19702
MIL-STD-882E Software System Safety Tutorial

An Approach for Focused and Effective Level of Rigor (LoR)

Stuart A. Whitford
Booz Allen Hamilton

20th Annual NDIA Systems Engineering Conference
Springfield, VA
23 October 2015

Agenda

- MIL-STD-882E Requirements for Software Safety
- DoD Guidance for Software Safety
- Software System Safety Hazard Analysis
- Functional Hazard Analysis (FHA) for Software
- In-Depth Safety-Specific Testing
- Requirements Analysis
- Architecture Analysis
- Design Analysis
- Code Analysis
- Wrap Up

Learning Objectives

Gain an understanding of:

- A framework for performing and documenting MIL-STD-882E-required software safety Level of Rigor (LoR)

NOTE: Blue font is used in these slides to highlight significant terms or statements.

Learning Objectives

Gain an understanding of :

- A framework for performing and documenting MIL-STD-882E-required software safety Level of Rigor (LoR)

NOTE: This framework will NOT be a detailed step-by-step process of exactly how to perform each analysis on every system

Learning Objectives

Gain an understanding of:

- A framework for performing and documenting MIL-STD-882E-required software safety Level of Rigor (LoR)
- How to focus analysis of software requirements and architecture on the command and control of Safety-Significant Functions

Learning Objectives

Gain an understanding of:

- A framework for performing and documenting MIL-STD-882E-required software safety Level of Rigor (LoR)
- How to focus analysis of software requirements and architecture on the command and control of Safety-Significant Functions
- How to focus analyses of the design and code on Safety-Critical Decision Points

Learning Objectives

Gain an understanding of:

- A framework for performing and documenting MIL-STD-882E-required software safety Level of Rigor (LoR)
- How to focus analysis of software requirements and architecture on the command and control of Safety-Significant Functions
- How to focus analyses of the design and code on Safety-Critical Decision Points
- How to derive the safety-specific test cases from the analysis

MIL-STD-882E Requirements for Software Safety

Some MIL-STD-882E Terminology

Software. A combination of associated computer instructions and computer data that enable a computer to perform computational or control functions. Software includes computer programs, procedures, rules, and any associated documentation pertaining to the operation of a computer system. Software includes new development, complex programmable logic devices (firmware), NDI, COTS, GOTS, re-used, GFE, and Government-developed software used in the system.

Software system safety. The application of system safety principles to software.

Some MIL-STD-882E Terminology

Software control category. An assignment of the degree of autonomy, command and control authority, and redundant fault tolerance of a software function in context with its system behavior.

SCC Software Control Category

SwCI Software Criticality Index

Level of rigor (LoR). A specification of the depth and breadth of software analysis and verification activities necessary to provide a sufficient level of confidence that a safety-critical or safety-related software function will perform as required.

Some MIL-STD-882E Terminology

Safety-critical. A term applied to a condition, event, operation, process, or item whose mishap severity consequence is either Catastrophic or Critical (e.g., safety-critical function, safety-critical path, and safety-critical component).

Safety-related. A term applied to a condition, event, operation, process, or item whose mishap severity consequence is either Marginal or Negligible.

Safety-significant. A term applied to a condition, event, operation, process, or item that is identified as either safety-critical or safety-related.

Some MIL-STD-882E Terminology

Safety-critical function (SCF). A function whose failure to operate or incorrect operation will directly result in a mishap of either Catastrophic or Critical severity.

SSF	Safety-Significant Function
SSSF	Safety-Significant Software Function

Requirements for Software Safety

[from MIL-STD-882E]

4.1 General. When this Standard is required in a solicitation or contract, but no specific tasks are included, only Sections 3 and 4 apply. The definitions in 3.2 and all of Section 4 delineate the minimum mandatory definitions and requirements for an acceptable system safety effort for any DoD system.

. . .

4.3.2 Identify and document hazards. Hazards are identified through a systematic analysis process that includes system hardware and software, system interfaces (to include human interfaces) . . .

Requirements for Software Safety

[from MIL-STD-882E]

4.4 Software contribution to system risk. The assessment of risk for software, and consequently software-controlled or software-intensive systems, cannot rely solely on the risk severity and probability. . . Therefore, another approach shall be used for the assessment of software's contributions to system risk that considers the potential risk severity and the degree of control that software exercises over the hardware.

Severity Categories

Description	Severity Category	Mishap Result Criteria
Catastrophic	1	Could result in one or more of the following: death . . . or monetary loss equal to or exceeding $10M.
Critical	2	Could result in one or more of the following: permanent partial disability, injuries or . . . monetary loss equal to or exceeding $1M but less than $10M.
Marginal	3	Could result in one or more of the following: injury . . . resulting in one or more lost work day(s) . . . or monetary loss equal to or exceeding $100K but less than $1M.
Negligible	4	Could result in one or more of the following: injury or occupational illness not resulting in a lost work day . . or monetary loss less than $100K.

Software Control Categories

Name	Level	Description
Autonomous (AT)	1	Software functionality that exercises autonomous control authority . . . ***without the possibility of predetermined safe detection and intervention*** **. . .**
Semi-autonomous (SAT)	2	Software functionality that exercises control . . . allowing time for predetermined ***safe detection and intervention by independent safety mechanisms*** **. . .**
Redundant Fault Tolerant (RFT)	3	Software functionality that issues commands . . . ***requiring a control entity to complete the command function*** **. . .**
Influential (INF)	4	Software ***generates information*** of a ***safety-related*** nature used to make decisions by the operator **. . .**
No Safety Impact (NSI)	5	Software functionality that does not possess command or control authority . . . and does not provide safety-significant information **. . .**

Booz | Allen | Hamilton

Software Safety Criticality Matrix

Severity \\ Control	Catastrophic (1)	Critical (2)	Marginal (3)	Negligible (4)
1 (AT)	SwCI 1	SwCI 1	SwCI 3	SwCI 4
2 (SAT)	SwCI 1	SwCI 2	SwCI 3	SwCI 4
3 (RFT)	SwCI 2	SwCI 3	SwCI 4	SwCI 4
4 (INF)	SwCI 3	SwCI 4	SwCI 4	SwCI 4
5 (NSI)	SwCI 5	SwCI 5	SwCI 5	SwCI 5

NOTE: The Influential (INF) SCC only applies to the generation of 'safety-related' information for the operator.

Software Safety Levels of Rigor

SwCI	Level of Rigor Tasks
SwCI 1	Program shall perform *analysis of requirements, architecture, design, and code; and conduct in-depth safety-specific testing*.
SwCI 2	Program shall perform analysis of requirements, architecture, and design; and conduct in-depth safety-specific testing.
SwCI 3	Program shall perform analysis of requirements and architecture; and conduct in-depth safety-specific testing.
SwCI 4	Program shall conduct safety-specific testing.
SwCI 5	Once assessed by safety engineering as Not Safety, then no safety specific analysis or verification is required.

Safety Risk for Failure to Perform LoR

RELATIONSHIP BETWEEN SwCI, RISK LEVEL, LoR TASKS, AND RISK		
SwCI	Risk Level	Software LoR Tasks and Risk Assessment/Acceptance
1	High	If SwCI 1 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as HIGH . . .
2	Serious	If SwCI 2 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as SERIOUS . . .
3	Medium	If SwCI 3 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as MEDIUM . . .
4	Low	If SwCI 4 LOR tasks are unspecified or incomplete, the contributions to system risk will be documented as LOW . . .
5	Not Safety	No safety-specific analyses or testing is required.

DoD Guidance for Software Safety

MIL-STD-882E Guidance for Software Safety

[from Tasks 102 and 103 – System Safety Program Plan and Hazard Management Plan]

102/103.2.6 Hazard analysis.

. . .

i. Describe a systematic software system safety approach to:

. . .

(4) Identify and assign the Software Criticality Index (SwCI) for each safety-significant software function (SSSF) and its associated requirements.

MIL-STD-882E Guidance for Software Safety

[from Task 208 – Functional Hazard Analysis]

208.2.1 . . .

g. An assessment of Software Control Category (SCC) for each Safety-significant Software Function (SSSF). Assign a Software Criticality Index (SwCI) for each SSSF mapped to the software design architecture.

JSSEH Guidance

4.2.1.4 Defining and Using the Software Criticality Matrix

... It is through this prioritization that safety-significant code can receive the appropriate robustness and level of rigor over the lifecycle, while effectively managing the critical resources of the program. The most important aspect of the activity is that the software with the highest level of control over safety-significant hardware must receive more attention or level of rigor than software with less safety risk potential. . .

JSSEH Guidance

4.2.1.4 Defining and Using the Software Criticality Matrix

... It is through this prioritization that safety-significant code can receive the appropriate robustness and level of rigor over the lifecycle, while effectively managing the critical resources of the program. The most important aspect of the activity is that the software with the highest level of control over safety-significant hardware must receive more attention or level of rigor than software with less safety risk potential. . . This methodology helps prioritize and manage the critical resources of schedule, budget, and personnel associated with the development of the system.

JS-SSA Software System Safety:

Implementation Process and Tasks Supporting MIL-STD-882E

3.5. [LoR] Allocations to Safety-Significant Functions

The allocation of SSFs to specific [LoR] categories is essential, both to ensure the provision of rigor to the functions of highest safety criticality and to ensure the management of the critical resources necessary to implement that rigor. . . [T]he accomplishment of the subtasks … must be thoroughly documented within the artifacts of the safety analysis.

Software System Safety Hazard Analysis – an Overview

Where to Focus

14-5.c. ... focus ... on hazard identification and mitigation of software causal factors, as opposed to error removal.

14-5.d. ... focus … on hazard and software causal factor identification and mitigation, as opposed to requirements perfection. [Software safety requirements should be based on mitigating software related hazards.]

NAVSEA SW020-AH-SAF-010, Section II, *Weapon System Safety Guidelines Handbook System Safety Engineering and Management*.

Software System Safety Hazard Analysis

Step 1 – Perform a software Functional Hazard Analysis (FHA)

Software System Safety Hazard Analysis

Step 1 – Perform a software Functional Hazard Analysis (FHA)

Step 2 – For each SSSF, perform (and document) all required tasks.

For each analysis task, identify:

a. Potential Causal Factors

b. Potential (or actual) Mitigations

c. Appropriate In-Depth Safety-Specific Testing for each CF and Mitigation

FHA for Software

[the beginning of a MIL-STD-882E software safety effort]

Performing the Software FHA

Step 1 – Perform a software Functional Hazard Analysis (FHA)

a. Identify each Safety-Significant Function (SSF) that has been allocated to software (a SSSF).

b. Assess the level of software control of the function (the Software Control Category, or SCC).

c. Identify associated safety requirements or design constraints.

d. For highly critical (SwCI 1) SSSFs, identify potential system or software design redundancies to lower the SwCI (and required LoR).

MIL-STD-882E Guidance for FHA

[from Task 208 – Functional Hazard Analysis]

208.1 Purpose . . . The initial FHA should be accomplished as early as possible in the Systems Engineering (SE) process to enable the engineer to . . .

- identify and document SCFs, SCIs, SRFs, and SRIs;
- allocate and partition SCFs and SRFs in the software design architecture;
- and identify requirements and constraints to the design team.

A working definition for ‘Function’

The following is a working definition we will use for the term ”software function” (somewhat modeled after a mathematical function):

> Given an input, or a set of related inputs, a software function produces one or more of the following outcomes:
>
> - An externally observable system action;
> - Externally observable digital information that can be used by a system operator or another software entity; or
> - An internal change of digital state.

NOTE: The use of ‘external’ and ‘internal’ refers to the context of the software component(s).

Naming a Function

Name each SSSF using a verb (performing an action) and a noun (object of the verb):

Examples:

- Arm the warhead
- Detonate the warhead
- Arm the booster
- Ignite the booster
- Release the missile
- Safe the booster
- Fire the weapon

Choosing the 'size' of a SSSF

Use engineering judgement to choose the best 'size' of SSSFs for effective and efficient analysis and test - too high a level puts too much functionality all in the same "analysis bucket," while too low a level breaks the analysis into too many pieces.

Some examples:

- Too high: Perform a Standard Missile engagement
- Too low: Close the K1 relay
- Good level: Arm the missile booster

Functional Failure Types
[from NAVSEA SW020-AH-SAF-0010]

Function:

1 Fails to operate

2 Operates incorrectly/erroneously

3 Operates inadvertently

4 Operates at wrong time (early)

5 Operates at wrong time (late)

6 Unable to stop operation

7 Receives erroneous data

8 Sends erroneous data

9 Conflicting data or information

Performing the Software FHA

-- Step 1a --

Step 1a. Identify each Safety-Significant Function (SSF) that has been allocated to software (a SSSF).

- Use the nine functional failure types to reason about the different ways the SSSF might fail with potential safety impact.

  Ex. – Software-allocated missile release function fails to operate after missile ignition.

- Document the level of mishap severity that might result from the functional failure.

  Note: For the weapon systems and combat systems we work with, this is most often CAT for software functional failures.

Performing the Software FHA

-- Step 1b --

Step 1b. Assess the level of software control of the function (the Software Control Category, or SCC).

To claim Semi-autonomous SCC, document how each SSSF failure is detected and what the independent safety mechanism that mitigates or controls the resulting hazard is.

To claim Redundant Fault Tolerant SCC, document what the redundancies are and how they mitigate or control each safety-significant failure type for the SSSF.

Examples of SSSF Functional Failures

- Safe weapon SSSF fails to operate
- Arm warhead SSSF operates inadvertently
- Detonate warhead SSSF operates inadvertently
- Detonate warhead SSSF operates at wrong time (early)
- Detonate warhead SSSF operates at wrong time (late)

NOTE: The SSSF hazard severity or the software control category may vary for each functional failure type.

Performing the Software FHA

-- Step 1c --

Step 1c. Identify associated safety requirements or design constraints.

The safety requirements and design constraints are mitigations for the safety-significant SSSF failures. Communicate these with the system and software engineers to ensure:

- They are included in the requirements and design (or coding standards) for the system
- There are appropriate tests (or inspections or analyses) included to validate the mitigations work to control identified safety-significant failures for the SSSF.

Examples of Safety Requirements and Design Constraints

- The Launcher shall include an independent Canister Deluge sub-system to command Canister Flooding in case of Launcher Overtemperature or Missile Restrained Firing.
- The Launcher shall only process Missile Launch-related commands if the Launcher has been placed in Tactical Mode by the Weapon Control System.
- The Launcher shall allow the selection of no more than two Missiles for Launch at the same time.

Performing the Software FHA

-- Step 1d --

Step 1d. For highly critical, SwCI 1 SSSFs, identify potential system or software design redundancies that could lower the SwCI (and required LoR).

These fault tolerant redundancies are mitigations for safety-significant SSSF failures. Communicate these with the system and software engineers to ensure:

- They are included in the requirements and design for the system

Ex. – The Boeing 777 primary flight software is implemented in three similar computation channels (triple modular redundancy), each with three dis-similar ‘computation lanes’ (written in different programming languages).

FHA Advantages
[from NAVSEA SW020-AH-SAF-010 Section III]

The following are significant advantages of the [FHA]:

a. Is easily and quickly performed.
b. Does not require considerable expertise.
c. Is relatively inexpensive, yet provides meaningful results.
d. Provides rigor for focusing on hazards associated with system functions.
e. Good tool for software safety analysis.

FHA Disadvantages
[from NAVSEA SW020-AH-SAF-010 Section III]

The following are disadvantages of the [FHA]:

a. . . . it might overlook other types of hazards, such as those dealing with hazardous energy sources or sneak circuit paths.
b. After a functional hazard is identified, further analysis is required to determine if the causal factors are possible.
c. Cannot completely replace the need for a PHA.

In-Depth Safety-Specific Testing

In-Depth Safety-Specific Testing

1. In-Depth Safety-Specific Testing should be derived from the software safety analyses

In-Depth Safety-Specific Testing

1. In-Depth Safety-Specific Testing should be derived from the software safety analyses

2. Test cases should be assigned to appropriate test events

In-Depth Safety-Specific Testing

1. In-Depth Safety-Specific Testing should be derived from the software safety analyses

2. Test cases should be assigned to appropriate test events

3. Ensure results are captured for safety evidence

Limits of Testing

[W]e can thoroughly test hardware and get out requirements and design errors [but we c]an only test a small part of potential software behavior.

•Leveson, Nancy G., "A New Approach to Ensuring Safety in Software and Human Intensive Systems." SECIE Safety in Software and Human Intensive Systems. July 2009.

Complacency may also have been involved, i.e., the common assumption that software does not fail and that software testing is exhaustive and therefore additional software checking was not needed.

•Leveson, Nancy G., "A Systems-Theoretic Approach to Safety in Software-Intensive Systems." 2004.

Limits of Testing

[O]ne of the most important limitations of software testing is that testing can show only the presence of failures, not their absence. This is a fundamental, theoretical limitation; generally speaking, the problem of finding all failures in a program is undecidable.

•Paul Ammonn, Jeff Offutt. *Introduction to Software Testing*. 2008.

Limits of Testing

We cannot test software for correctness: Because of the large number of states (and the lack of regularity in its structure), the number of states that would have to be tested to assure that software is correct is preposterous. Testing can show the presence of bugs, but, except for toy problems, it is not practical to use testing to show that software is free of design errors.

•David L. Parnas, A. John van Schouwen, and Shu PO Kwan. “Evaluation of Safety-Critical Software.” *Communications of the ACM*, June 1990.

An interview with Watts Humphrey

(the "Father of Software Quality")

Humphrey: . . . When you think about a big program, big complex system program, 2 million lines of code something like that, and you run exhaustive tests, what percentage of all the possibilities do you think you've tested? Any idea?

Booch: Oh it's going to be an embarrassingly small number probably in the less than 20, 30% would be my guess. . .

Humphrey: You're way off. Way off. I typically ask people and I get back numbers 50%, 30%, that kind of thing. I asked the people at Microsoft, the Windows people, what they thought. And then we chatted about it a bit and they said about 1%.

Booch: Oh my goodness.

Humphrey: And my reaction is they're high by several orders of magnitude. . . the number of possibilities is so extraordinary you literally couldn't do a comprehensive test in the lifetime of the universe today.

"An Interview with Watts Humphrey, Part 26: Catastrophic Software Failures and the Limits of Testing" Watts S. Humphrey and Grady Booch, Aug 16, 2010, provided by the Computer History Museum.

Purpose of Testing

Assess quality. This is a tricky objective because quality is multi-dimensional. . . For example, reliability is . . . about the number of reliability-related failures that can be expected in a period of time or a period of use. . .To make this prediction, you need a mathematically and empirically sound model that links test results to reliability. Testing involves gathering the data needed by the model. . .

Verify correctness of the product. It is impossible to do this by testing.

Assure quality. Despite the common title, quality assurance, you can't assure quality by testing. . .

Assess conformance to specification. . .

Find defects. . . the classic objective of testing. . . Generally, we look for defects in all interesting parts of the product. . .

Kaner, C. "What Is a Good Test Case?" 2003.

Purpose of Safety-Specific Testing

In-Depth Safety-Specific Testing should clearly demonstrate additional testing rigor.

Test cases should attempt to show that:

1) Causal Factor instances ***can be realized*** and
2) Identified Mitigations ***don't work as intended***

The test scenarios should include credible "load" or "stress" relevant to the SSSF.

Types of In-Depth Testing

Boundary limit testing:

- Data range limits (e.g., highest or lowest possible values of a safety-critical input, at or near zero, or near/at/over capacity limits of a data storage).
- Timing limits (e.g., at the expiration of a timer or time limit).

Robustness testing:

- Response to abnormal inputs and conditions while ensuring safe SSSF performance, e.g., high rates of new track acquisitions and drop-outs.

Fault injection testing:

- Response to faults injected during SSSF performance.

Stress testing:

- Response to credible system stress during SSSF performance.

Types of In-Depth Testing

Safe state transition testing:

- Exercise all possible state transitions during SSSF performance.

Out of sequence testing:

- Software response to out-of-sequence inputs and conditions while ensuring safe performance of the SSSF.

Out-of-range value testing:

- Assurance of safe performance of the SSSF in response to out-of-range inputs or data values.

Error and exception handling testing:

- Response to errors and exceptions during SSSF performance.

Types of In-Depth Testing

Timing analysis testing:

- For safety-critical hard real time requirements, use targeted load or stress testing of the time-critical SSSF functionality to support the findings of timing analyses performed.

Algorithm correctness testing:

- Targeted stress testing of safety-critical algorithms associated with the SSSF.

Independent test:

- Testing of prioritized SSSFs by an independent test team, if determined to be needed by analysis.

Regression testing:

- Focused regression testing of SwCI 1 or 2 SSSF as determined from changes to related functionality.

Examples of In-Depth Testing

- Script a “Restrained Firing” in a Launcher followed immediately by a communication failure and “hand-off” of the Launcher to the alternate Launch Controller:
  - See if all missile launches in the Launcher are “safed,” as required after a Restrained Firing
- Script a second Launch Inhibit Command just as the first Launch Inhibit Command timeout is occurring, which should clear the first Launch Inhibit condition
- Script a “failover” of the primary Launch Controller to the alternate Launch Controller just after a Launch Inhibit Command has been processed.
- Script a “Restrained Firing” during a Max Launch test scenario.

Requirements Analysis

Safety Requirements Analysis (SRA)

The safety requirements are the driving force behind a designer's ability to design safety into a system and its subsystems. . .

From a safety perspective, there are three categories of SSRs [software safety requirements] . . . contributing software safety requirements (CSSR), generic software safety requirements [GSSR], and mitigating software safety requirements (MSSR).

[from the *Joint Software System Safety Engineering Handbook* (2010)]

Generic Software Safety Requirements (GSSRs)

GSSRs are requirements that have been documented over the years under the heading of lessons learned and best practices. . . The requirements themselves are not safety specific and may not yet be tied to a specific system hazard.

[from the *Joint Software System Safety Engineering Handbook* (2010)]

Some Example GSSRs

- The Launcher software shall adhere to all MISRA C++ guidelines for safety-critical software, with the exception of those documented, with rationale for non-compliance, in Table X.
- The Launcher software shall not perform dynamic memory allocation, except during program Initialization.
- The Launcher software shall not use C++ templates for any safety-significant software data objects or functions.

Contributing Software Safety Requirements (CSSRs)

The CSSRs are requirements that should already exist in the specifications and were likely authored by someone other than a safety engineer. CSSRs are related to the performance of the system to accomplish its intended function or mission. These requirements are not present for the mitigation or control of a hazard; in fact, they will often contribute to the existence of a hazard. An example of a CSSR is “Fire the Weapon.” . . .

[from the *Joint Software System Safety Engineering Handbook* (2010)]

Some Example CSSRs

- The Launcher shall power up the Missile for preparation to launch on the receipt of a valid Missile Select Command.
- The Launcher shall arm the Missile's First Stage Booster on successful completion of Launch Preparation.
- The Launcher shall apply Ignition Power on detection of all Missile-Launcher Ready to Launch conditions.
- The Missile shall arm the Warhead on detection of Safe Separation from the Launch Platform.

Mitigating Software Safety Requirements (MSSRs)

MSSRs are requirements derived from in-depth mishap and hazard causal analyses. . . . the safety engineer [performs] the safety analysis to determine whether the GSSRs have successfully mitigated the known causal factors of the mishaps and hazards. . .

MSSRs are usually authored by safety engineers, with input and assistance from the design engineers and domain experts associated with the design or subsystem being analyzed. These MSSRs must be added to the specifications . . .

[from *Joint Software System Safety Engineering Handbook* (2010)]

Some Example MSSRs

- The Launcher Deluge subsystem shall continuously monitor for Canister and Launcher Overtemperature and for Restrained Firing, and command Canister Deluge on those Canisters effected by the occurrence of any detected Hazards.
- The Launcher shall set a 75 second timer for the completion of each Missile Launch Sequence, and Safe any selected Missile that has not completed a Launch within that time period.

Analysis of Requirements

Assess all tagged CSSRs/MSSRs for:

- Completeness
- Potential conflict with other requirements
- Ambiguity

Example Conflicting/Ambiguous

- Potential conflicting requirements:
  - Automated train doors must open only when train is stopped and properly aligned with the platform.
  - Automated train doors must open for evacuation in the event of an emergency.

- Potential ambiguous requirement:
  - Aircraft shall inhibit thrust reversal when the aircraft is in flight.

Architecture Analysis

Some Terminology
(from the JSSSEH and other sources)

Architecture: The organizational structure of a system or component (IEEE 610.12 – 1990).

- ‘Architecture is concerned with the selection of architectural elements, their interaction, and the constraints on those elements and their interactions’ (Perry & Wolf, 1992, p. 40-52).
- ‘Architecture focuses on the externally visible properties of software “components” (Bass, Clements, & Kazman, 1998).

System Architecture: The arrangement of elements and subsystems and the allocation of functions to meet system requirements (*INCOSE Systems Engineering Handbook*).

Safety in a Control System

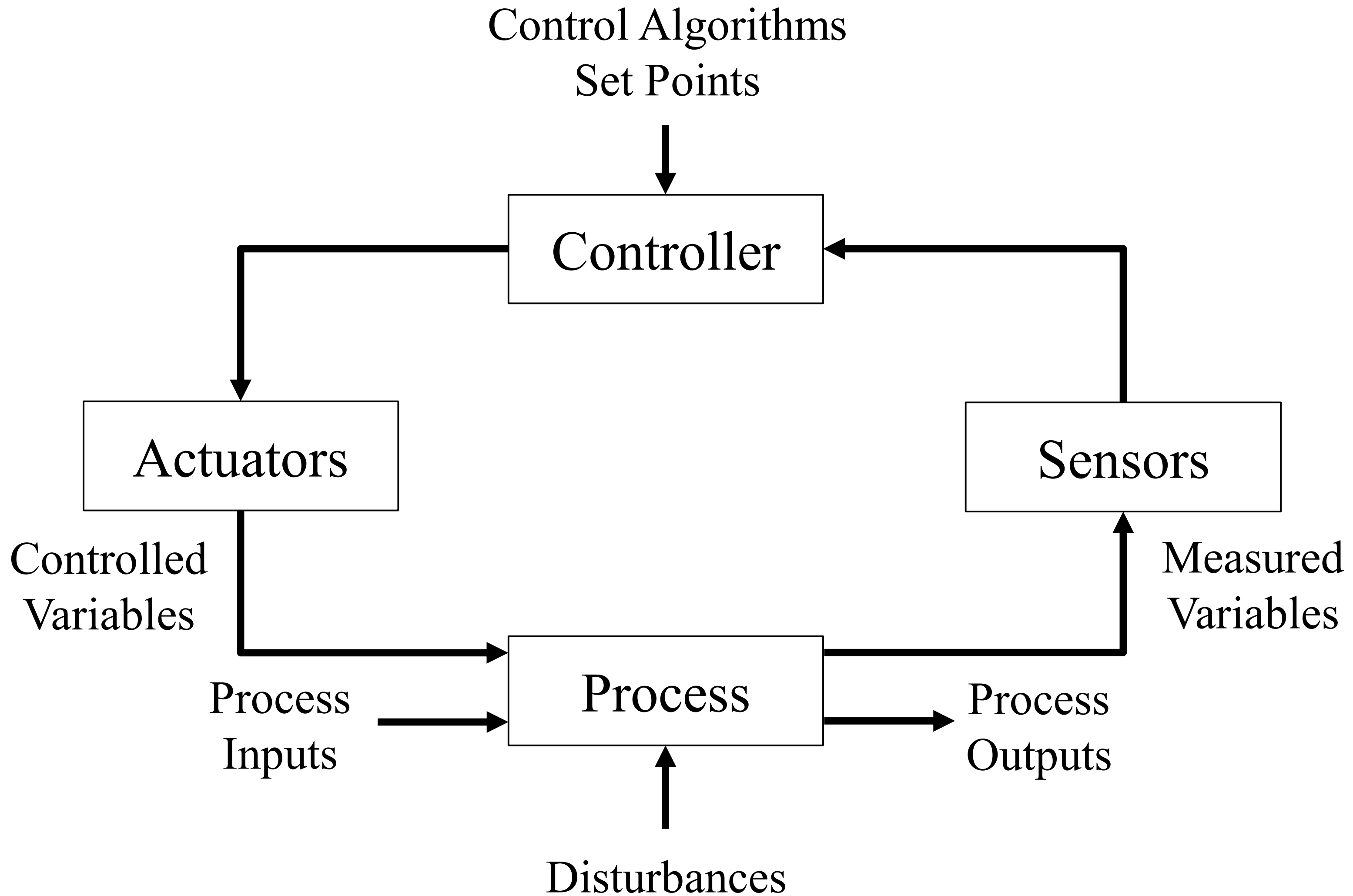
In control theory, open systems are viewed as interrelated components that are kept in a state of dynamic equilibrium by feedback loops of information and control.

. . . [A]ccidents often occur . . . as a result:

1. Incorrect or unsafe control commands are given
2. Required control actions (for safety) are not provided
3. Potentially correct control commands are provided at the wrong time (too early or too late), or
4. Control is stopped too soon or applied too long.

Nancy G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*, 2011.

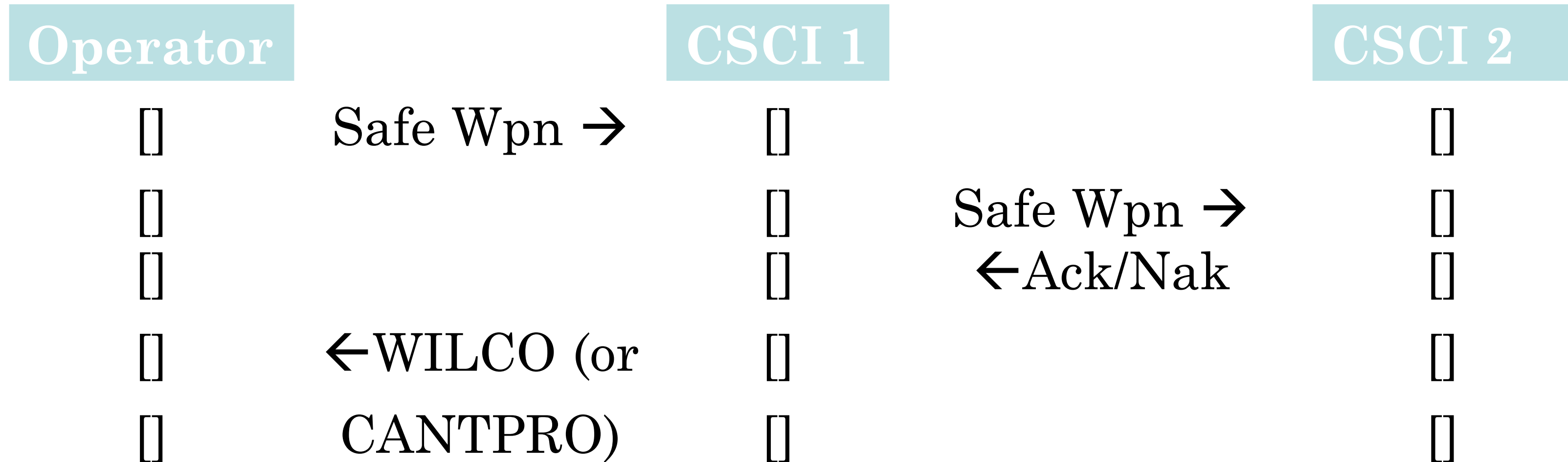
The Classic "Control Loop"



Inter-Process Architecture Analysis

- Treat each distributed SSSF as a control loop allocated across the system architecture.
- Think of ways the control or feedback signals (messages) might be corrupted, delayed or lost (potential Causal Factors).
- For each of the Causal Factors identified, think of existing or potential mitigations.

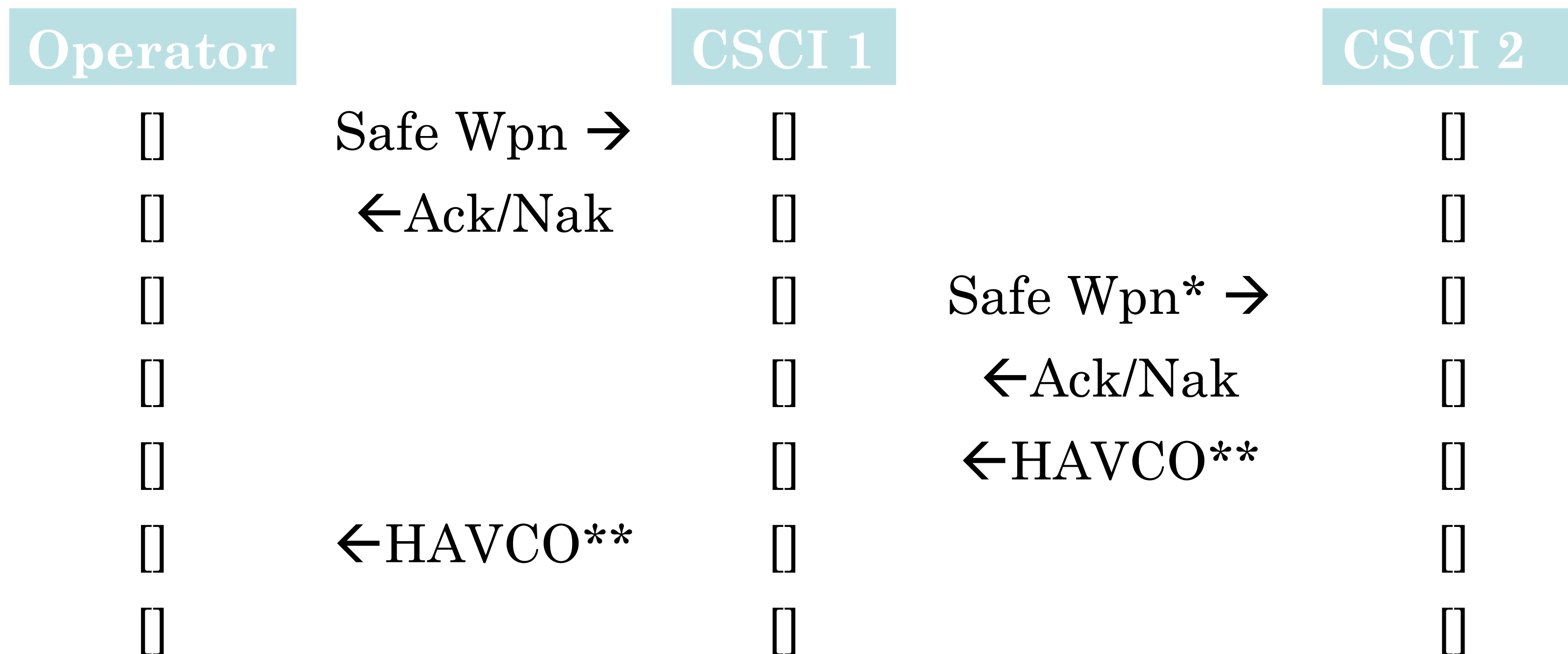
System Path/Thread Analysis for a 'Safe Weapon' SSSF



CSCI = Computer Software Configuration Item
WILCO = "Will Comply"
CANTPRO = "Cannot Process"

Ack = 'Valid' Message Acknowledge
Nak = 'Invalid' Message (Negative) Acknowledge
Safe Wpn = Safe Weapon

More Robust 'Architecture' for a 'Safe Weapon' SSSF



* CSCI 1 timer on CSCI 2's HAVCO/CANTCO response

** or CANTCO

CSCI = Computer Software Configuration Item
HAVCO = "Have Complied"
CANTCO = "Cannot Comply"

Ack = 'Valid' Message Acknowledge
Nak = 'Invalid' Message (Negative) Acknowledge
Safe Wpn = Safe Weapon

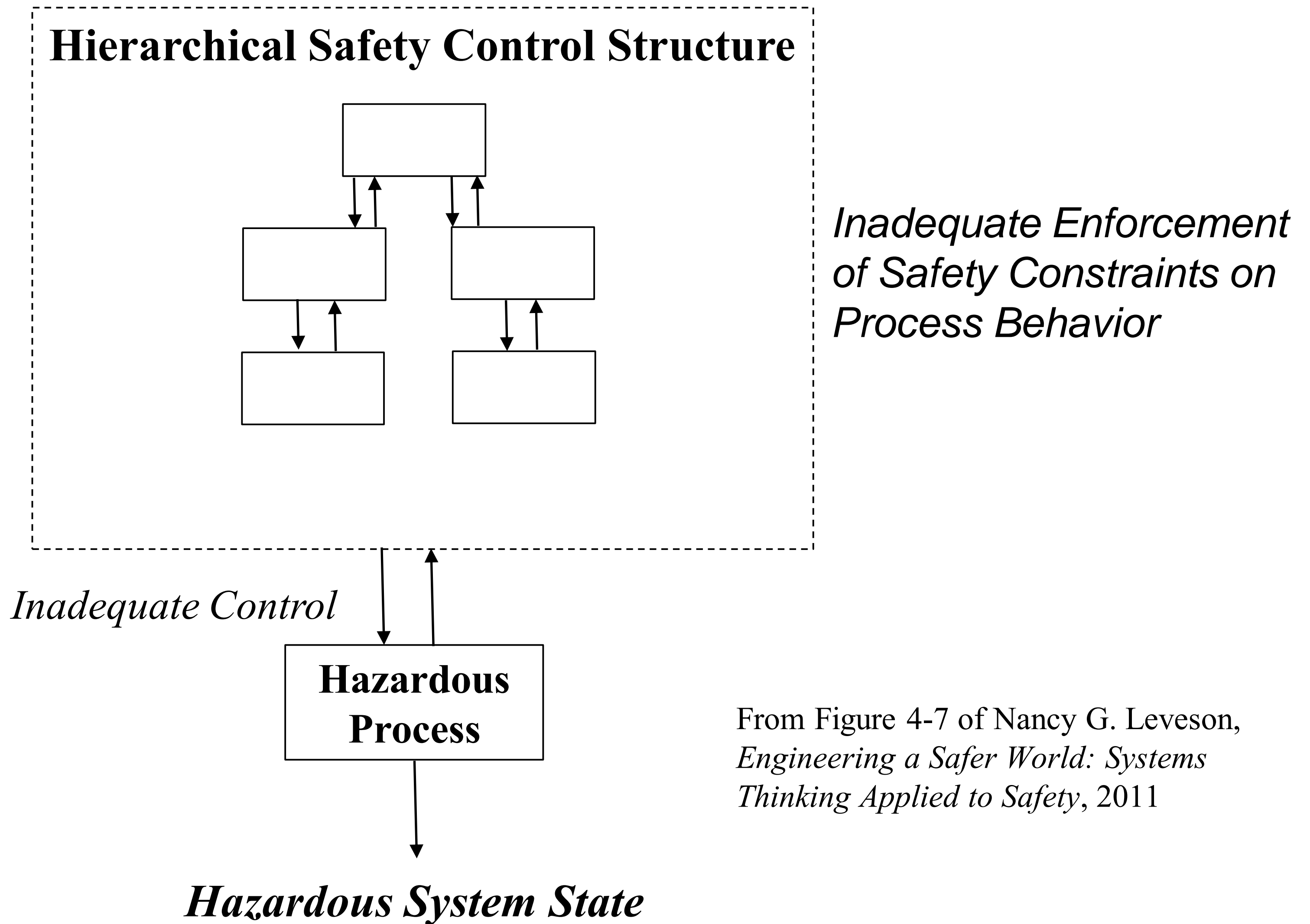
System-Theoretic Accident Model and Processes (STAMP)

In systems theory, emergent properties, such as safety, arise from the interactions among the system components. The emergent properties are controlled by imposing constraints on the behavior and the interactions among the components. Safety then becomes a control problem where the goal of the control is to enforce the system constraints. Accidents result from inadequate control or enforcement of safety-related constraints on the development, design, and operation of the system.

. . . Feedback is a basic part . . . of treating safety as a control problem. Information flow is a key in maintaining safety.

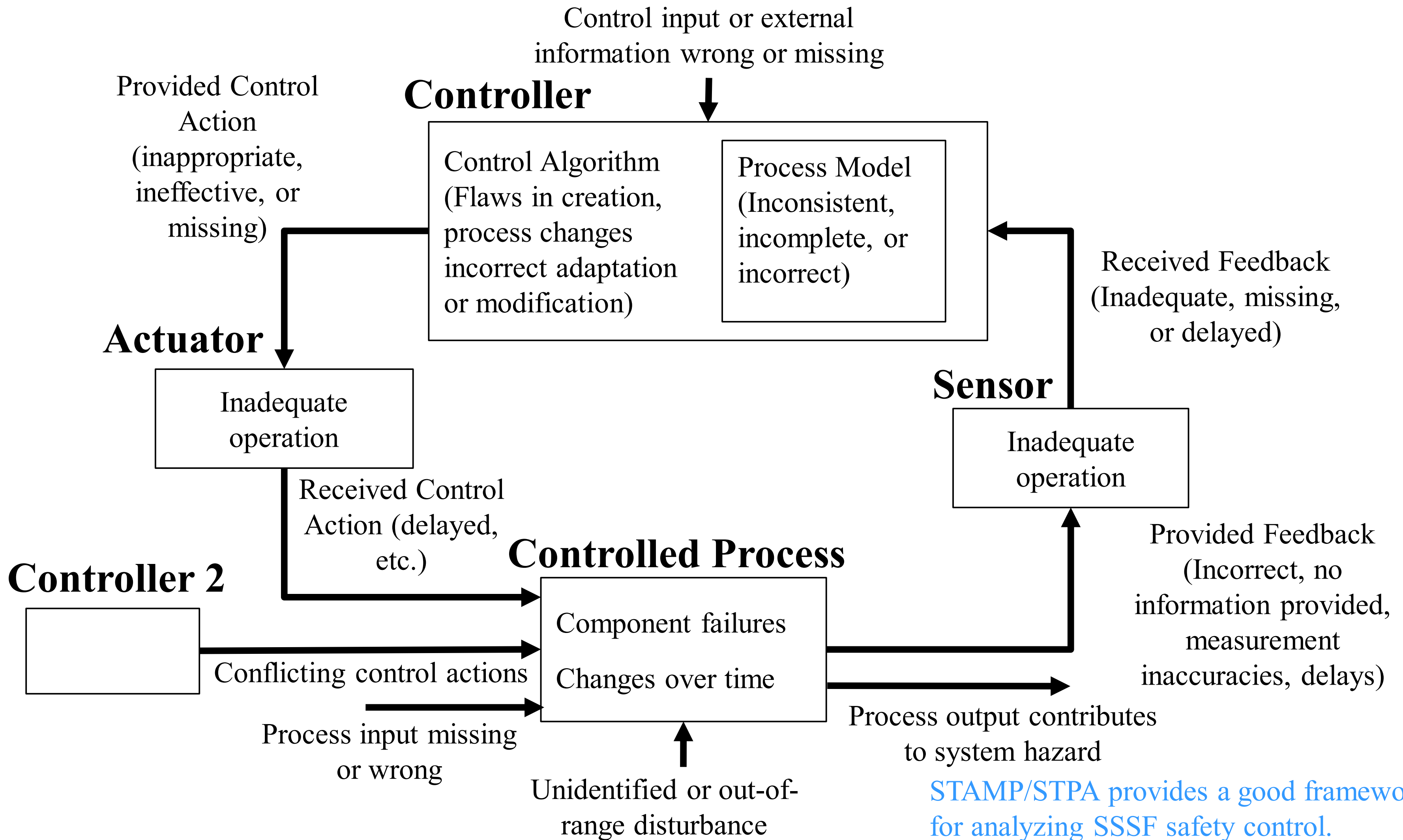
Nancy G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*, 2011.

STAMP View of System Safety



From Figure 4-7 of Nancy G. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety*, 2011

General Control Loop with Causal Factors
(from *Safety Assurance in NextGen*, NASA/CR-2012-217553)



STAMP/STPA provides a good framework for analyzing SSSF safety control.

Booz | Allen | Hamilton

Some Thoughts On STAMP, STPA, and Meeting MIL-STD-882E Required LoR

We do not assign a SwCI because in STAMP software can and should be treated in the same way as hardware, i.e., the hazards are identified along with causal scenarios leading to the hazards. Then engineers can eliminate or mitigate those causes according to standard system safety practice and design precedence . . .

Nancy G. Leveson, “STPA (System-Theoretic Process Analysis) Compliance with Army Safety Standards and Comparison with SAE ARP 4761,” a whitepaper on the compliance of STPA with MIL-STD-882E and Army AMCOM Regulation 385-17.

Some Thoughts On STAMP, STPA, and Meeting MIL-STD-882E Required LoR

We do not assign a SwCI because in STAMP software can and should be treated in the same way as hardware, i.e., the hazards are identified along with causal scenarios leading to the hazards. Then engineers can eliminate or mitigate those causes according to standard system safety practice and design precedence . . .

Nancy G. Leveson, "STPA (System-Theoretic Process Analysis) Compliance with Army Safety Standards and Comparison with SAE ARP 4761," a whitepaper on the compliance of STPA with MIL-STD-882E and Army AMCOM Regulation 385-17.

THOUGHTS:

- STAMP/STPA is a very good framework for software safety architecture analysis.

Some Thoughts On STAMP, STPA, and Meeting MIL-STD-882E Required LoR

We do not assign a SwCI because in STAMP software can and should be treated in the same way as hardware, i.e., the hazards are identified along with causal scenarios leading to the hazards. Then engineers can eliminate or mitigate those causes according to standard system safety practice and design precedence . . .

Nancy G. Leveson, “STPA (System-Theoretic Process Analysis) Compliance with Army Safety Standards and Comparison with SAE ARP 4761,” a whitepaper on the compliance of STPA with MIL-STD-882E and Army AMCOM Regulation 385-17.

THOUGHTS:

- STAMP/STPA is a very good framework for software safety architecture analysis.
- It would be a very “heavy lift” for an individual program or PFS to make the case that STAMP/STPA is replacement for required LoR.

Some Thoughts On STAMP, STPA, and Meeting MIL-STD-882E Required LoR

We do not assign a SwCI because in STAMP software can and should be treated in the same way as hardware, i.e., the hazards are identified along with causal scenarios leading to the hazards. Then engineers can eliminate or mitigate those causes according to standard system safety practice and design precedence . . .

Nancy G. Leveson, "STPA (System-Theoretic Process Analysis) Compliance with Army Safety Standards and Comparison with SAE ARP 4761," a whitepaper on the compliance of STPA with MIL-STD-882E and Army AMCOM Regulation 385-17.

THOUGHTS:

- STAMP/STPA is a very good framework for software safety architecture analysis.
- It would be a very "heavy lift" for an individual program or PFS to make the case that STAMP/STPA is replacement for required LoR.
- My experience has been that MANY software problems are not at the architecture level (and can't be eliminated there).

Design Analysis

What is “Design”?

‘Design focuses on the properties of software “components” that are not externally visible.’

[S. Whitford, 2015]

Design

What is NOT Externally Visible

What is NOT externally visible?

- The organization of elements inside each software component, e.g.:
  - Is it object oriented (Java, C++) or not (C, Assembler)?
  - Is it single threaded or multi-threaded?
- The data flow between the elements inside each software component, e.g.:
  - Message passing
  - Call parameters
  - Global data
- The control flow between the elements inside each software component, e.g:
  - Procedure/function calls
  - Semaphores/mutexes/monitors

Safety-Critical Decision Points

- Most SwCI 1 or SwCI 2 SSSFs are safety-critical because the software has command authority over a safety-critical system action.

Safety-Critical Decision Points

- Most SwCI 1 or SwCI 2 SSSFs are safety-critical because the software has command authority over a safety-critical system action.
- The software is therefore responsible for making the decision to take that action, often the release of lethal energy.

Safety-Critical Decision Points

- Most SwCI 1 or SwCI 2 SSSFs are safety-critical because the software has command authority over a safety-critical system action.
- The software is therefore responsible for making the decision to take that action, often the release of lethal energy.
- If the data used to make the safety-critical decision is corrupted or stale, the software can make the wrong decision with catastrophic results.

Safety-Critical Decision Points

- Most SwCI 1 or SwCI 2 SSSFs are safety-critical because the software has command authority over a safety-critical system action.
- The software is therefore responsible for making the decision to take that action, often the release of lethal energy.
- If the data used to make the safety-critical decision is corrupted or stale, the software can make the wrong decision with catastrophic results.
- Design (and Code) Analysis should be focused on how the software maintains, or could fail to maintain, the integrity of the data used at each Safety-Critical Decision Point in the SSSF.

SCDP: An Example

Is it safe to launch the missile?

- Was a valid Launch Command received from the Operator?
- Is the Cell Hatch fully open?
  - Does the Cell Hatch No. 1 sensor report “open”?
  - Does the Cell Hatch No. 2 sensor report “open”?
- Is the Uptake Hatch fully open?
  - Does the Uptake Hatch No. 1 sensor report “open”?
  - Does the Uptake Hatch No. 2 sensor report “open”?
- Has it been long enough since the last missile lunched?
- Is the Close-In Weapon System (CIWS) ***not*** currently firing? (Implemented as a launchInhibited Boolean (TRUE/FALSE) data item.)

'Launch Inhibited' implemented with multiple threads

Thread A:

[Launch Missile Command received]

```
boolean isMslLaunchOK ()
   If . . . hatch statuses and
            last missile launch time are "ok"
   . . . && (launchInhibited == FALSE)
        return TRUE
   else
        return FALSE
```

launchInhibited is set to TRUE when a CIWS engagement is about to start.

'Launch Inhibited' implemented with multiple threads (cont'd)

Thread B (higher priority):

[Launch Inhibit Command received]

. . .

setLaunchInhibit ()
. . . *if old timer active, cancel it*
. . . launchInhibited = TRUE
. . . *Initiate a 20s timer to clear inhibit*

Thread C (lower priority):

[20s Launch Inhibit timer expires]

. . .

clearLaunchInhibit ()
. . . launchInhibited = FALSE

Intent is to clear a *pre-existing* Launch Inhibit condition after 20 seconds.

Analysis of 'Launch Inhibited'

Thread B (higher priority):

[Receipt of new Launch Inhibit command unblocks thread]
. . .
setLaunchInhibit ()
. . . *if old timer active, cancel it (but, it's too late)*
. . . launchInhibited = TRUE
. . . *Initiate a (new) 20s timer*
[thread blocks on task completion]

Thread C (lower priority):

[Old 20s Launch Inhibit timer expires]
. . .

clearLaunchInhibit ()
. . . launchInhibited = FALSE

Timer intended to clear OLD Launch Inhibit condition clears NEW one instead!.
A data synchronization mechanism should be used to protect the shared data item.

Some Sources of Design Causal Factors

Establish the pros and cons of the design of each software component to which the SSSF is allocated and determine whether they could be Causal Factors or Mitigations for a SSSF functional failure due to an erroneous Safety-Critical Decision by the software. (It's all about the safety-critical data integrity.)

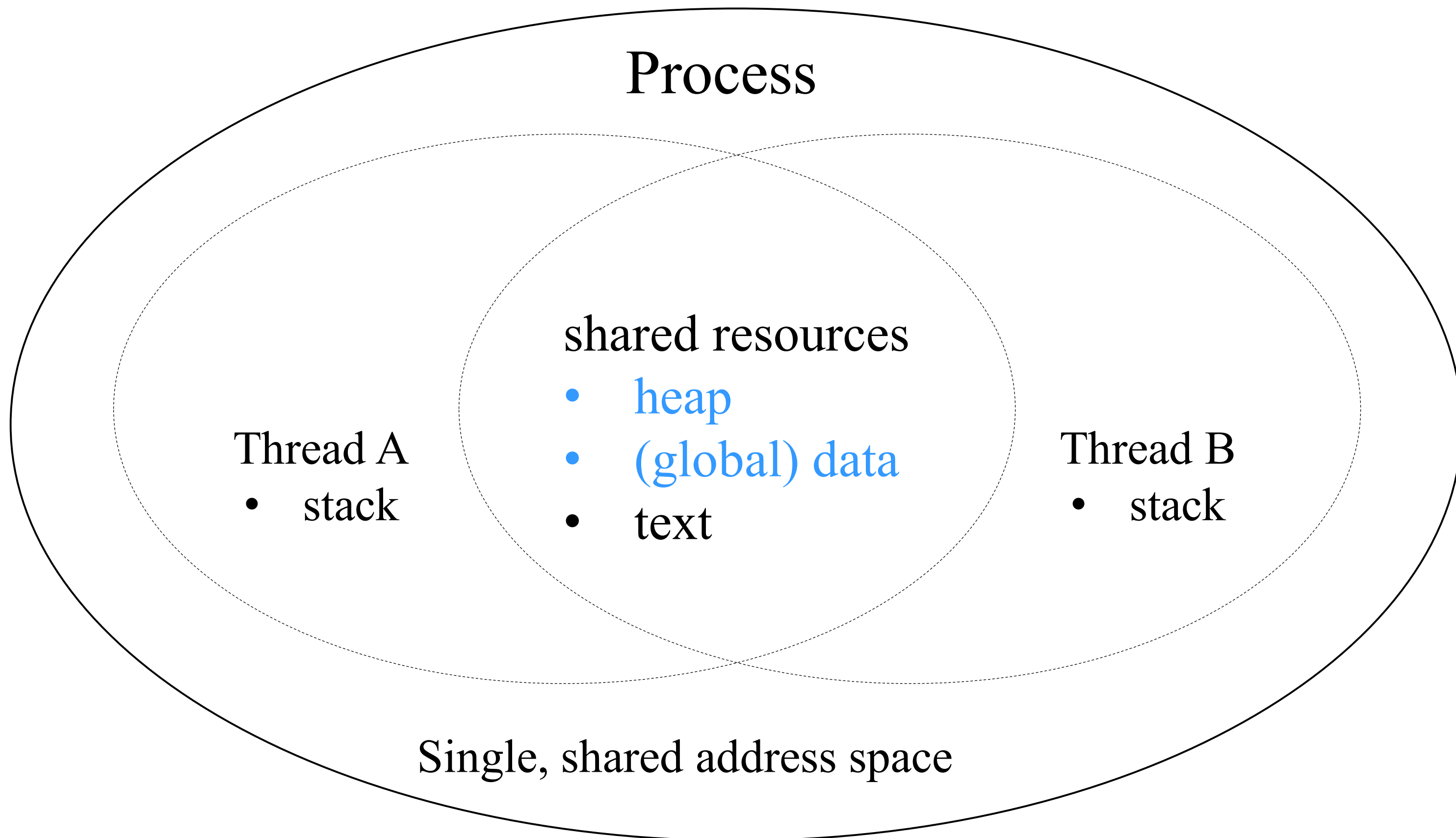
Some Sources of Design Causal Factors

Establish the pros and cons of the design of each software component to which the SSSF is allocated and determine whether they could be Causal Factors or Mitigations for a SSSF functional failure due to an erroneous Safety-Critical Decision by the software. (It's all about the safety-critical data integrity.)

Design weaknesses with respect to data integrity, e.g.:

- Shared data "race conditions"
- Loss of data in software "failovers"
- Failure to refresh temporal data
- Unhandled exceptions

Multi-(two)threaded Design



Variables or objects in the *heap* or *data* can be shared by the threads. This can lead to race conditions or thread deadlock. (*Text* can also be shared, but (usually) does not change in value.)

Pros and Cons of Multi-threaded Design

Pros for multi-threaded design:

- Allows software to be more responsive to an unpredictable external environment (new inputs from an operator, another computer, or a sensor)
- Each thread can be 'appropriately prioritized'

Cons for single threaded design:

- Improperly synchronized threads can corrupt shared data
- Improperly synchronized threads can deadlock (block each other forever)
- Improperly prioritized threads can cause starvation or unpredictable delays
- Much more difficult to analyze or test than single-threaded designs

On the Difficulties with Multi-threading

'Concurrency in software is difficult. However, much of this difficulty is a consequence of the abstractions for concurrency that we have chosen to use. The dominant one in use today for general-purpose computing is threads. But non-trivial multi-threaded programs are incomprehensible to humans.'

[*The Problem with Threads,* Technical Report No. UCB/EECS-2006-1, Edward A. Lee, Professor, Chair of EE, Associate Chair of EECS, University of California at Berkley, January 10, 2006]

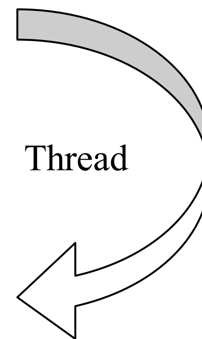
Single Threaded Design

A (usually infinite) loop, for an embedded program to "do its thing forever."

- Checks for input(s) [e.g., messages, sensor inputs]
- Performs any necessary processing of the input(s)
- Produces output(s) [e.g., messages, actuator control signals]

Example:

```
int main(void)
{   // initialization code here – done once
    for ( ; ; )  // or while (true) or while (1)
    { // read or detect stuff
      // do some calculation
      // write or command stuff
    }
}
```



Pros and Cons of Single Threaded Design

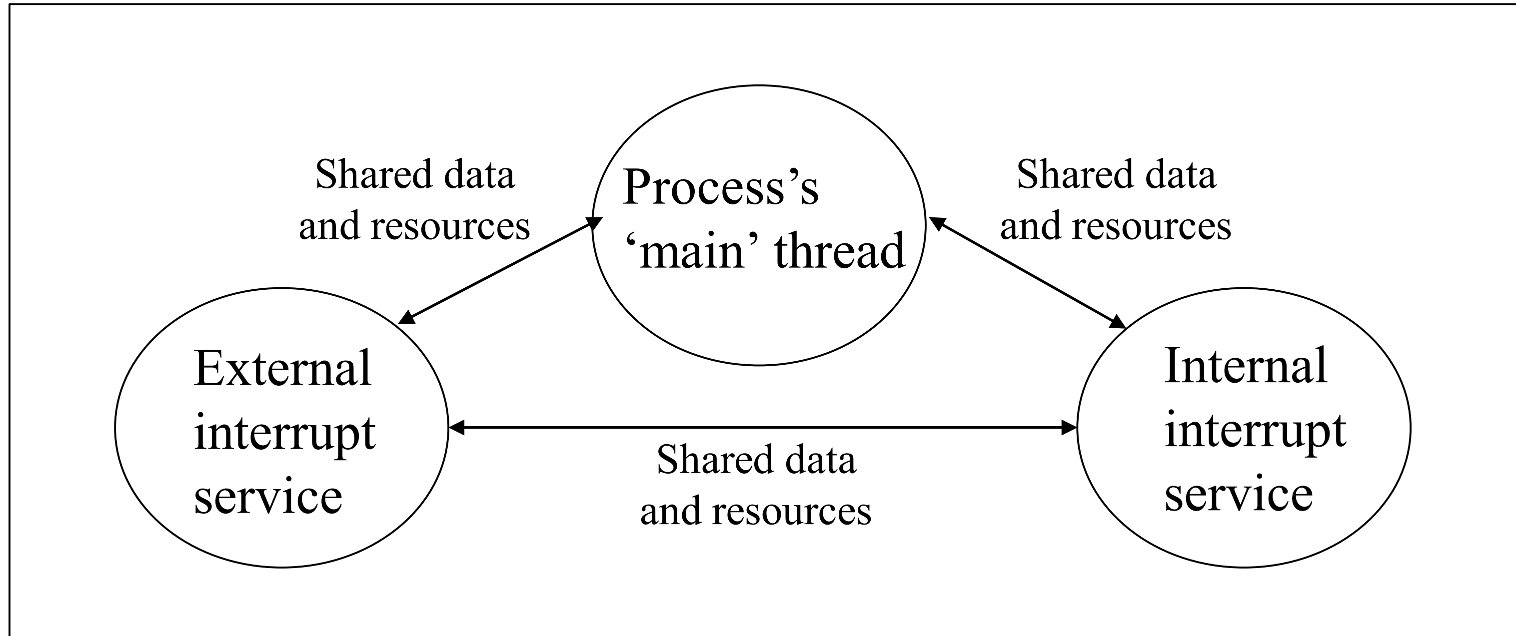
Pros for single threaded design:

- Easier to perform analysis (e.g., design, code, worst case timing)
- Easier to implement the first time

Cons for single threaded design:

- Delay in responding to external inputs
- Can become a bottleneck in the larger system
- Hard to prioritize multiple competing “tasks”
- Must implement the details for handling all I/O
- Becomes hard to maintain as more functionality is added

Single Threaded Design With Interrupt Service



- With few exceptions, Interrupt Service Routines (ISRs) should be short and sweet. For input, read the data into a buffer or queue, set a flag for 'main' to see, then get out of the way (let 'main' process the data).
- Non-atomic access by 'main' to data shared with an ISR must be protected from potential corruption (e.g., locking out the interrupt that drives the ISR while 'main' is reading from or writing to the shared data).

Pros and Cons of Design With Interrupt Service

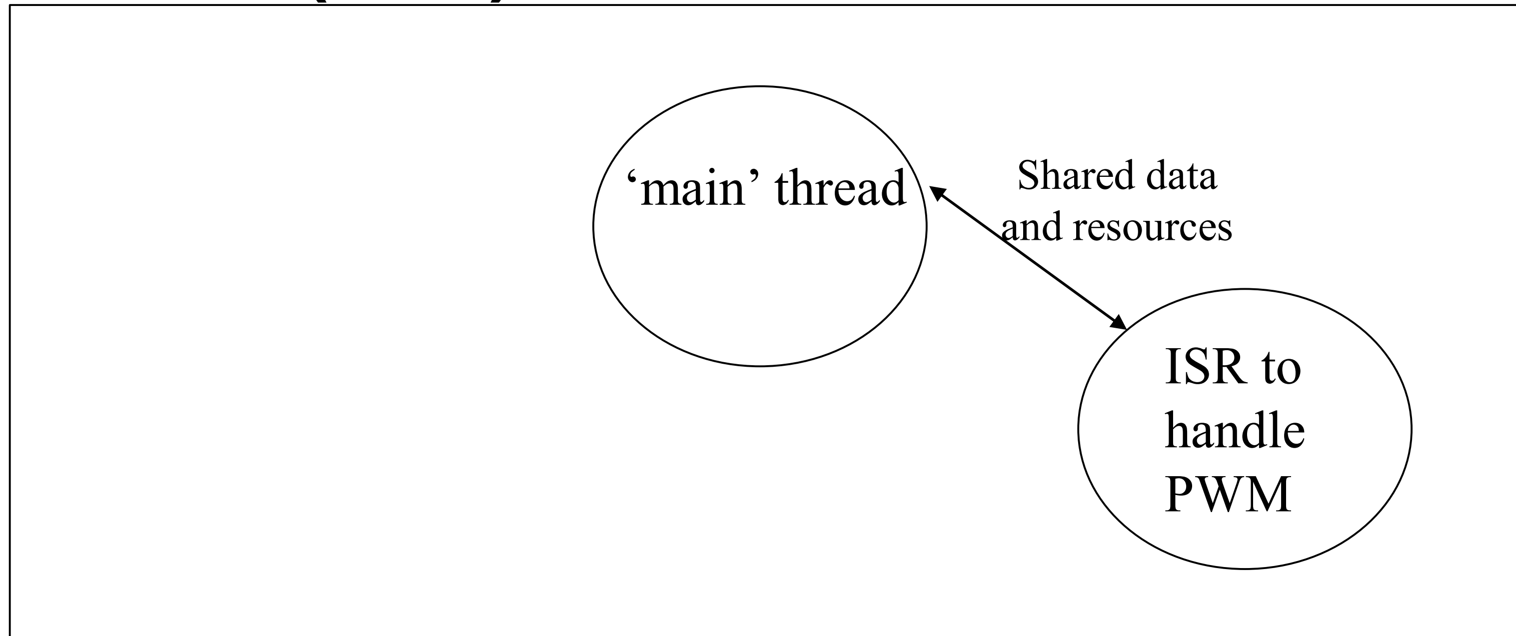
Pros for single threaded design with interrupt service:

- Somewhat more responsive to external inputs
- Relatively easy to perform analysis (e.g., design, code, worst case timing)
- Still easy to implement the first time

Cons for single threaded design with interrupt service:

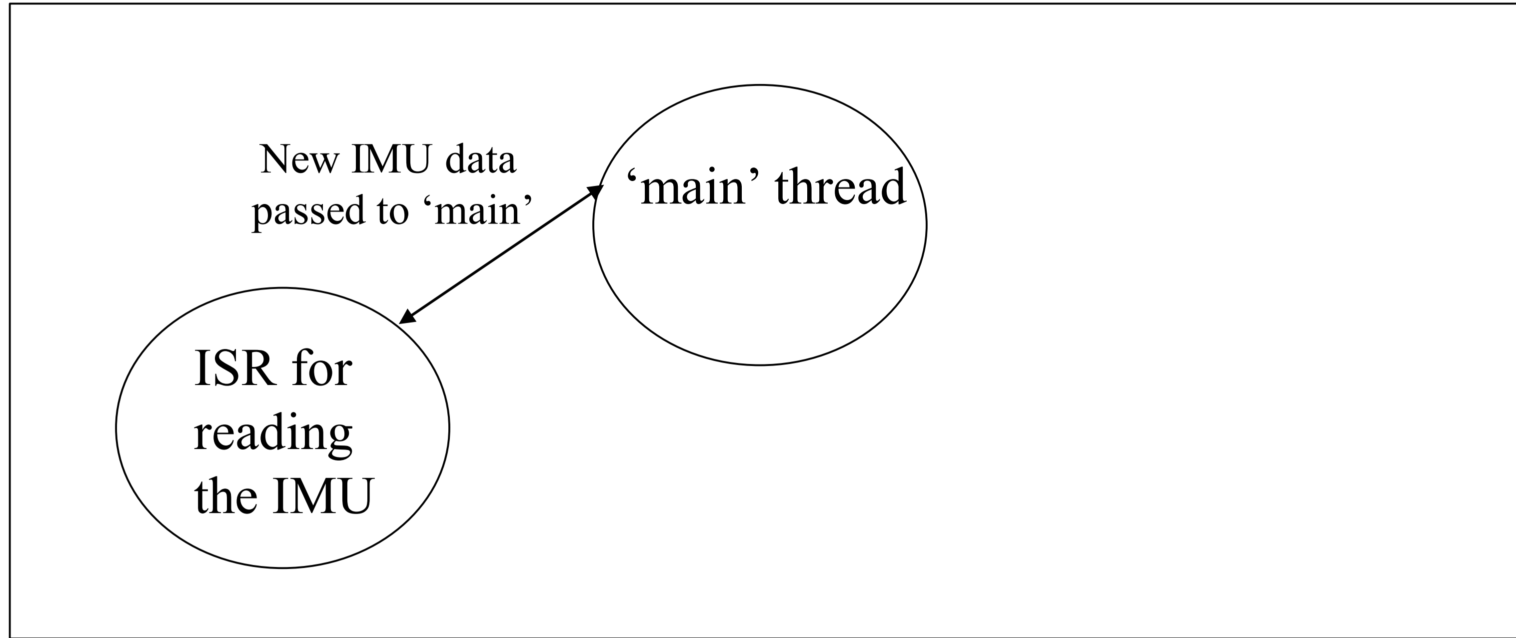
- Delay in responding to external inputs
- Main loop can still become a bottleneck (input queue overflow, delay in responding to external system)
- Still hard to prioritize multiple competing “tasks”
- Potential for corrupting data shared between ISRs and ‘main’
- Still hard to maintain as more functionality is added

Details of Pulse Width Modulation (Mis-)handled in the ISR



- A Programmable Power Supply was implemented so that almost all processing of sensors and control commands for pulse-width modulation (PWM) of the power output to power up missiles in a launcher for preparation to launch was performed inside the ISR.
- When a new missile was introduced, the interrupt occurred every 10 u-sec's and the ISR to 11 u-sec's to execute the additional processing for the power requirements for the new missile's launch preparation.

An ISR / ’main’ example of non-atomic data sharing



- Non-atomic access by ‘main’ to data shared with an ISR must be protected from potential corruption (e.g., locking out the interrupt that drives the ISR while ‘main’ is reading from or writing to the shared data).
- Inertial Measurements include several values - linear accelerations (x, y, and z) and rotational measurements (about each axis). Is the IMU ISR locked out while ‘main’ is reading the shared IMU data?

Code Analysis

Code Analysis vs. Design Analysis

The difficulty of using the term "design" in relation to software is that in some sense, the source code of a program is the design for the program that it produces.

[*Wikipedia* article on “Software Design,” February 7, 2015]

Focus for LoR 1 Code Analysis

SwCI 1 code is typically responsible for releasing potentially catastrophic energy or for detecting a potentially catastrophic hazardous condition. Either way that usually involves one or more Safety-Critical Decision Points (SCDPs) in the software. These SCDPs use one or more software data items to make the decision.

- Focus code analysis on identification of internal data items used by software to make critical decisions to perform a safety-critical action or not.
  - Scope may expand as analysis progresses.
- Investigate how a data item's value is set and referenced by the software.
- Static or dynamic code analysis tools should be used for a detailed analysis and to document important technical aspects.

Program Slicing

In computer programming, program slicing is the computation of the set of programs statements, the program slice, that may affect the values at some point of interest, referred to as a slicing criterion. Program slicing can be used in debugging to locate source of errors more easily. Other applications of slicing include software maintenance, optimization, program analysis, and information flow control.

[*Wikipedia* article on “Program Slicing,” March 17, 2015]

Some Code Analysis Tools

Tools to help an analyst explore the code:

- Eclipse (Java, C/C++) (Open Source Software)
- NetBeans (Java, C++)
- Understand for C++/Java (SCI Tools)

Tools to do automated static code analysis:

- CodeSonar (GrammaTech)
- Klocwork (Rogue Wave)
- Code Advisor (Coverity)
- PC-lint (Gimpel Software)

Code Analysis

1. For each SwCI 1 SSSF, identify and locate the SCDPs associated the SSSF.
2. Using appropriate automated or semi-automated code analysis tools, perform a “backward flow” analysis of the code from safety-critical decision points in the software.
3. Based on the results of the Requirements, Architecture, and Design Analyses, perform other appropriate code analyses, especially analysis of the implementation of safety critical mitigations for the SSSF.

Code Analysis

-- Step 1 --

1. For each SwCI 1 SSSF, identify and locate the SCDPs associated the SSSF.

- Locate the code that performs energy release. e.g., weapon firing, detonation, booster ignition. (potential Causal Factor) or that detects and responds to a hazardous condition.

An Example L-DETS SCDP

Source code for safety-critical function SetFirePulse and associated functions in file SafetyCritical_RX.c

```
//***********************************************************************
// @Function      void SetFirePulse(void)
//-----------------------------------------------------------------------
// @Description   This function applies a 30 millisecond firing Pulse to detonate the unit.
//***********************************************************************
void SetFirePulse(void)
{
    if(G_SCV.SC_DisableSafetyCriticalProcessing == SC_PROCESSING_ENABLED)
    {
        if(IsArmPinRemoved() == ARM_PIN_HAS_BEEN_REMOVED)
        {                                   // When pin is pulled we get a high
                FIRE_PULSE_PORT = 1;
                G_SCV_PortFImage |= FIRE_PULSE_BIT;
                G_SCV.SC_DetonatorHasFired = DETONATOR_HAS_FIRED;
                DelayMilliSecondsNoInterrupt(30);
                SetToSafeState();
        }
    }
}
```

Is detonation currently enabled?

Booz | Allen | Hamilton

Code Analysis

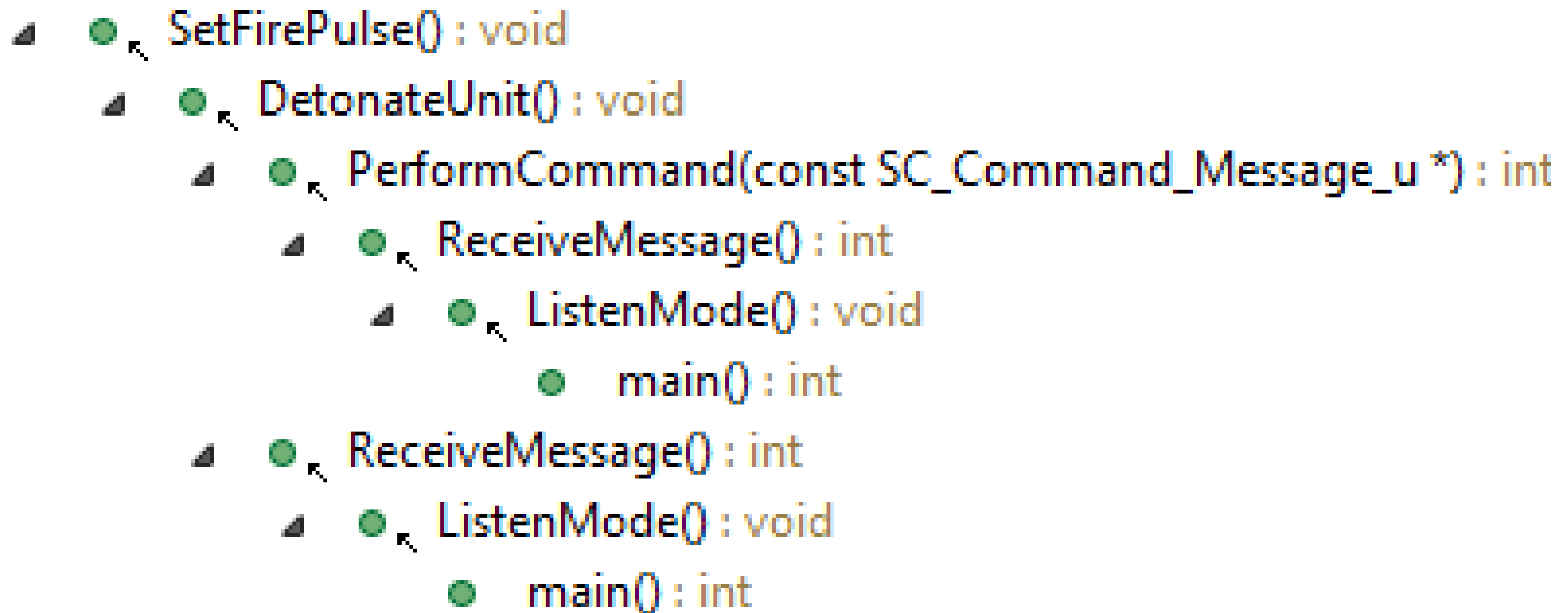
-- Step 2 --

2. Using appropriate automated or semi-automated code analysis tools, perform a “backward flow” analysis of the code from safety-critical decision points in the software.

- The analysis should focus on identifying potential causes of stale or corrupt data being present at the safety-critical decision point.

An Example L-DETS SCDP (cont'd)

Control Flow Analysis: How is SetFirePulse called in the L-DETS Detonator software?



SetFirePulse is only called from the function DetonateUnit, which is called on two paths within the "main" thread: one if the Fire Command is received directly from the Controller and the second if it has been forwarded from another Detonator.

An Example L-DETS SCDP (cont'd)

Data Flow Analysis: Where/how is SC_DisableSafetyCriticalProcessing updateded?

References to '(anonymous)::SC_DisableSafetyCriticalProcessing' (11 matches)

- LDETS
  - src
    - SafetyCritical_RX.c (11 matches)
      - ChargeCapacitor, line 82: if(G_SCV.SC_DisableSafetyCriticalProcessing == SC_PROCESSING_ENABLED)
      - EnableFiring, line 109: if(G_SCV.SC_DisableSafetyCriticalProcessing == SC_PROCESSING_ENABLED)
      - SetFirePulse, line 175: if(G_SCV.SC_DisableSafetyCriticalProcessing == SC_PROCESSING_ENABLED)
      - EnableSafetyCriticalProcessing, line 336: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_ENABLED;
      - DisableSafetyCriticalProcessing, line 354: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_DISABLED;
      - ClearSafetyCriticalVariables, line 377: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_ENABLED;
      - ClearSafetyCriticalVariables, line 381: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_DISABLED;
      - ClearSafetyCriticalVariables, line 401: if(G_SCV.SC_DisableSafetyCriticalProcessing != SC_PROCESSING_ENABLED)
      - ClearSafetyCriticalVariables, line 416: if(G_SCV.SC_DisableSafetyCriticalProcessing != SC_PROCESSING_DISABLED)
      - PowerUpClearSafetyCriticalVariables, line 450: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_DISABLED;
      - PowerUpClearSafetyCriticalVariables, line 468: if(G_SCV.SC_DisableSafetyCriticalProcessing != SC_PROCESSING_DISABLED)

SC_DisableSafetyCriticalProcessing is only enabled and disabled at five locations in the software. Understanding the purpose and use of each location is needed to assess for potential weaknesses or problems..

Blue highlighting indicates SC_DisableSafetyCriticalProcessing is referenced but not changed.

An Example L-DETS SCDP (cont'd)

Data Flow Analysis: Where/how is SC_DisableSafetyCriticalProcessing updateded?

References to '(anonymous)::SC_DisableSafetyCriticalProcessing' (11 matches)

- LDETS
  - src
    - SafetyCritical_RX.c (11 matches)
      - ChargeCapacitor, line 82: if(G_SCV.SC_DisableSafetyCriticalProcessing == SC_PROCESSING_ENABLED)
      - EnableFiring, line 109: if(G_SCV.SC_DisableSafetyCriticalProcessing == SC_PROCESSING_ENABLED)
      - SetFirePulse, line 175: if(G_SCV.SC_DisableSafetyCriticalProcessing == SC_PROCESSING_ENABLED)
      - EnableSafetyCriticalProcessing, line 336: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_ENABLED;
      - DisableSafetyCriticalProcessing, line 354: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_DISABLED;
      - ClearSafetyCriticalVariables, line 377: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_ENABLED;
      - ClearSafetyCriticalVariables, line 381: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_DISABLED;
      - ClearSafetyCriticalVariables, line 401: if(G_SCV.SC_DisableSafetyCriticalProcessing != SC_PROCESSING_ENABLED)
      - ClearSafetyCriticalVariables, line 416: if(G_SCV.SC_DisableSafetyCriticalProcessing != SC_PROCESSING_DISABLED)
      - PowerUpClearSafetyCriticalVariables, line 450: G_SCV.SC_DisableSafetyCriticalProcessing = SC_PROCESSING_DISABLED;
      - PowerUpClearSafetyCriticalVariables, line 468: if(G_SCV.SC_DisableSafetyCriticalProcessing != SC_PROCESSING_DISABLED)

SC_DisableSafetyCriticalProcessing is only enabled at two locations in the software.

Blue highlighting indicates SC_DisableSafetyCriticalProcessing is referenced but not changed.

Code Analysis
-- Step 3 --

3. Based on the results Requirements Analysis, Architecture Analysis, or Design Analysis, perform other appropriate code analyses that might have potential safety-critical impacts, such as:

- **Timing analysis** – for safety-critical hard real time requirements, using appropriate static or dynamic code analysis tools to analyze the worst case execution time (WCET).
- **Interrupt analysis** – analysis of the coordination of interrupt handling with interruptible and non-interruptible safety-critical processing.
- **Algorithm correctness** – analysis of the correctness of the implementation of any safety-critical algorithm(s)..
- **Data structure/usage analysis** – analysis of the structure and use of safety-critical data objects associated with the SSSF.
- **OS function analysis** - analysis of correct use of OS functions used to implement LOR 1 functionality for the SSSF.

Wrap Up

Some Key Points

- Purpose of LoR is to focus and manage

Some Key Points

- Purpose of LoR is to focus and manage
- Software FHA should:
  - Be performed as early as reasonable
  - "Rack and stack" SSSFs by SwCI/LoR
  - Identify potential redundancies to reduce SwCI 1 SSSFs

Some Key Points

- Purpose of LoR is to focus and manage
- Software FHA should:
  - Be performed as early as reasonable
  - "Rack and stack" SSSFs by SwCI/LoR
  - Identify potential redundancies to reduce SwCI 1 SSSFs
- Requirements analysis should focus on:
  - Incompleteness
  - Ambiguities
  - Conflicts

Some Key Points

- Architecture analysis should focus on weaknesses in the command and control of distributed Safety-Significant Software Functions

Some Key Points

- Architecture analysis should focus on weaknesses in the command and control of distributed Safety-Significant Software Functions
- Design and code analysis should focus on Safety-Critical Decision Points (can the internal data items used by the software be corrupted or stale)

Some Key Points

- Architecture analysis should focus on weaknesses in the command and control of distributed Safety-Significant Software Functions

- Design and code analysis should focus on Safety-Critical Decision Points (can the internal data items used by the software be corrupted or stale)

- In-Depth Safety-Specific Testing should be derived from the analysis results

Some Key Points

- Architecture analysis should focus on weaknesses in the command and control of distributed Safety-Significant Software Functions

- Design and code analysis should focus on Safety-Critical Decision Points (can the internal data items used by the software be corrupted or stale)

- In-Depth Safety-Specific Testing should be derived from the analysis results

- All analyses and testing should be focused on Causal Factors and Mitigations