ITEA Journal 2009; 30: 409–416

Challenges in Software Safety for Army Test and Evaluation

Frank Fratrik
U.S. Army Aberdeen Test Center,
Aberdeen Proving Ground, Maryland

As the capabilities of software intensive systems grow so does the complexity of functions controlled via software. Similarly, software test and evaluation (T&E) efforts have become increasingly difficult to quantify and scope appropriately. T&E efforts for traditional programs have hinged on system level testing in realistic or simulated environments to verify and validate the systems. Application of these traditional methods to software intensive systems continues to hold value, but it no longer provides exhaustive data. A number of T&E deficiencies are surfacing in test programs for these software intensive systems as controllability and visibility related to software functionality decreases. The result is testing that can fail to uncover critical problems, potentially with catastrophic results. This article describes successes and shortcomings with current test and analysis methodologies for software intensive systems. As a part of the Army Test and Evaluation Command (ATEC), the author looks from an unbiased viewpoint at relevant current practices and the outlook for future T&E in regards to software safety. Recent examples of Army software test and analysis efforts, current Army T&E guidance for software safety, and a path forward for increasing confidence in software safety will be discussed.

Key words: Residual risk; risk; software intensive systems; software safety; T&E current practices; test and analysis methodologies.

Developmental test (DT) programs for many military systems over the past several decades have successfully verified safety to a high degree of confidence. However, with the rise of complexity and criticality allotted to software control, software test methodologies must be expanded to achieve the same high confidence required by those responsible for system safety.

The goal of this discussion is to increase awareness of software safety and identify practices and methods, based on existing guidance, which will reduce hazards associated with military systems that use software to achieve system safety objectives. The target audience includes developers, program managers (PMs), acquirers, and others involved in test and evaluation (T&E) as it relates to system safety.

Because of the data gaps when testing software in a completed system, analysis of both safety engineering and software engineering efforts must be accepted as data points in building a case for software safety. Only after earlier development efforts have been assessed can the right system level DT scope and environments be identified.

Current Department of Defense (DoD) software safety guidance includes a series of analyses in combination with testing to provide the best confidence for systems with safety critical software. Many current programs still lack the suggested development processes and rigor. In recent years this has led to systems with safety critical software displaying incomplete hazard analysis, lack of requirements traceability, poor design, and insufficient testing. Deficiencies in these key analysis areas resulted in only a partial data set being available for safety decision makers.

To fill these data gaps several methods are considered. Test results ranging from unit level up to system level will be assessed for their value in software safety. Relevant analyses, including hazard analysis, requirements analysis, architectural and detailed design analysis, and software code analysis, used to compliment test data will also be discussed.

DoD guidance for software safety

Safety Critical *(from MIL-STD 882C). A term applied to a condition, event, operation, process,*

Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **SEP 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE **Challenges in Software Safety for Army Test and Evaluation**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **U.S. Army Aberdeen Test Center,Aberdeen Proving Ground,MD,21005**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**8**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

or item of whose proper recognition, control, performance or tolerance is essential to safe system operation or use, e.g., safety critical function, safety critical path, safety critical component.

As previously stated, the goal in this discussion is not to introduce novel concepts relating to safety critical software. Instead, shortcomings with adherence to existing guidance are highlighted and a path forward to concurrence with those guidelines is identified. Multiple current resources exist for obtaining software safety guidance in a DoD context; however, little is regulatory. In 1999, the Joint Software System Safety Committee (JSSSC) Software System Safety Handbook (JSSSC, 1999) was released, which provides significant content in regards to identification, development, and verification of safety critical software. This is the most comprehensive single source for DoD software safety guidance.

Multiple other sources are also available, including an International Test Operating Procedure (ITOP) for Safety Critical Software Analysis and Testing (U.S. Army, 1999) the U.S. Army Communications and Electronics Command (CECOM) Software System Safety Guide (U.S. Army, 1992), and the U.S. Naval Sea Systems Command (NAVSEA) Weapon System Safety Guidelines Handbook (NAVSEA, 2006). A draft Software System Safety Policy is also in circulation for the U.S. Army Aviation and Missiles Command (AMCOM) (AMCOM, 2008).

These are by no means the only sources of guidance. It is critical to be aware that guidance for software safety exists and to find the most relevant information that applies to your system and its goals.

Software testing versus hardware testing

For those who have been in the T&E domain for a number of years, it may seem unclear why software information cannot be obtained as part of a traditional system level testing for verification of safety. Software testing limitations are characterized by the JSSSH E.13.1, General Testing Guidelines as follows:

"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." (JSSSC, 1999)

It is because of these limitations that a gap exists when testing systems with safety critical computer software components.

Safety Critical Computer Software Components *(from MIL-STD 882C). Those computer software components and units whose errors can result in a potential hazard, or loss of predictability or control of a system.*

Generally, traditional system level DT is able to characterize the system down to the smallest hardware components. Environments can be simulated for whole systems or parts of systems. Failures can be isolated and characterized by looking at failure points. As an example, system level vibration testing will submit all hardware elements to the test environment. Elements such as vehicle chassis, weapon mounts, engine components, and bolts are all stressed in this test. Failure points can be visually inspected.

On the contrary, software is not intended to be characterized by exposure to traditional system level testing. Complex software will commonly use only a subset of the total software code to achieve common mission functionality. This leaves rarely used (but possibly safety critical) code untested. Specific software functions can be difficult to stimulate or difficult to observe for correct output. Consider the example of a built-in-test (BIT) function achieved via software. The BIT may require internal subsystem failures to occur to exercise software code responsible for detection and reaction. A system level test might require causing damage to some internal subsystems or it may not be possible even via destructive testing due to constraints with observing outputs in a black box test.

Safety critical computer software component identification

If additional methods are to be applied to gain confidence in software safety, it is wise to focus the increased scope on safety critical computer software components. In order to do this, we must identify what software in a given system has a safety impact. Software safety must be considered a subset of system safety and flow system level hazards down to subsystems that are relevant. System level hazards should be allocated to hardware, software, or some combination of both.

Software can play a role in safety in many systems where it is a cause of a hazard. Examples could be causing hardware to perform unsafe actions or guiding an operator to make unsafe decisions. Software can also be a control of a hazard. It may be used to prevent hazards or limit severity after a mishap occurs. In either case, the software role in system safety must be defined based on top level hazards related to the entire system.

Software by itself does not impact safety; however, when coupled with critical hardware the software can become safety critical. Software that performs functions like these is likely to be safety critical:

- arm, enable, release, launch, fire, or detonate a weapon system;
- control movement of gun mounts, launchers, and other equipment;
- control movement of munitions or hazardous materials;
- monitor the state of the system for purposes of ensuring its safety;
- sense hazards and/or display information concerning the protection of the system;
- control or regulate energy sources in the system;
- exercise autonomous control over safety critical hardware;
- generate outputs that display the status of safety critical hardware systems;
- compute safety critical data.

These are merely a sample of potential examples derived from the JSSSH, section E.1.1.3 (JSSSC, 1999). For every system a detailed analysis of the software role in system safety must be conducted.

Increasing confidence in safety critical software

Moving forward requires the following two assumptions:

1. Correct safety critical software functionality is required for safe system operation.
2. Software is typically not sufficiently characterized by system-level testing alone.

To bridge the implied gap resulting from these assumptions, the recommendation is a disciplined approach to system, safety, and software engineering that manages software's role in system safety.

This approach results in evidence that safety is a prominent part of specifying, designing, building, and testing safety critical software. Analysis of this evidence can in itself result in increased confidence in the safety of associated software. Also, this analysis can be used to better scope and execute system level safety testing. JSSSH guidance covers a broad spectrum of analyses and artifacts; however, the minimum set of information required for the software safety analyses in this discussion includes these elements:

1. Hazard Tracking System (HTS) data,
2. software requirements,
3. software design,
4. software problem reports (SPRs),
5. software test plans and results,
6. safety assessment report (SAR).

Software safety analyses

The analyses described in this section are not exhaustive but are considered in this discussion to be the *minimum* required to support safety decision makers. In all cases, the number and rigor of software safety analysis methods should be proportional to the software hazard criticality, as defined in the JSSSC Handbook (JSSSC, 1999) and MIL-STD 882C (DoD, 1993). In some cases, the analyses will be minimal and aim to prove that software is not safety critical. For each area, there are typically multiple different methodologies that can be applied. They are ideally to be conducted by a safety agency independent of developers and acquirers, such as the Developmental Test Command (DTC) in the Army domain. The safety agent does not have the primary responsibility for developing these artifacts but is instead a consumer who uses and assesses them. Nominal responsible developing entities for Army systems are listed for each artifact; however, this may vary between organizations or programs.

Generally, these six elements flow chronologically in their development, but all will typically mature and develop iteratively. The value of any one of these elements is limited in and of itself. The largest benefits can be realized if all six are analyzed and compared in the context of each other.

HTS data analysis

Data from a HTS, sometimes called a Hazard Tracking Database, is typically a PM responsibility in an Army setting. Often generation of this artifact is delegated to a developer. This is considered a starting point for software safety analysis, as it is a primary source for identification of system level hazards and subsystems that could contribute to or prevent mishap occurrence. From a software safety perspective it is key that hazards allocated to software are clearly identified. Chapter 12 in Section II of the NAVSEA Weapon System Safety Guidelines Handbook is a helpful resource for planning, implementing, and updating an HTS (NAVSEA, 2006).

Software requirements analysis

Software requirements are normally a developer responsibility. Software requirements must be derived to specify the software role in both system functionality and system safety.

In addition to general requirements best practices for all requirements, safety requirements should be identified explicitly. Tracing up to system requirements and system hazards provides validity. Tracing down to software test cases, and possibly software design elements, is needed to allow for verification. All requirements need to be written in a way that they are testable.

Software requirements analysis can identify any gaps in traceability or other shortcomings that could impact the software artifacts to be derived from the requirements specification. More detailed expectations for derivation and tracing of safety critical software requirements is detailed in the JSSSH, section 4.3.4 Derive System Safety Critical Software Requirements and section 4.3.5.3 Traceability Analysis (JSSSC, 1999).

Software design analysis

Software design is typically a developer responsibility and normally flows out of software requirements. This can include both architectural design and detailed design. Software design best practices, such as maximizing cohesion and minimizing coupling, translate well to the safety domain. Isolation of software with safety impact in design can decrease the risk of a safety impact from other outside software modules.

Identification of critical software modules will alert software maintainers to potential impacts of changing software code in a safety critical area. Another benefit is that the impact of changes can be assessed using design information. This aids in identifying proper regression test scoping.

There are many design elements that can aid in coding with safety in mind, such as using interlocks, safety flags, watchdogs, or other techniques. Each of these can be considered during software design analysis. More detailed expectations for design of safety critical software can be found in the JSSSH, section 4.3.6 Detailed Software Design, Subsystem Hazard Analysis (JSSSC, 1999).

Developer SPR analysis

SPR generation is typically a developer responsibility and begins as early as practical after software implementation has begun. Formal SPR development ensures that all discovered problems are addressed and tracked to closure. SPRs that are not closed in a given software version can then be compiled and assessed for their impact.

It is important that SPRs be categorized for criticality. Those with safety impact should be made explicit and given highest priority for fix implementation. SPRs intended for closing should show traceability to test cases for verification of a successful fix. SPR tracking as a function of time can also be a useful metric for considering software maturity or completeness of test. SPR tracking is supported by the JSSSH, section E.13.1 General Testing Guidelines (JSSSC, 1999).

Developer software test planning and execution

SPRs are typically generated as a result of testing, which is the next focus area. To be clear, this is developer-run software focused testing, scoped to fully verify software requirements. Often this is called software formal qualification test (FQT). This may or may not use the full, complete system. Many times parts of the system are simulated or decomposed to increase the ability to stimulate inputs or observe outputs of the software.

This is the first activity where it is highly desirable to have a participant attend as a representative of the independent safety agency. The results need to be considered a primary data source for software safety. On-site involvement increases confidence when leaning heavily on a developer-run event.

Since this is a software requirements–based test, traceability from requirements should be evident. Specifically, identification of test cases aimed to verify safety requirements must be identified in test plan and test description documents. More rigorous testing should be executed on safety elements to ensure that off nominal or abnormal execution cannot result in a hazard. Examples could be fault injection, boundary condition testing, zero value testing, or input rate variation.

Software testing in this environment should measure the amount of software code that is exercised by testing, and justify any shortcomings, especially for untested safety critical code. More detailed expectations for developer software test planning and execution can be found in the JSSSH, section 4.4.1 Software Safety Test Planning and section E.13.3 Formal Test Coverage (JSSSC, 1999).

SAR analysis

The SAR is typically a PM responsibility in an Army setting. It can be considered a snapshot in time of the current system and its safety implications. A current description of system hazard analyses is mapped against the intended use, which could be a test event, demonstration, or field use. Required mitigations should be evident for those who will control the environment where the system is to be used.

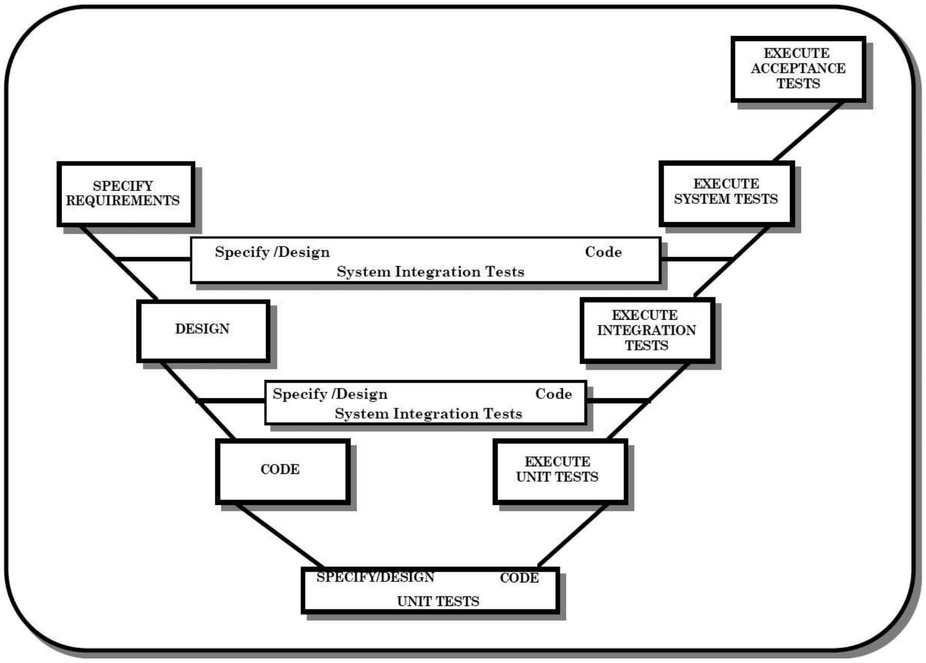

Figure 1. Levels of software testing (from Joint Software System Safety Committee (JSSSC) Software System Safety Handbook, section 2.6.4.2.2).

For software, there should be a description of the software safety approach, to include design constraints, coding standards, or other applicable software safety methodology. There must be a description of the safety characteristics of the specific hardware and software configuration intended for use. Open and residual hazards with causes allocated to software should be identified. Chapter 27 in Section II of the NAVSEA Weapon System Safety Guidelines Handbook is a potential resource when generating and updating a SAR (NAVSEA, 2006).

Software safety analyses summary

Without delivery of the artifacts in this section, the resulting analyses cannot be completed. The result of this information gap is that hazard contributions allocated to software should be considered to have elevated probability.

A second key point is that this information must be developed and delivered in order to lower software-related hazard probabilities even for "off-the-shelf" or prototype systems. The choice to buy products to perform safety critical functions or resource limitations does not preclude them from being assessed for their correct and safe performance.

Last, this is an iterative set of analyses. Any resulting outputs apply only to the configuration and environment for which the analyses were completed. As changes are made, each element should be revisited and considered for its impact.

Levels of software safety testing

Traditionally, upon delivery of the SAR, independent system level DT can begin. Verification of safety requirements should occur at the highest level of system integration possible; however, some safety critical software requirements will not be verifiable at the system test level.

As discussed previously, software FQTs are often a primary source for verification of software. Even at this FQT level, it will sometimes not be possible to fully verify all requirements of the software.

It is less desirable, but in some cases software integration tests and software unit tests can be used as data sources to achieve complete requirements or SPR fix verification. In this lower-level testing it is critical to identify assumptions and limitations of the test environment.

Although not preferred, static software code inspection can be used as a verification method for requirements not able to be verified via dynamic testing. Normally this method is not used as a primary verification source for a safety requirement but can be a good secondary source for additional verification confidence. This flow and scope of sources for safety testing is shown in *Figure 1*.

In addition to full verification of software requirements during developer testing, independent DT should include system level software-focused safety testing. This testing should be designed to compliment developer testing and address capabilities as well as

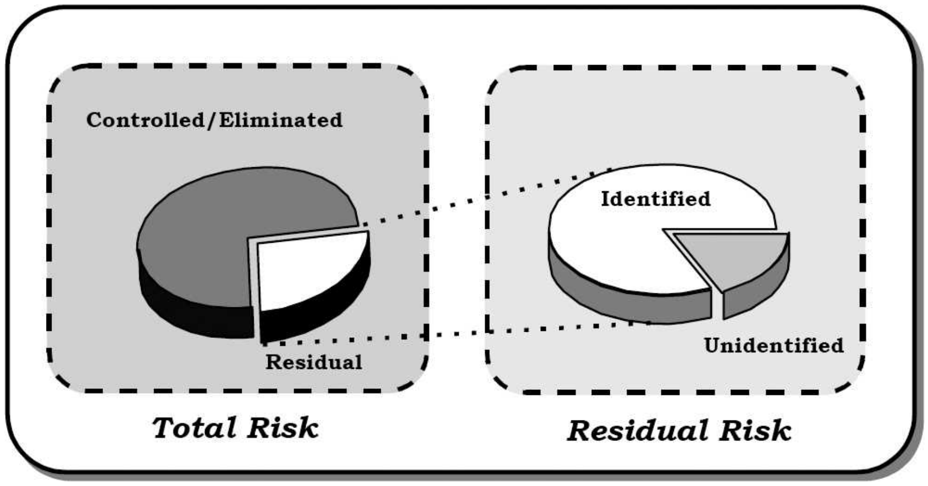

Figure 2. Total risk versus residual risk (from Joint Software System Safety Committee (JSSSC) Software System Safety Handbook, section 3.3).

requirements. System level software test design can focus time spent testing on safety-related software functionality. Training is key to allowing independent testers to identify expected test results. At this system level, exploratory testing is encouraged—often the defects not found during requirements-based testing are found during exploratory testing.

It is imperative to be aware that the primary method for showing safe software is verification of safety critical software requirements via test.

Residual risk identification

At the completion of Army DT, there is often a need to identify residual risks prior to soldier interaction with the system. This is done through DTC-issued safety releases and safety confirmations. Following analyses and test completion, residual risk can be derived from the total risk subset, as visualized in *Figure 2*.

For systems with safety critical computer software components, it is essential that any residual risk documents are limited to well-defined, tested configurations. Residual risk outputs should trace observed test results and safety analysis to identify residual system hazards. If the software is modified, the residual risk outputs must be updated or amended. Having requirements and design analyses completed will help properly scope any required regression testing without excessive budget or schedule impact.

A key difference in assessing residual risks relating to software is awareness of expected reliability. Software reliability considers errors resulting from specification, design, or implementation that are unfound during testing. Historically, software can be shown to meet a failure rate of no better than the 10^{-3} to 10^{-4} range according to the American National Standards Institute (ANSI) R-013-1992 and T/AST/046 (Nuclear Safety Directorate [UK], 2003). Goals for system safety often include showing 10^{-6} or better residual probability of mishap occurrence. Because of this, hazard mitigation relying on software should include other mitigation methods in hardware and/or procedural use to reduce probabilities to an acceptable level. Even with a most rigorous software development and test effort, the lower ends of mishap probability may be unreachable via software alone. Systems relying on software as the sole mitigation to a hazard should consider the residual risk if probability falls in the 10^{-3} to 10^{-4} range, which is labeled *remote mishap probability* in MIL-STD 882 (DoD, 1993). Initial design efforts can greatly reduce the amount and impact of safety critical software.

Application

The following are recent examples that show limitations associated with relying solely on system level testing to achieve confidence in system safety. Also, expected results of applying the existing DoD software safety guidance detailed in the previous sections are described.

Case 1—weaponized Unmanned Ground Vehicle (UGV)

In this example, a small, weaponized UGV system entered DT with weak system hazard analysis. Also, it had no hazard allocation to software, software requirement specification, software design documentation, or software requirements–based testing.

Actual results during DT included occurrence of multiple uncommanded motion events that were not prevented or halted by system software. Also, system-level software safety testing discovered that the system was capable of weapon fire via a switch other than the trigger.

The result via proper application of existing software safety guidance would have been earlier identification of elevated probability for hazards during test and field use associated with both uncommanded platform motion and uncommanded weapon firing.

Case 2—large, fast UGV

In this example, a 5,000-lb UGV, capable of teleoperation up to 50 mph, entered government DT with no hazard analysis, software requirement specification, software design documentation, or software requirements–based testing.

Actual results included system-level software safety testing, discovering that remote vehicle power loss left actuators in their previous state. The system was driving when power loss was induced. The result was continued UGV motion at the original speed with no method to emergency stop the UGV.

The result via proper application of existing software safety guidance would have been earlier identification of elevated probability for hazard during DT and field use associated with uncommanded motion.

Case 3—remote fire control system

In this example, a remote fire control system entered DT with incomplete system hazard analysis. Also software requirement specification, software design documentation, and evidence of software requirements–based testing were not delivered.

Safety findings were initially prepared solely using system level test results. Software safety artifacts were requested from the developer and delivered for analysis. Analytical results showed five additional system hazards, three of which had catastrophic severity, to have residual risk.

These risks did not manifest during testing and were not apparent without consideration of analytical outputs. Proper residual risk identification then allowed mitigations to be put in place to address the risks. Following iterations were able to further reduce the risks via design changes.

Application summary

Each of these systems required significant external mitigations to allow safe system testing. In all cases, significant safety issues were discovered late in the development cycle and resulted in costly rework to correct the problems. Fortunately, the problems in the first two cases were discovered through exploratory testing; however, consistency of problem discovery would increase significantly assuming a structured integration and test series leading up to system level DT. Each of the UGV systems had only system level test results as available inputs for safety decision makers, resulting in higher probabilities of residual hazard occurrence.

True value can be seen in case three, where hazards were identified analytically, mitigated via design when feasible, and accurately elevated for acceptance when needed.

Benefits to system safety

The primary safety benefits of consistently applying existing software safety guidance would be seen in several ways. Test safety would be increased by raising awareness of potential hazards during test. Those executing DT would not be exposed to unknown hazards.

Another critical benefit is that safety releases and safety confirmations could most accurately present residual hazards allocated to software. Appropriate mitigation techniques could be identified to safely maximize system functionality during testing, demonstrations, or field use.

In addition, the disciplines identified for generating software safety outputs are in line with current best practices for software engineering identified in the Institute of Electrical and Electronics Engineers (IEEE) 12207 "Standard for Information Technology–Software Life Cycle Processes" (IEEE, 1998) or the Software Engineering Institute (SEI) Capability Maturity Model® Integration (CMMI) (Carnegie Mellon, 2006). The added benefit would be promotion of software quality, software maintainability, software testability, and the discovery of software problems earlier in the life cycle when they are cheaper to fix.

Conclusions

System functionality executed by complex software has been increasing and will continue to grow in the future. System hazards related to software will be more important to quantify, while at the same time they cannot be exhaustively characterized through traditional system level safety testing. Those involved in T&E must be aware that DoD guidance is in place for software safety. Only through awareness and application of these practices to software specification, design, implementation, and testing can we ensure safe and appropriate tests resulting in credible residual hazard identification. ❑

Frank Fratrik is an electrical engineer at the US Army Aberdeen Test Center in Aberdeen Proving Ground, MD.

He has been performing test and analysis on software intensive systems for the past 5 years. Fratrik has been the system software test lead for unmanned ground vehicle systems with varied size and capability. His test work has also included software controlled systems such as: landmines, artillery fuzes, fire control systems, and other ground vehicle systems. Fratrik holds a B.S. in Electrical Engineering from Pennsylvania State University, University Park, PA and is also nearing completion of a M.S. in Systems Engineering from Johns Hopkins University, Baltimore, MD.

References

American National Standards Institute (ANSI). 1992. *ANSI R-013-1992, Software Reliability, 1992*. http://webstore.ansi.org/RecordDetail.aspx?sku=R-013-1992 (March 5, 2009).

Carnegie Mellon. 2006. *Capability Maturity Model (CMMI) CMMI® for Development, Version 1.2*. Carnegie Mellon Software Engineering Institute, August 2006. Pittsburgh, Pennsylvania: Carnegie Mellon Software Engineering Institute.

Department of Defense (DoD). 1993. *MIL-STD 882C, System Safety Program Requirements, 19 January 1993*. Wright Patterson Air Force Base, Ohio: HQ Air Force Materiel Command (SES). http://crc.army.mil/guidance/system_safety/882C.pdf (accessed March 5, 2009).

IEEE. 1998. *IEEE 12207, Standard for Information Technology-Software Life Cycle Processes*. New York, New York: Institute of Electrical and Electronics Engineers.

Joint Software System Safety Committee (JSSSC). 1999. *Joint Software System Safety Committee (JSSSC) Software System Safety Handbook, December 1999*. Washington, D.C.: Joint Services Computer Resources Management Group, U.S. Navy, U.S. Army, and the U.S. Air Force. http://www.system-safety.org/Document/Software_System_Safety_Handbook.pdf (accessed March 5, 2009).

NAVSEA. 2006. *NAVSEA Weapon System Safety Guidelines Handbook, SW020-AH-SAF-010, 1 February 2006*. Indian Head, Maryland: Naval Sea Systems Command Warfare Center Enterprise, Naval Support Facility, Indian Head.

Nuclear Safety Directorate [UK]. 2003. *T/AST/046, Technical Assessment Guide: Computer Based Safety Systems, (United Kingdom) Nuclear Safety Directorate—Business Management System, 10 January 2003*. http://www.hse.gov.uk/foi/internalops/nsd/tech_asst_guides/tast046.pdf (accessed March 5, 2009).

U.S. Army. 1992. *CECOM Software System Safety Guide, TR 92-02, May 1992*. Washington, D.C.: U.S. Department of the Army Headquarters.

U.S. Army. 1999. *FR/GE/UK/US International Test Operations Procedure (ITOP) 1-1-057 Safety Critical Software Analysis and Testing, 4 June 1999*. Washington, D.C.: U.S. Department of the Army Headquarters.

U.S. Army Aviation and Missiles Command (AMCOM). 1998. Software System Safety Policy, AMCOM Reg 385-17. Draft, March 15, 2008.