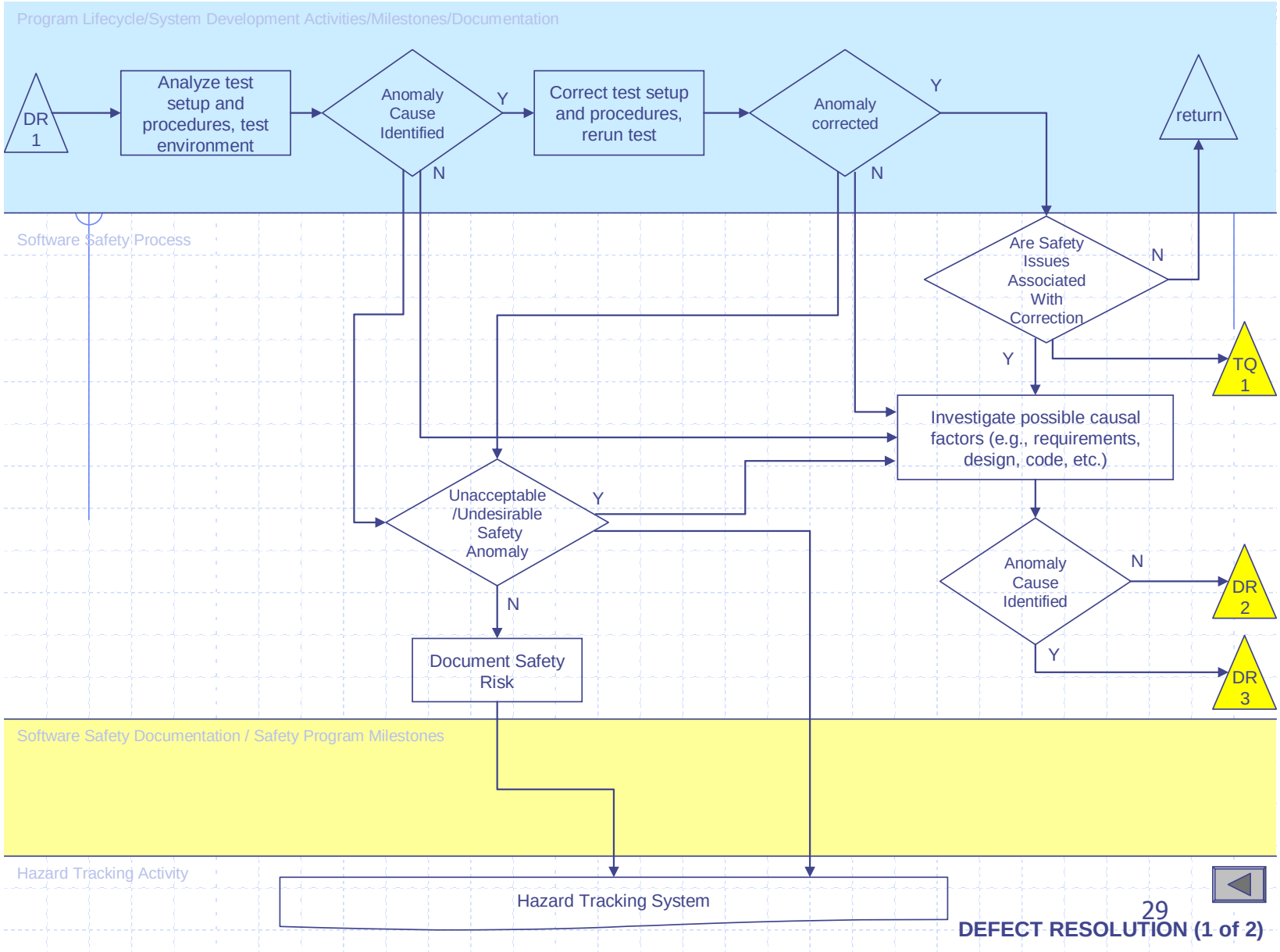
27

SOFTWARE TESTING & VALIDATION (1 of 2)



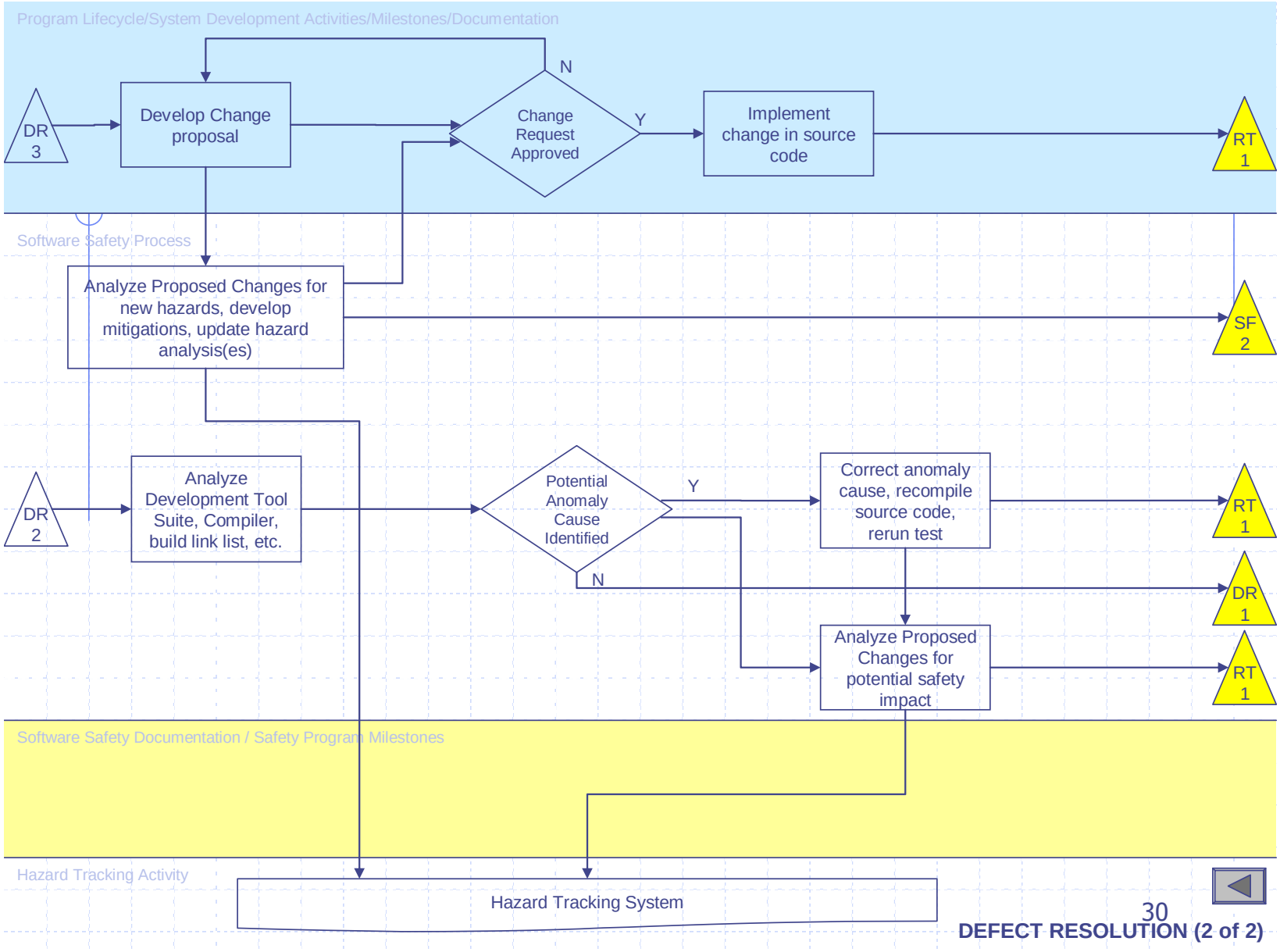
28

SOFTWARE TESTING & VALIDATION (2 of 2)

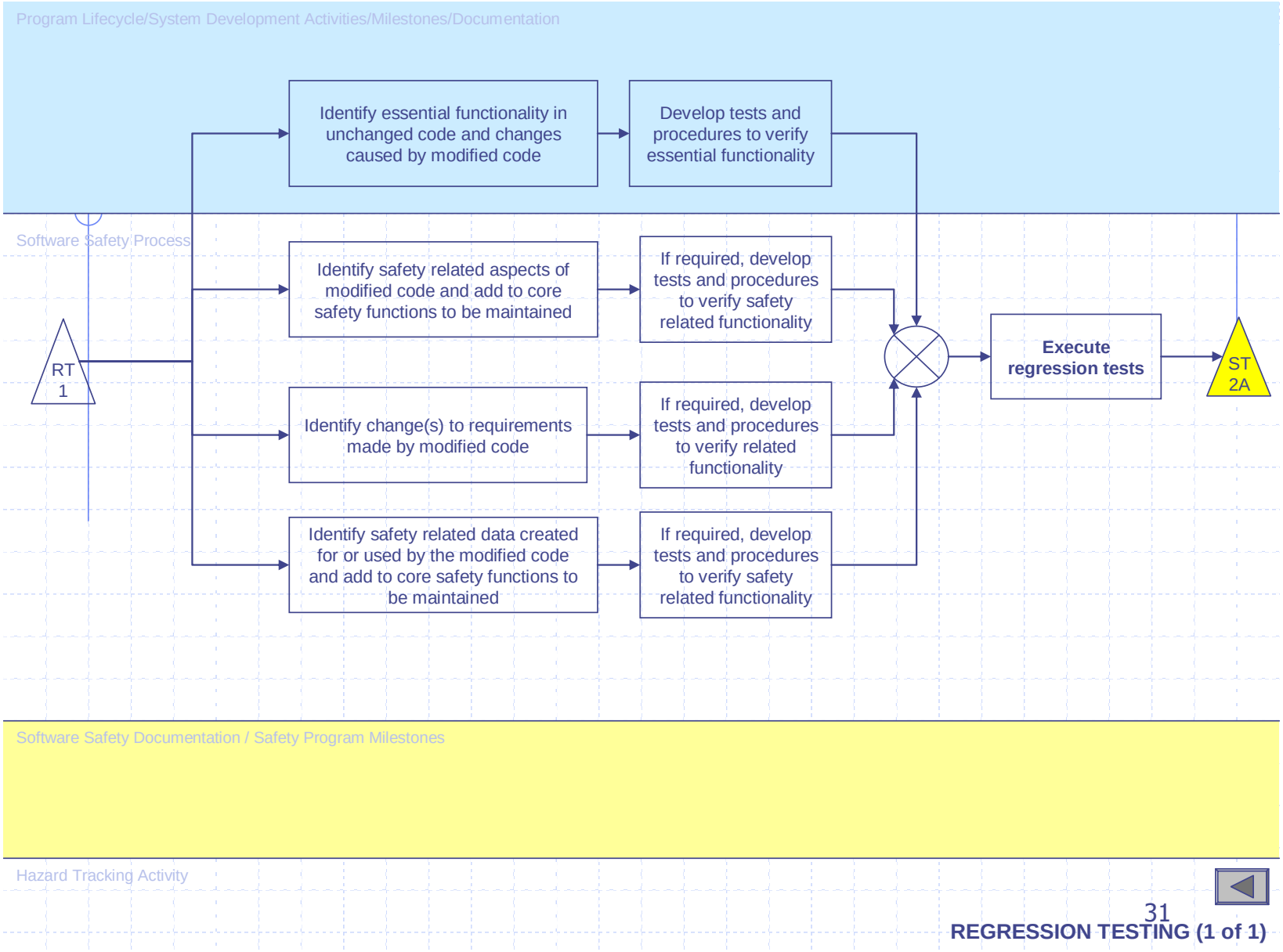


29

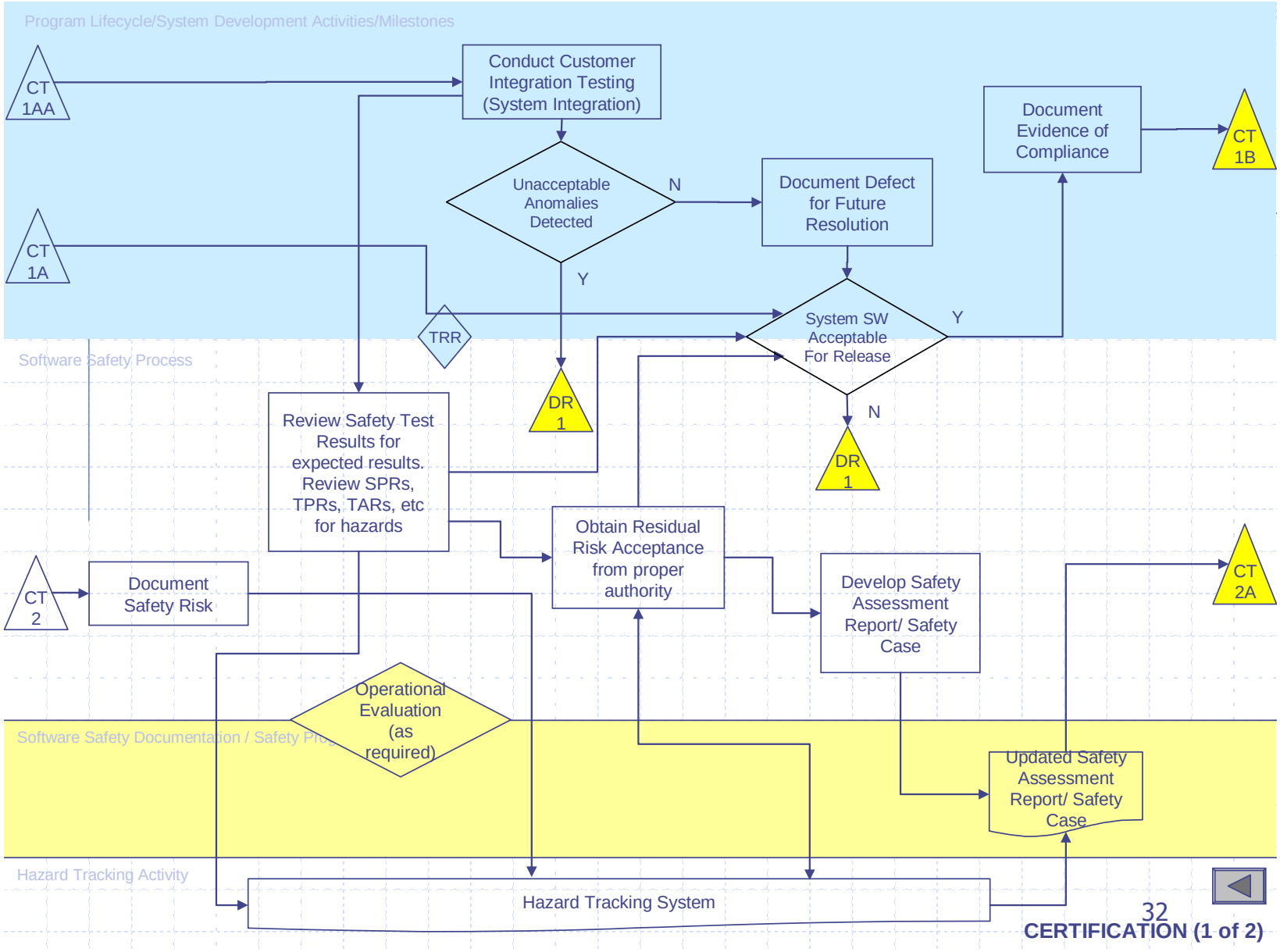
DEFECT RESOLUTION (1 of 2)



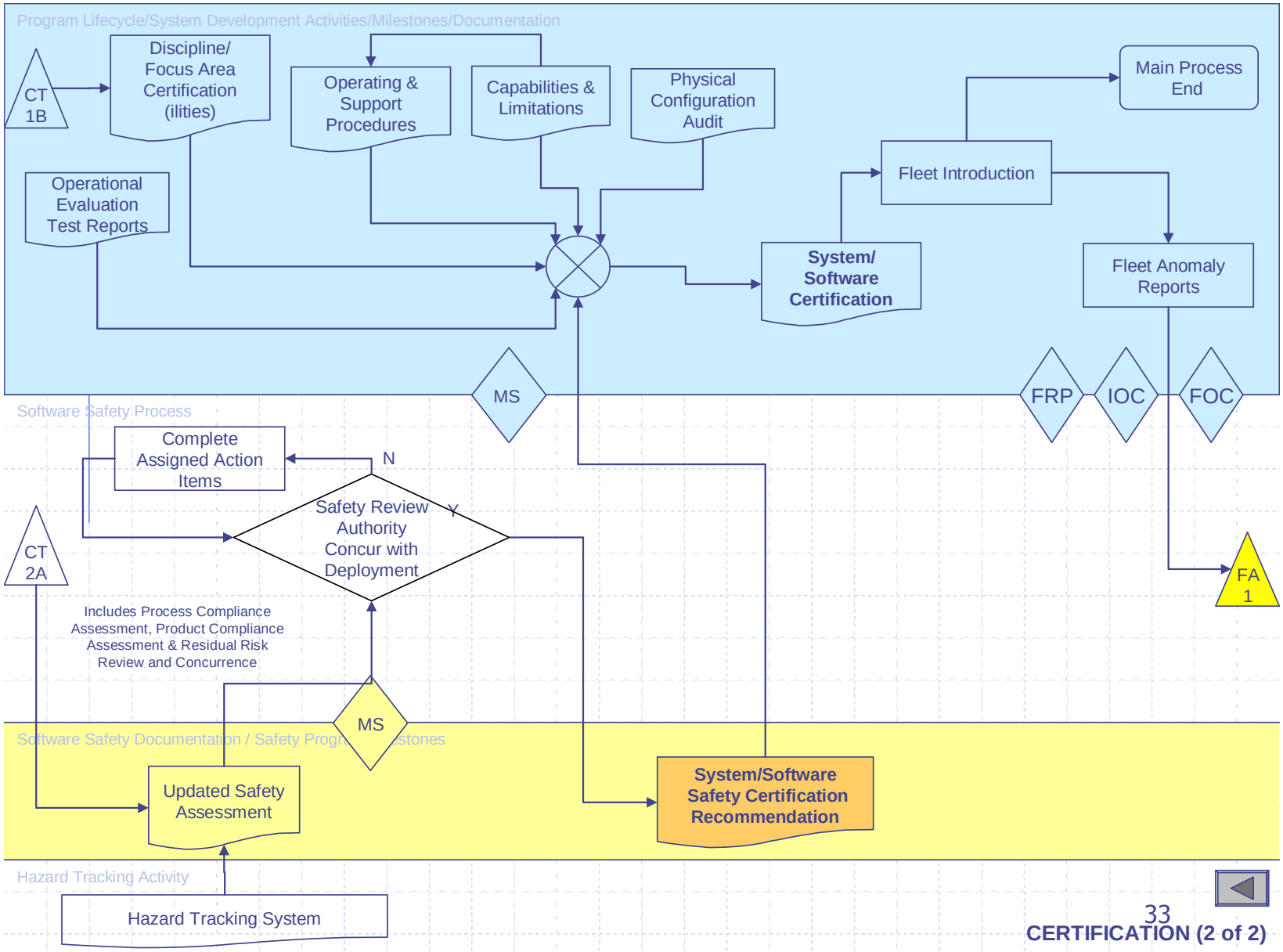
30
DEFECT RESOLUTION (2 of 2)

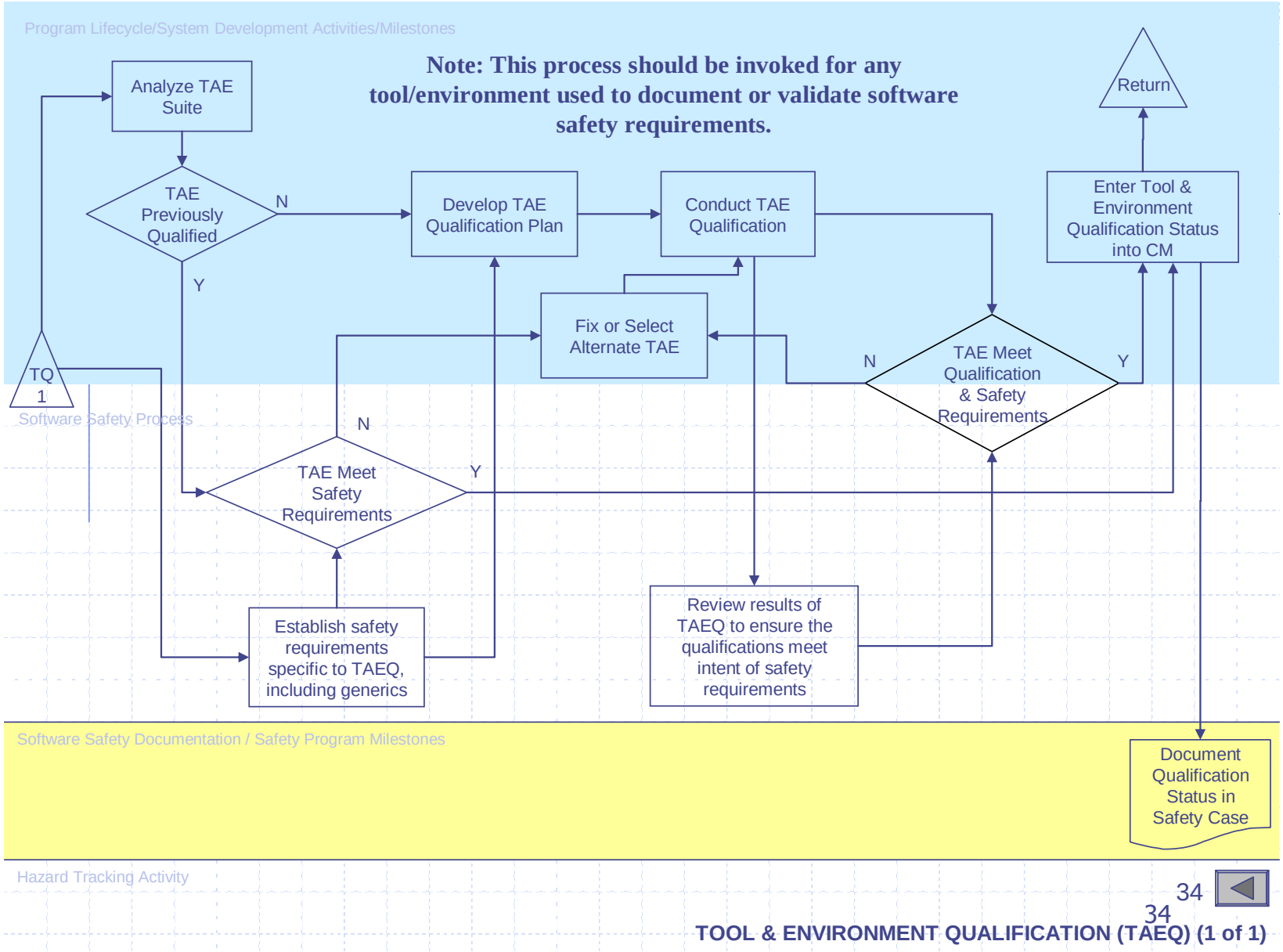


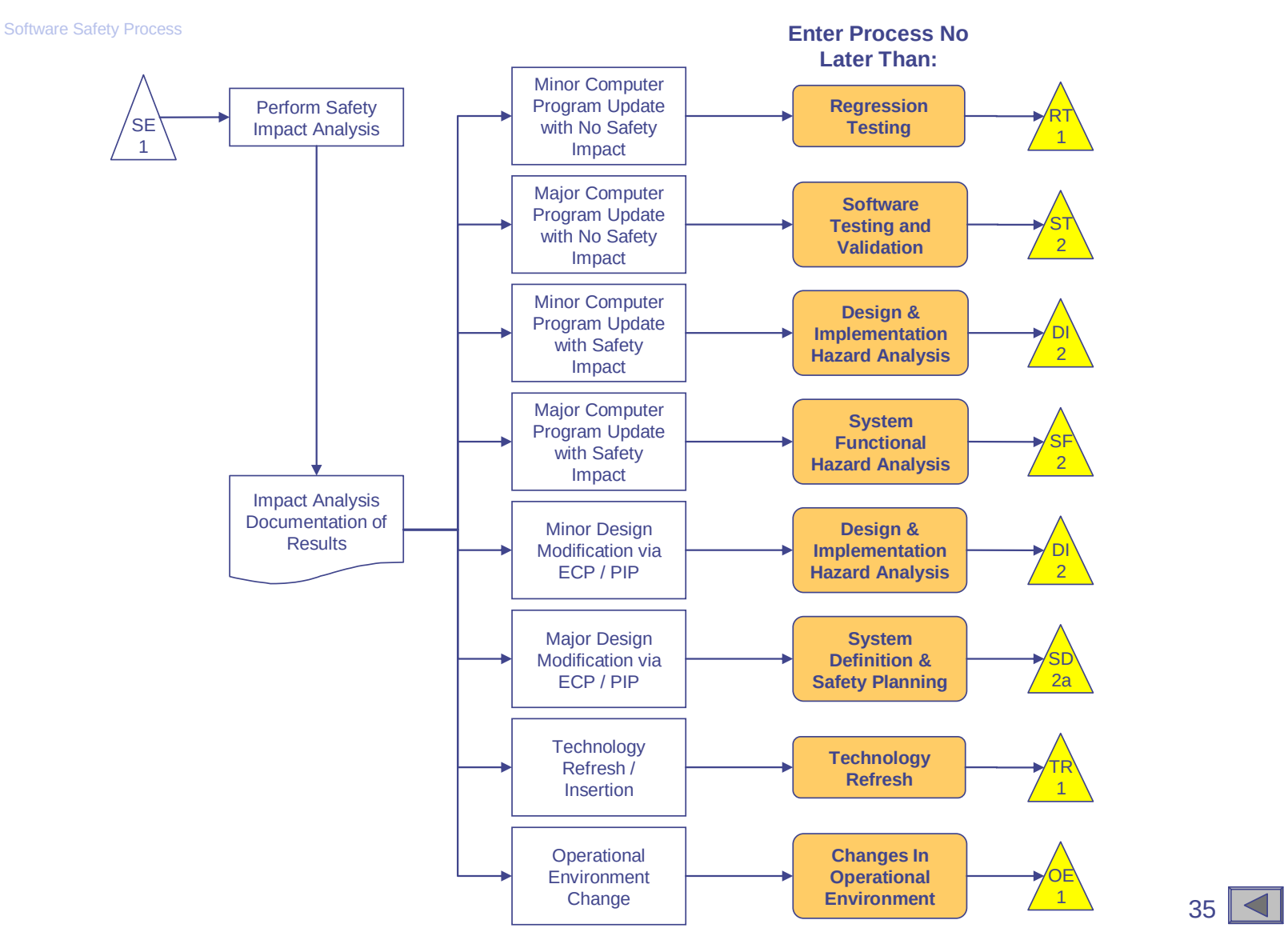
31
REGRESSION TESTING (1 of 1)

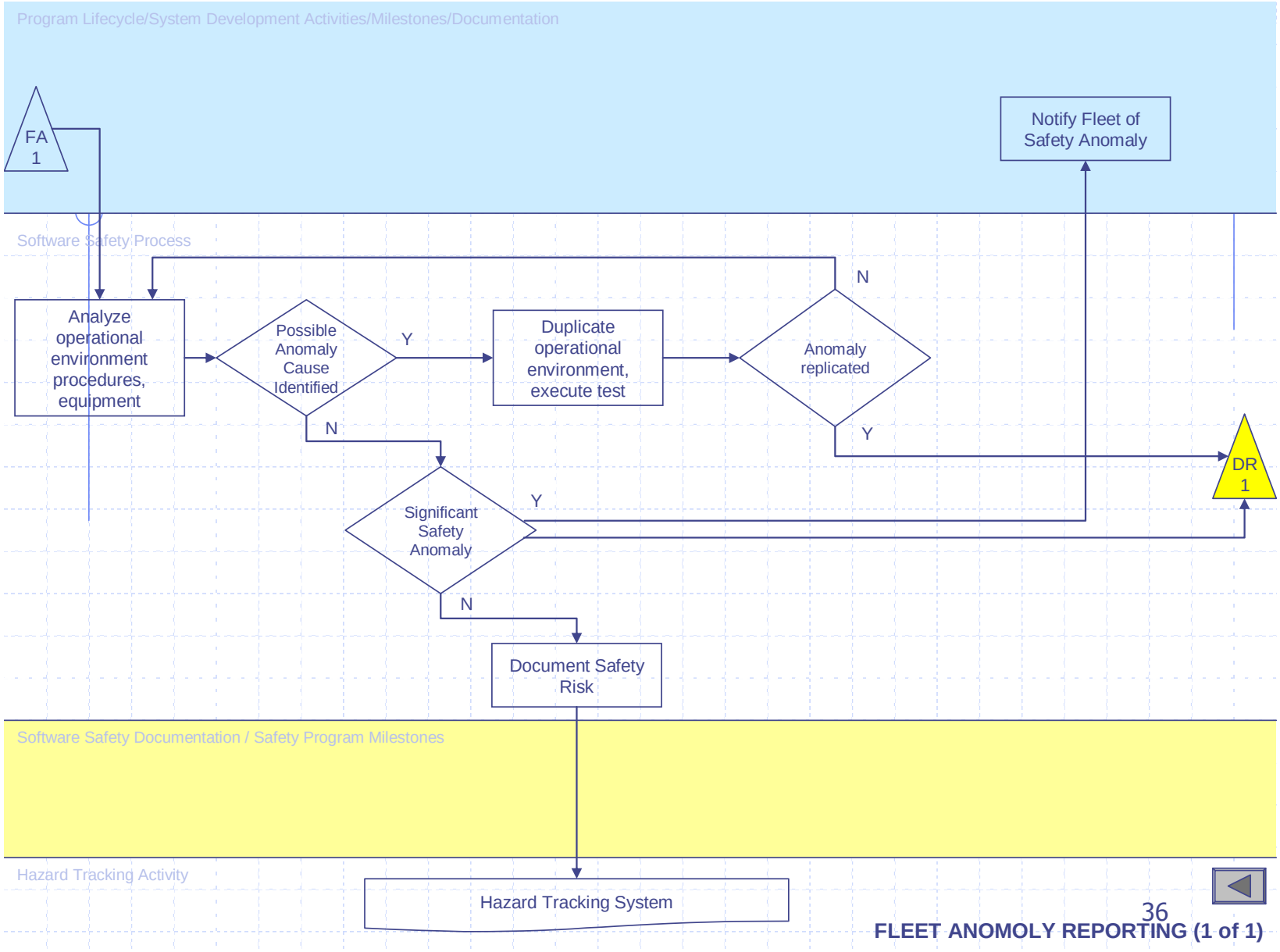


32
CERTIFICATION (1 of 2)

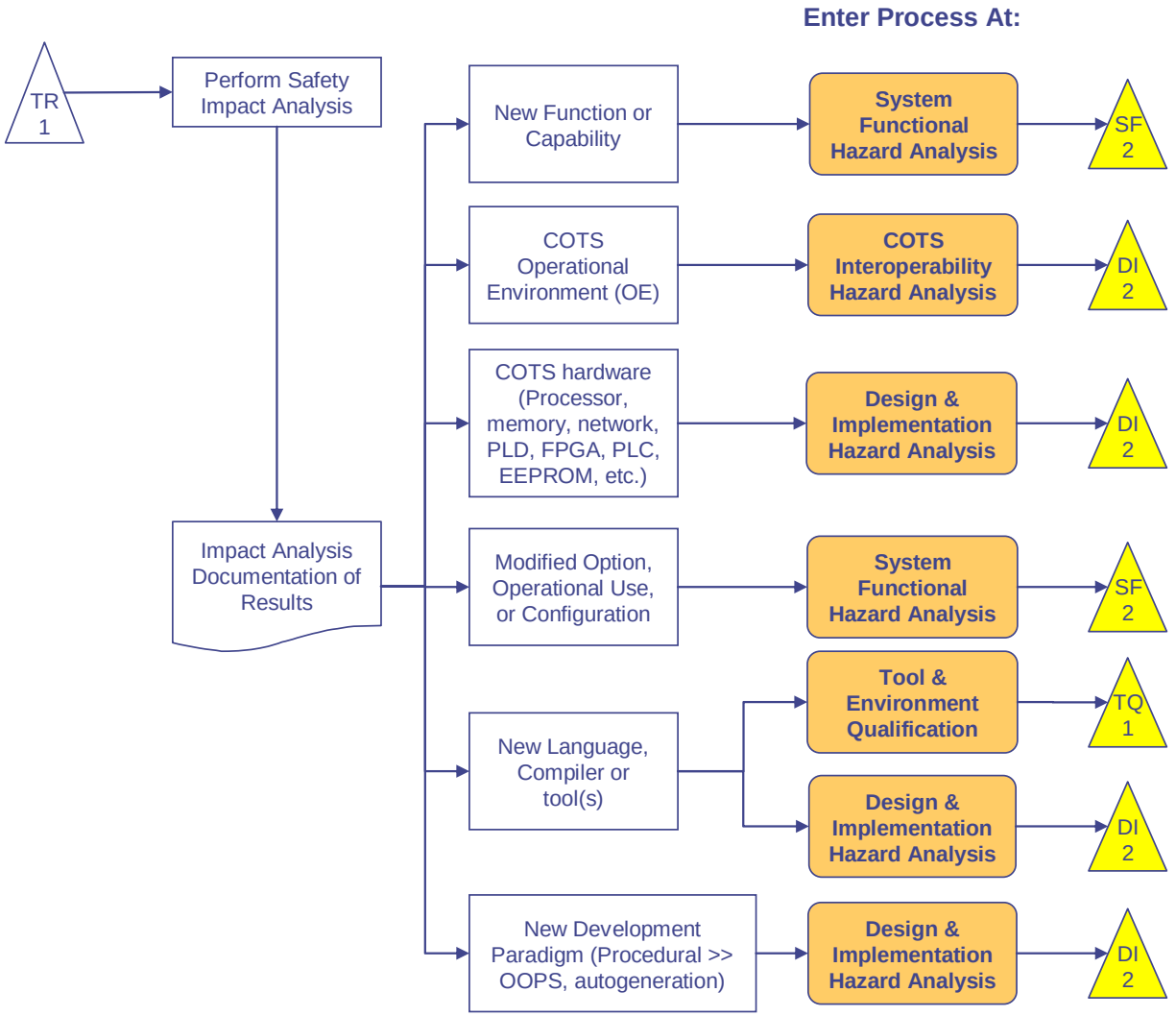

Program Lifecycle/System Development Activities/Milestones/Documentation
CT 1B
Discipline/ Focus Area Certification (ilities)
Operating & Support Procedures
Capabilities & Limitations
Physical Configuration Audit
Main Process End
Operational Evaluation Test Reports
Fleet Introduction
System/ Software Certification
Fleet Anomaly Reports
MS
FRP
IOC
FOC
Software Safety Process
Complete Assigned Action Items
N
Safety Review Authority Concur with Deployment
Y
CT 2A
FA 1
Includes Process Compliance Assessment, Product Compliance Assessment & Residual Risk Review and Concurrence
MS
Software Safety Documentation / Safety Program Milestones
Updated Safety Assessment
System/Software Safety Certification Recommendation
Hazard Tracking Activity
Hazard Tracking System
33
CERTIFICATION (2 of 2)


Program Lifecycle/System Development Activities/Milestones
Note: This process should be invoked for any tool/environment used to document or validate software safety requirements.
Analyze TAE Suite
TAE Previously Qualified
N
Y
Develop TAE Qualification Plan
Conduct TAE Qualification
Return
Enter Tool & Environment Qualification Status into CM
Fix or Select Alternate TAE
TAE Meet Qualification & Safety Requirements
N
Y
TQ 1
Software Safety Process
N
TAE Meet Safety Requirements
Y
Establish safety requirements specific to TAEQ, including generics
Review results of TAEQ to ensure the qualifications meet intent of safety requirements
Software Safety Documentation / Safety Program Milestones
Document Qualification Status in Safety Case
Hazard Tracking Activity
34
34
TOOL & ENVIRONMENT QUALIFICATION (TAEQ) (1 of 1)

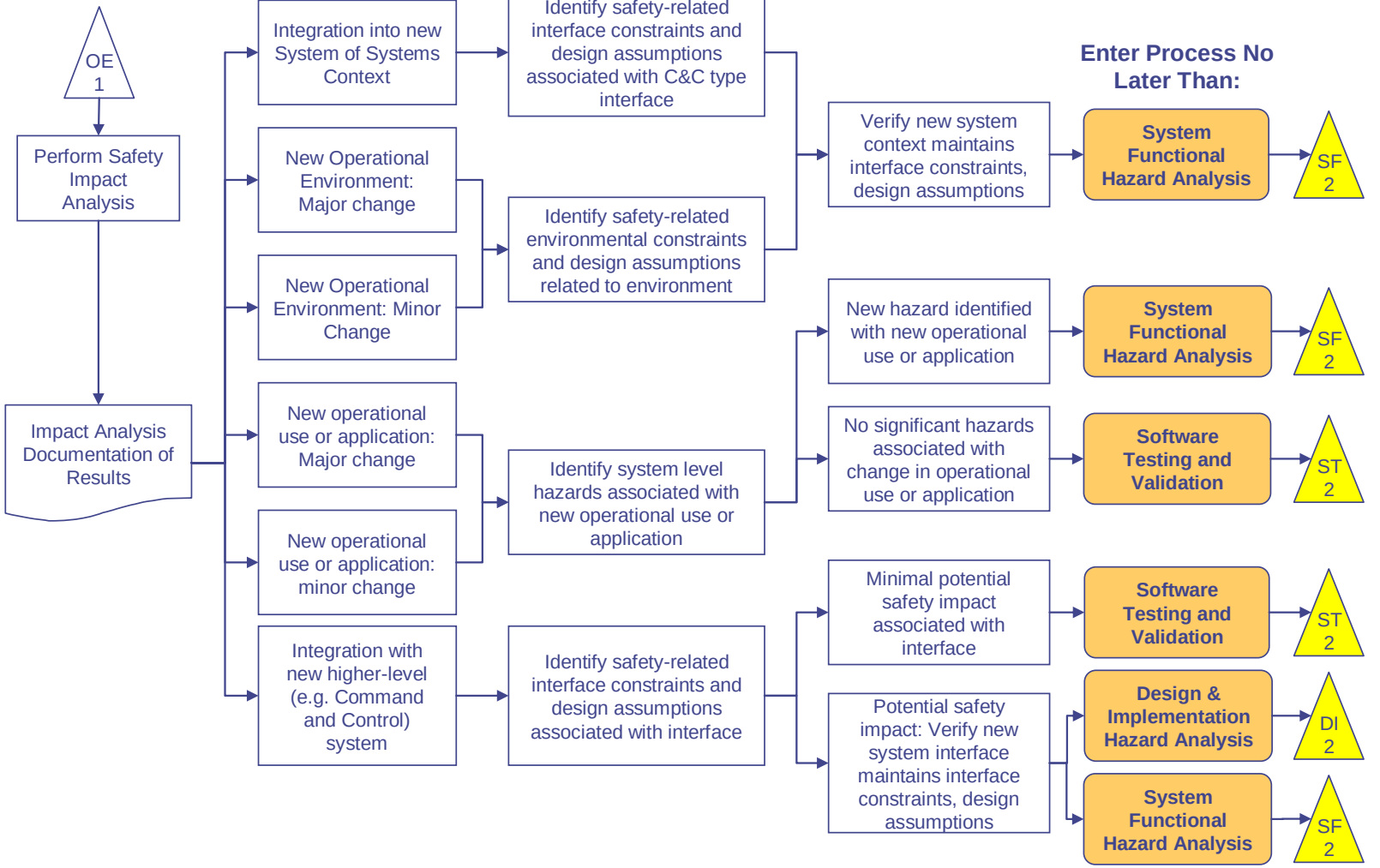

Software Safety Process
Enter Process No Later Than:
SE 1
Perform Safety Impact Analysis
Impact Analysis Documentation of Results
Minor Computer Program Update with No Safety Impact
Major Computer Program Update with No Safety Impact
Minor Computer Program Update with Safety Impact
Major Computer Program Update with Safety Impact
Minor Design Modification via ECP / PIP
Major Design Modification via ECP / PIP
Technology Refresh / Insertion
Operational Environment Change
Regression Testing
Software Testing and Validation
Design & Implementation Hazard Analysis
System Functional Hazard Analysis
Design & Implementation Hazard Analysis
System Definition & Safety Planning
Technology Refresh
Changes In Operational Environment
RT 1
ST 2
DI 2
SF 2
DI 2
SD 2a
TR 1
OE 1


Program Lifecycle/System Development Activities/Milestones/Documentation
FA 1
Notify Fleet of Safety Anomaly
Software Safety Process
N
Analyze operational environment procedures, equipment
Possible Anomaly Cause Identified
Y
Duplicate operational environment, execute test
Anomaly replicated
N
Y
DR 1
Significant Safety Anomaly
Y
N
Document Safety Risk
Software Safety Documentation / Safety Program Milestones
Hazard Tracking Activity
Hazard Tracking System
36
FLEET ANOMOLY REPORTING (1 of 1)

Software Safety Process



Software Safety Process



38

CHANGES IN OPERATIONAL ENVIRONMENT OR USE (1 of 1)

Level of Rigor Determination

Software Criticality Matrix

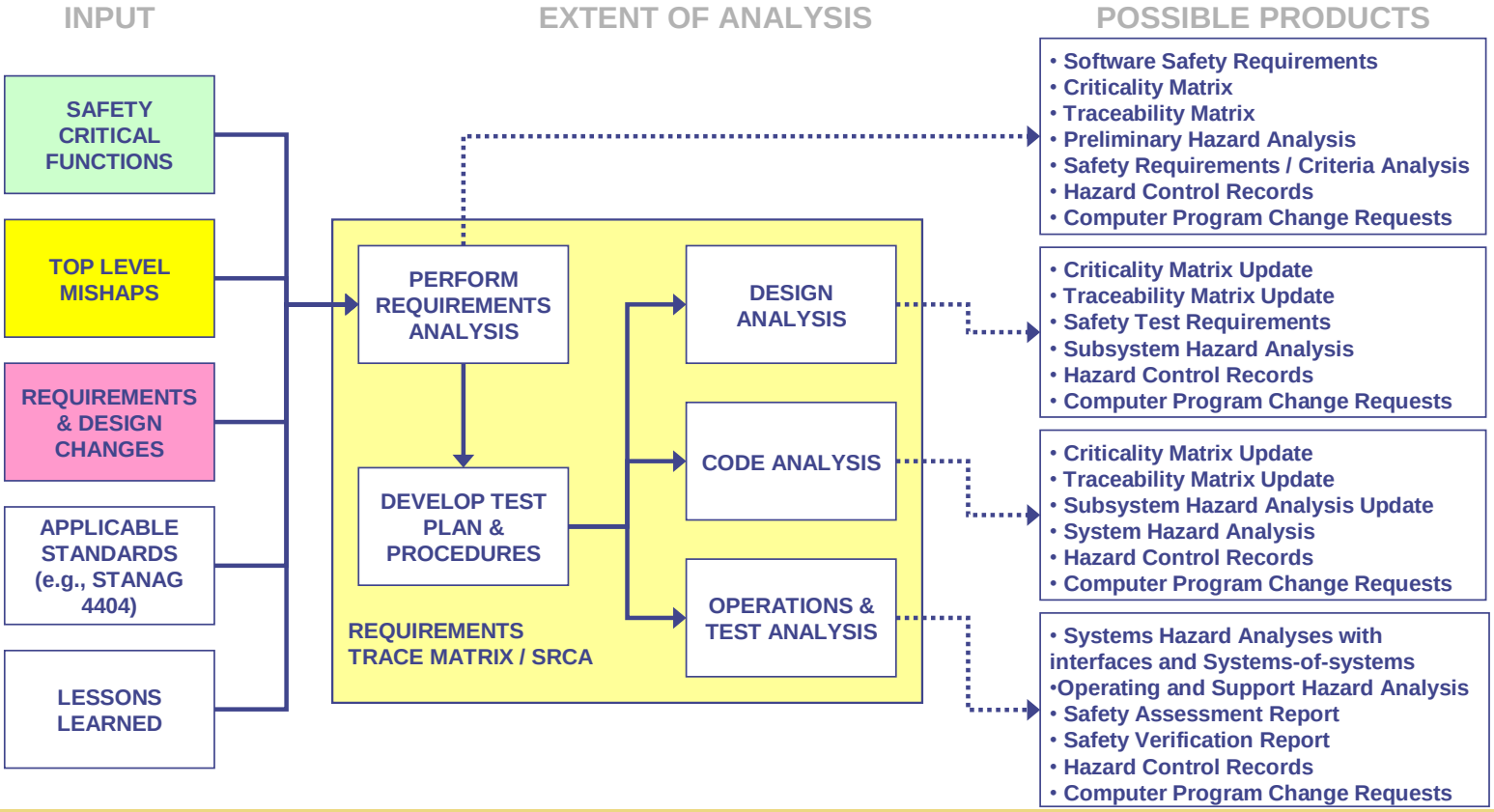
SOFTWARE CONTROL CATEGORY	MISHAP SEVERITY POTENTIAL			
	Catastrophic	Critical	Marginal	Negligible
Autonomous	SHRI 1	SHRI 1	SHRI 2	SHRI 4
Semi-Autonomous	SHRI 1	SHRI 2	SHRI 3	SHRI 4
Semi-Autonomous with Redundant Back-Up	SHRI 2	SHRI 3	SHRI 4	SHRI 4
Influential	SHRI 3	SHRI 3	SHRI 4	SHRI 4
No Safety Involvement	No Safety Analysis Required.			

SHRI 1	High Risk – Safety verification requires requirements analysis, design analysis, code analysis and in-depth safety-specific testing
SHRI 2	Serious Risk – Requires requirements analysis, design analysis & in-depth safety specific testing
SHRI 3	Medium Risk – Requires requirements analysis & safety specific testing
SHRI 4	Low Risk – Requires high level safety testing

Hyperlinks: SR 2 | CI 2 | DI 2 | DI 2A

Level of Rigor Determination

(SHRI = 1)

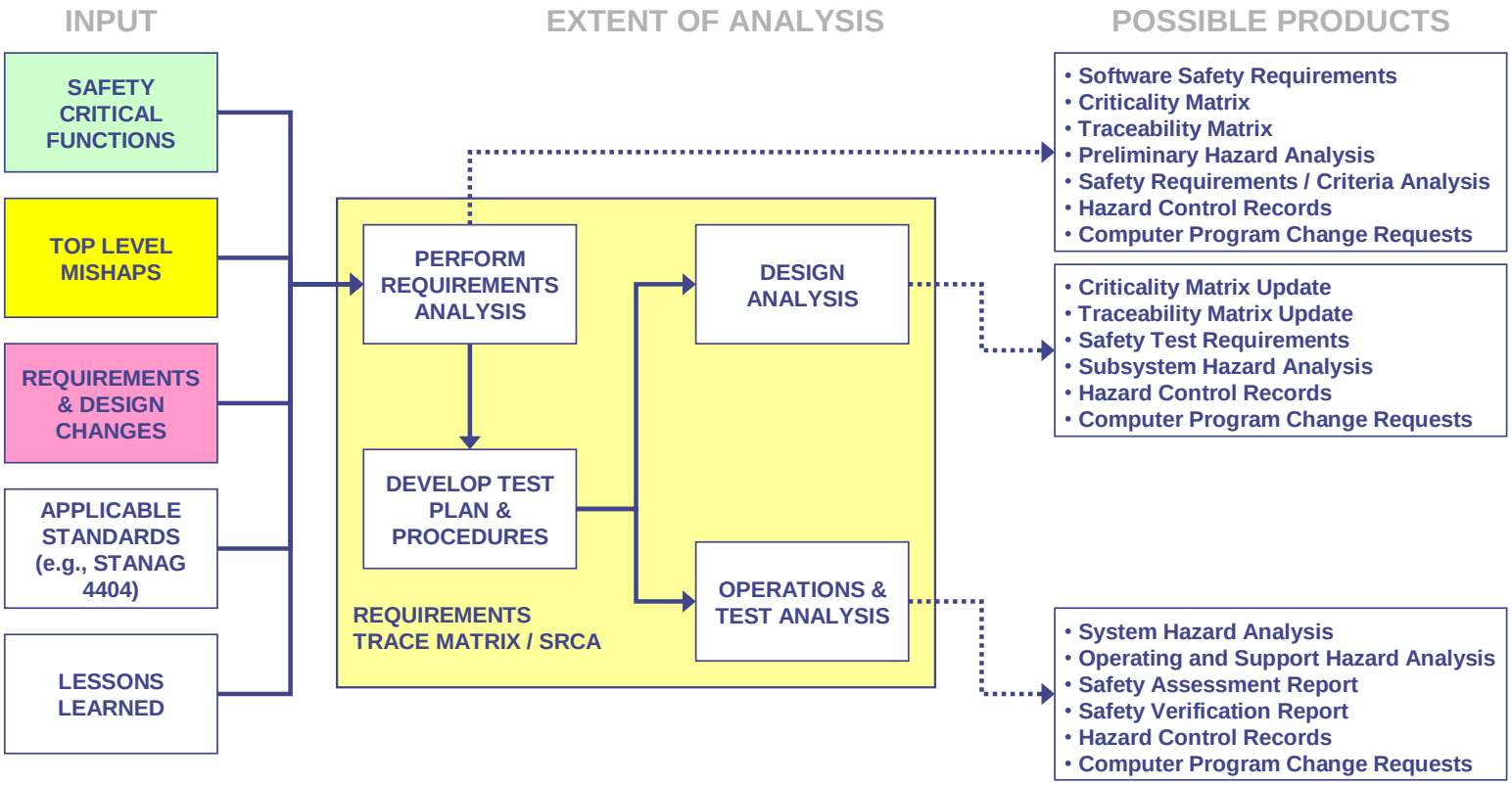


Hyperlinks: SR 2 | CI 2 | DI 2 | DI 2A

Software Criticality Matrix

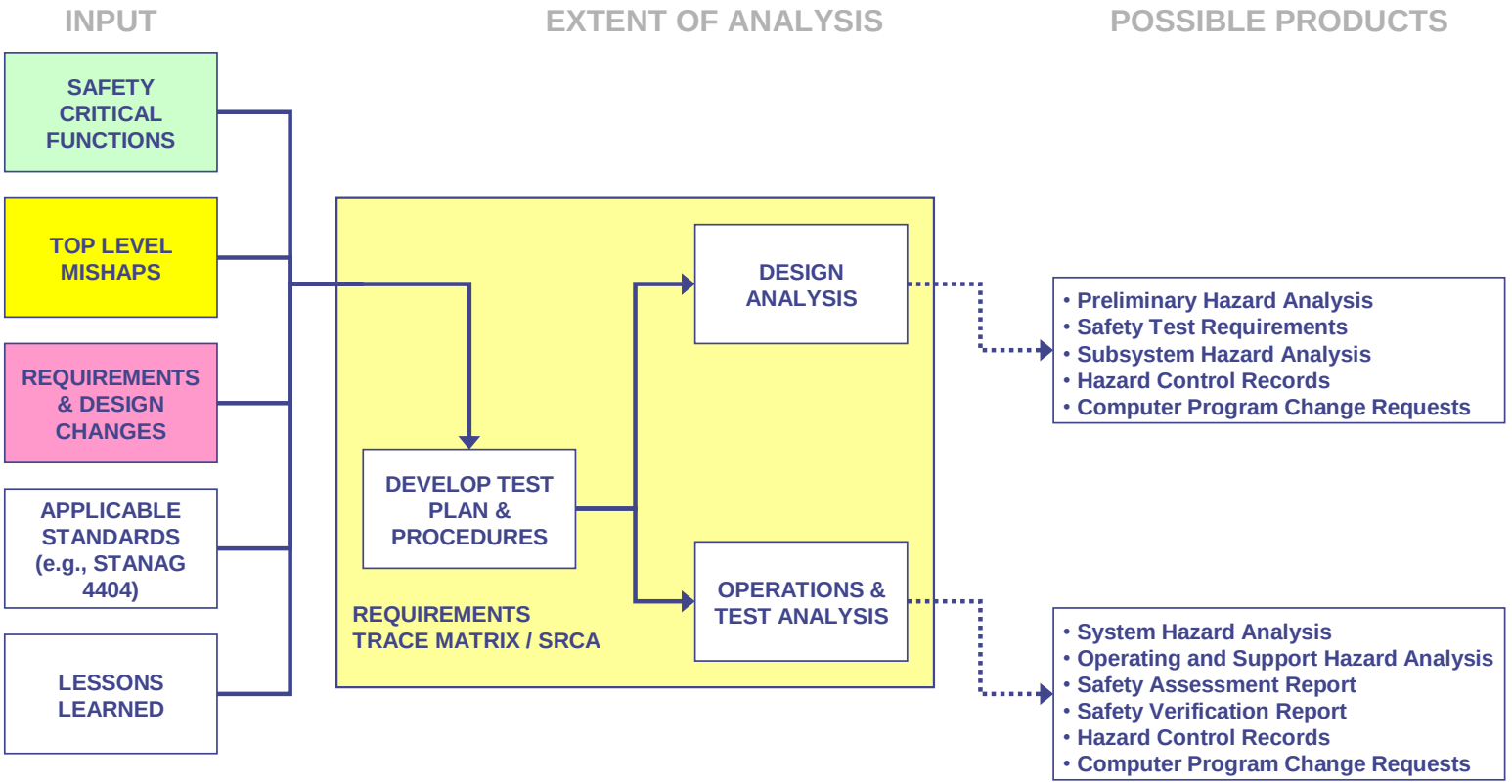
40

Level of Rigor Determination
(SHRI = 2)

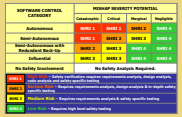


Hyperlinks: SR 2 | CI 2 | DI 2 | DI 2A | Software Criticality Matrix

Level of Rigor Determination
(SHRI = 3)



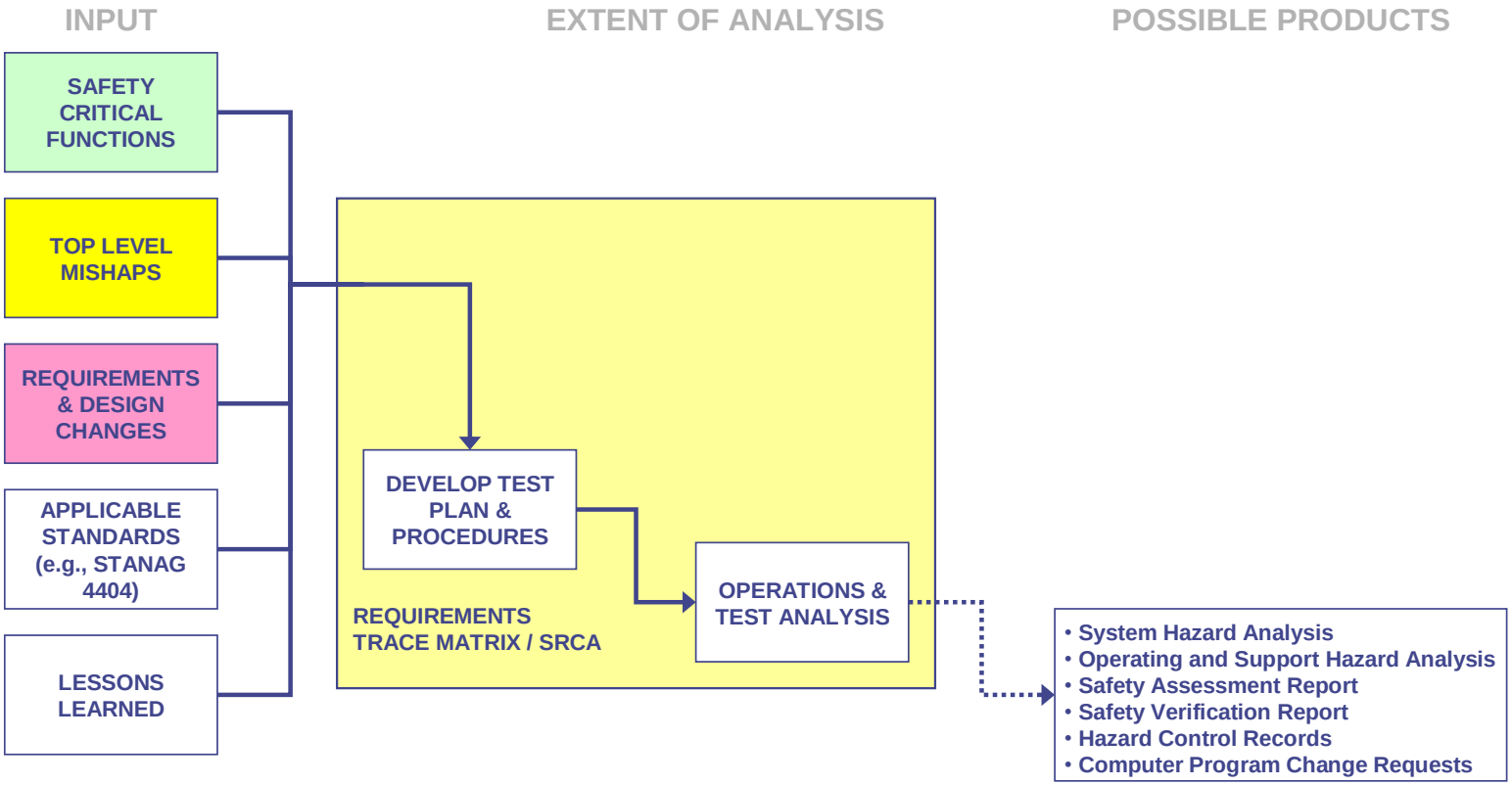
Hyperlinks:



Software Criticality Matrix

42

Level of Rigor Determination
(SHRI = 4)



Hyperlinks: SR 2 | CI 2 | DI 2 | DI 2A | Software Criticality Matrix

AOP-52(B)(1)