17803
Software Safety Functionality Hazard Assessment

Focusing Level of Rigor (LOR) on the *Safety-Critical Software Decision Points*

Stuart A. Whitford
Booz Allen Hamilton

18th Annual NDIA Systems Engineering Conference
Springfield, VA
28 October 2015

Booz | Allen | Hamilton

Agenda

- The Challenge
- MIL-STD-882E Guidance
- Focusing the Effort
- Safety-Significant Software Function (SSSF) Hazard Assessment
- Examples
- Conclusion

NOTE: Blue highlighting in this presentation is for ***emphasis.***

The Challenge: Achieving Required "Level Rigor"

MIL-STD-882E paragraph 3.2.18 defines Level of Rigor (LOR) as:

- "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."

From MIL-STD-882E, paragraph 4.4.3, Table VI:

- If LOR 1 (the highest) tasks are unspecified or incomplete, the Standard *requires* a Program Manger to "prepare a formal risk assessment for acceptance of a HIGH risk."
- If LOR 2 tasks are unspecified or incomplete, the Standard *requires* a Program Manger to "prepare a formal risk assessment for acceptance of a SERIOUS risk."

"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." [MIL-STD-882E paragraph 4.1]

MIL-STD-882E Mishap Severity

3.2.36 Severity. The magnitude of potential consequences of a mishap to include: death, injury, occupational illness, damage to or loss of equipment or property, damage to the environment, or monetary loss.

SEVERITY CATEGORIES (from MIL-STD-882E, Table I)

1) Catastrophic:
   - death . . . loss equal to or exceeding $10M
2) Critical:
   - permanent partial disability . . . exceeding $1M but less than $10M
3) Marginal:
   - one or more lost work day(s) . . . exceeding $100k but less than $1M
4) Negligible:
   - injury . . . not resulting in a lost work day . . . less than $100k

MIL-STD-882E Software Control Categories

3.2.38 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.

SOFTWARE CONTROL CATEGORIES (SCC) (from -882E, Table IV)

1) Autonomous (AT)
2) Semi-Autonomous (SAT)
3) Redundant Fault Tolerant (RFT)
4) Influential (INF)
5) No Safety Impact (NSI)

MIL-STD-882E Software Safety Criticality Matrix (Table V)

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

SwCI = Software Criticality Index

MIL-STD-882E Required Levels of Rigor (Table V – continued)

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.

MIL-STD-882E Task 208: Functional Hazard Analysis

208.1 Purpose. … safety-critical functions (SCFs), … safety-related functions (SRFs) ... will be allocated or mapped to the system design architecture in terms of hardware, software, and human interfaces to the system. ... allocate and partition SCFs and SRFs in the software design architecture; and identify requirements and constraints to the design team. … Assign a SwCI for each SSSF [Safety-Significant Software Function] mapped to the software design architecture.

MIL-STD-882E Guidance on Performing Software Safety

B.2.2.3 Software Safety Criticality Matrix (SSCM) tailoring ... SwCI 1 from the SSCM implies that the assessed software function or requirement is highly critical …and requires more design, analysis, and test rigor than software that is less critical...

Process tasks. Process tasks to consider include … safety review, design walkthrough, code walkthrough, independent design review, independent code review, independent safety review, traceability of SSFs, SSFs code review, SSFs, Safety-Critical Function (SCF) code review, SCF design review, test case review, test procedure review, safety test result review …

Test tasks. Test task considerations include SSF testing, functional thread testing, … failure modes and effects testing, out-of-bounds testing, safety-significant interface testing, … independent testing of prioritized SSFs, …

SSF = Safety-Significant Function

Achieving *Focused* Software Safety *Level of Rigor*

Perform a system-level FHA
- Allocate SSSFs to the software architecture
- Assign SwCI to each SSSF

For each SwCI 1, 2, or 3 SSSF identified in the FHA, perform a SSSF Hazard Assessment
- Perform required analyses, focusing on safety-critical software decision points
- Perform In-Depth Safety-Specific Testing
- Document a worksheet

For each SwCI 4 SSSF
- Perform required Safety-Specific Testing along with non-safety testing
- Ensure that SwCI 4 tests are tagged and reported appropriately

Safety-Critical Software Decision Points – some examples:

For Navy weapon systems, typical safety-critical software decision points are:

- Is it safe to arm/fire/launch the weapon?
- Is the track a friendly or non-hostile track?
- Is there a dangerous system condition that needs immediate response?

Each of the required analyses (requirements, architecture, design, code) and the in-depth safety-specific testing should be focused on these safety-critical decision points within each SSSF.

SSSF Hazard Causal Factors

For each safety-significant software decision point within the SSSF, assess the requirements/architecture/design/code for potential problems that could impact the decision. Software makes decisions based upon data. Problems with data include:

- Data latency (late or early arrival of data used in the decision)
- Data corruption or loss (e.g., from transmission or mishandling)
- Data coherency (e.g., mismatched elements in a “set” of data used in the decision)
- Invalid or erroneous data value (e.g., message from an external system or function)

Each required analysis (requirements/architecture/design/code) should look for weaknesses that could impact the integrity of the data used in the safety-critical software decision.

SSSF Mitigations

For potential causal factors identified within the SSSF, assess the requirements, architecture, design, code for potential mitigations that prevent or detect and respond to ***each*** causal factor. These are often some form of redundancy, such as:

- (in communication) checksums, CRCs, required response, repeated transmission
- (in data) additional messages, additional message fields, input validation criteria
- (in processing) checkpoint/restart, recovery blocks, N-version programming

Each required analysis (requirements/architecture/design/code) should look for strengths that help ensure the integrity of the data used in the safety-critical software decision.

In-Depth Safety-Specific Testing

For the causal factors and mitigations found, identify appropriate in-depth safety-specific testing to 1) validate the SSSF mitigations and 2) provide some assurance of absence of occurrence of the potential SSSF causal factors under credible levels of system stress. The following kinds of testing should be considered:

- Stress
- Endurance
- Load
- Boundary limit
- Error handling
- Failover
- Out of sequence
- Out of range
- Fault injection

A Tale of Two "Threads"

At the System Engineering Level:

4.3.7.2.3 Safety-Critical Path Analysis, Thread Analysis, and UML Sequence Diagrams ... a path would be defined as events that, when performed in a series (one after the other), cause the software to perform a particular function. . . UML sequence diagram . . . Functional Flow Diagrams (FFDs) and DFDs . . . [Joint Software System Safety Engineering Handbook (JSSSEH), 2010]

At the Software Design Level

Thread (computing) . . . In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. [August 26, 2015 from the Wiki: http://en.wikipedia.org/wiki/]

What is "Architecture"?

There are many definitions out there. The following is what I tend to use:

'Architecture is concerned with the selection of *architectural elements,* their *interaction,* and the *constraints* on those elements and their interactions.'

[D. E. Perry, A. L. Wolf (1992). Foundation for the Study of Software Architecture." ACM SIGSOFT Software Engineering Notes 17 (4), pp. 40—52.]

'Architecture focuses on *the externally visible properties of software "components*." '

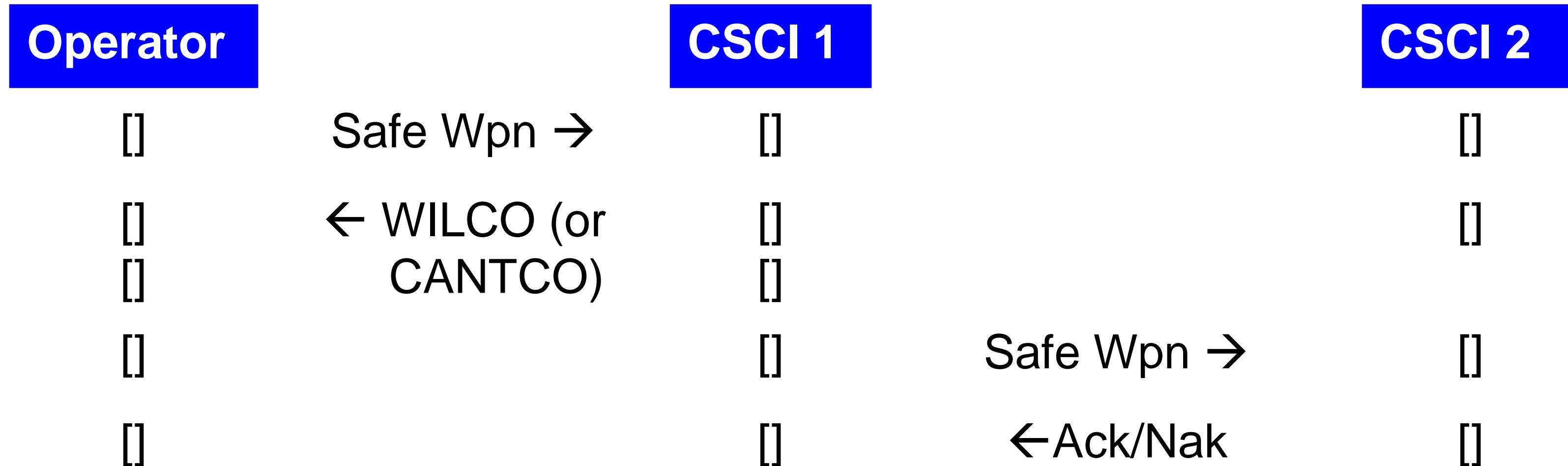
[L. Bass, P. Clements, R. Kazman (1998). Software Architecture in Practice. Reading, MA: Addison Wesley Longman, Inc.]

Architectural Analysis (an example)

Safety critical command and control allocated to a distributed architecture:

- Does the architecture provide adequate checks for the integrity of communications (e.g., checksums, CRCs)?
- Does the architecture provide adequate, timely positive and negative feedback (e.g., “Can’t Comply”, “Will Comply”, “Have Complied”)?
- Do the interfaces between software components provide effective and timely communication of safety-significant fault detection and handling?

System Thread (Path) Analysis for a 'Safe Weapon' SSSF



CSCI = Computer Software Configuration Item
WILCO = "Will Comply"
CANTCO = "Can't Comply"

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

Multi-threaded Software Design

Strengths:

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

Weaknesses:

- 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
- Poor language or tool support for the programmer

Multi-threaded design of a 'State Manager' SSSF (example)

Thread A (lower priority):

If wpnState not eq SHUTDOWN

. . . getStateMutex
. . . superState = Homing
. . . wpnState = initiateFuzing
. . . releaseStateMutex

Thread B (higher priority):

[thread "unblocks"]
Critical fault detected
. . . getStateMutex
. . . superState = Operating
. . . wpnState = SHUTDOWN
. . . releaseStateMutex
[thread eventually "blocks"]

'State Manager' SSSF (example continued)

Thread A (lower priority):

Critical fault detected
. . . getStateMutex

. . . superState = Operating
. . . wpnState = SHUTDOWN
. . . releaseStateMutex
[this unblocks Thread B]

Thread B (higher priority):

[thread "unblocks"]
If wpnState not eq SHUTDOWN
. . .getStateMutex *(attempts)*
[this blocks thread]

. . .getStateMutex *(succeeds)*
. . . superState = Launch
. . . wpnState = launchInit
. . . releaseStateMutex

‘State Manager’ SSSF (solution)

Thread A (any priority):

getStateMutex
If wpnState not eq SHUTDOWN
. . superState = Homing
. . wpnState = initiateFuzing
releaseStateMutex

[move the check inside the mutex block]

Thread B (any priority):

Critical fault detected
. . . getStateMutex
. . . superState = Operating
. . . wpnState = SHUTDOWN
. . . releaseStateMutex

Multi-threaded design of a 'Weapon Inhibit' SSSF (example)

Thread A (lower priority):

Old 20s Weapon Inhibit
timer expires
clearWpnInhbt ()

. . . wpnInhibit = FALSE

[note that no 'shared data' mutex is used for wpnInhibit]

Thread B (higher priority):

[unblocks on receipt of new Weapon Inhibit command]
. . . if old timer active, cancel it
. . . wpnInhibit = TRUE
. . . Initiate a (new) 20s timer
[thread blocks on task completion]

‘Weapon Inhibit’ SSSF (solution)

Thread A (any priority):

20s Weapon Inhibit timer expires
clearWpnInhbt(storedTimestamp)
. . getInhibitMutex
. . if wpnInhibitTimestamp EQ
storedTimestamp
[from the OLD timer’s
“currentTime”]
. . . . wpnInhibit = FALSE
. . . . wpnInhibitTimestamp = 0
. . releaseInhibitMutex

Thread B (any priority):

New Weapon Inhibit command
. . getInhibitMutex
. . if old timer active, cancel it
. . wpnInhibitTimestamp =
currentTime()
. . wpnInhibit = TRUE
. . Initiate a new 20s timer and
“store” a copy of the
wpnInhibitTimestamp with it
. . releaseInhibitMutex

Multi-threaded Software Design

‘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]

Safety Critical Data 'Corruption'

A correctly implemented algorithm operating on corrupted safety-critical data can have unintended catastrophic results.

Some sources of corrupted data:

- Noise in digital message transmission
- Physical events/upsets during data storage
- Multi-threaded shared data
- Shared data between 'main' and Interrupt Service Routines
- Caching of data
- Loss of transient status data in failover or 'recovery'

Conclusion

Programs often stop the ***analysis*** of software safety with the identification of the SwCI, then focus the software safety effort on the tagging of software requirements (as 'Safety') and the testing of tagged requirements.

- Provides, at best, evidence of the accomplishment of LOR 3 to service safety review boards
- Results in a requirement to document HIGH or SERIOUS unknown software safety risk for LOR 1 and 2 software functionality, due to incomplete LOR, for acceptance by the appropriate service authority

The focused SSSF Hazard Assessment approach:

- Provides clear evidence of application of the appropriate LOR focused on *each SSSF*
- Identifies the CFs, mitigations, and In Depth Safety-Specific Testing performed for each SSSF, focused on the key safety-critical decision points

Questions?

Stuart Whitford
Senior Lead Scientist

Booz | Allen | Hamilton

Booz Allen Hamilton
1550 Crystal Dr, Suite 1100
Arlington, VA 22202
Tel (540) 903-7035
whitford_stuart@bah.com

Backup Slides

Requirements Analysis

Tag requirements associated with each SSSF as “safety” and assess for:

- Completeness
    - -- Do the requirements cover: Input validity/sequence? Early/late/non- arrival of input?
    - -- Are the requirements unambiguous?
    - -- Are they testable?
- Potential conflict with other requirements
- Bi-directional traceability

Architecture Analysis

Map each SSSF to the architecture and assess for:

- Partitioning/isolation of SSRs (allocation of SSRs to software components).
- Coordination of command and control of safety-critical system functionality among software components.
  - -- Does the architecture provide adequate checks for the integrity of communications (e.g., checksums, CRCs)?
  - -- Does the architecture provide adequate, timely positive and negative feedback (e.g., CANTPRO, CANTCO, WILCO, HAVCO - see MIL-STD-2045-47001D/DOD Interface Standard: Connectionless Data Transfer Application Layer Standard)?
  - -- Do the interfaces between software components provide effective and timely communication of safety-significant fault detection and handling?
- Message structure/usage
  - -- Are safety-significant data mapped to interface messages in a manner to facilitate safe, reliable, and timely communication between the software components?

Design Analysis

Map each SSSF to the software design and assess for safety impacts from:

- Potential control flow problems between design elements
- Potential latency issues
- Potential OS functional failures on the SSSF
- Problems with thread synchronization
- Problems with interrupt servicing

Code Analysis

Perform a “backward flow” analysis of the code from safety-critical decision points in the software.

Based on the results of the Requirements, Architecture, and Design Analyses, perform other appropriate code analyses:

- Timing analysis – For safety-critical hard real time requirements, use appropriate static or dynamic code analysis tools to analyze the implementation of the time-critical SSSF functionality to determine worst case execution time (WCET).
- Interrupt analysis – Perform a code analysis of the coordination of interrupt handling with interruptible and non-interruptible safety-critical processing associated with SSSF.
- Algorithm correctness – Perform a code analysis of the correctness of the implementation of any safety-critical algorithm(s) associated with the SSSF. This should cover both correctness and timeliness of the execution of the algorithm.
- Thread analysis – Perform a code analysis of thread synchronization and use of safety-critical data objects associated with the SSSF (looking for shared data race problems or thread deadlock).

In-depth Safety-Specific Testing

In-depth Safety-Specific Test cases should come from the SSSF analyses and be focused tests beyond normal requirements-based testing:

- Boundary limit testing:
    - -- Data range limits
    - -- Timing limits
- Robustness/stress testing
- Fault injection testing
- State transition testing
- Out of sequence testing
- Out-of-range value testing
- Error and exception handling testing

Tools to Support Software Safety Analysis

Use tools to help analyze the SSSF in the context of the Architecture, Design, or Code (leverage those in use by the software developers or obtain):

- Software architecture and design modeling and analysis tools, such as those supporting Architecture Analysis and Design Language (AADL), Unified Model Language (UML), or Systems Modeling Language (SysML)
- Static code analysis tools that support focused design and code analyses, such as thread race/deadlock detection or program slicing
- Source code cross reference tools that support searching, cross-referencing, and navigation (forward and backward) of source code trees

Some References

- Joint Software Systems Safety Engineering Workgroup. (2010). Joint Software System Safety Engineering Handbook (JSSSEH). Indian Head, MD: Naval Ordnance Safety and Security Activity.
- S. Beatty (2003). "Where Testing Fails." *Embedded Systems Programming*. August, 2003. pp. 36-41.
- P. Butcher (2014). *Seven Concurrency Models in Seven Weeks: When Threads Unravel.* The Pragmatic Programmers, LLC.
- B. P. Douglass (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston, MA: Addison-Wesley.
- P. Koopman (2010). *Better Embedded System Software*. Carnegie Mellon University. Drumnadrochit Press.
- D. Simon (1999). *An Embedded Software Primer*. Boston, MA: Addison-Wesley.