# Lessons Learned in a Complex Software Safety Program

*by Nathaniel Ozarin*
*Hauppauge, New York*

Development of a system software safety program was required as part of an effort to secure government safety certification of a complex and intrinsically hazardous software-controlled system under development by several contributing companies. The author was part of a team of software safety support engineers reporting to one of the contributing companies. This paper summarizes some of the highlights of the lessons learned during development of this program.

The initial challenge was to develop a practical and understandable Software System Safety Plan (SSSP) and the associated supporting documents that, together, addressed concerns from the program's governing guidance standards and handbooks, and provided a map to eventual certification of the subject system. In this development, the principal guideline document was MIL-STD-882E [Ref. 1] and two of its cited software-specific guidelines: *Joint Software Systems Safety Engineering Handbook* (JSSSEH) [Ref. 2] and *Guidance on Software Safety Design and Assessment of Munitions Related Computing Systems* (AOP-52) [Ref. 3] — a total of 653 pages in all, and far too much raw material for developers to follow during design processes. What was needed was a set of step-by-step procedures tailored to address the specific project's safety-significant issues, capturing the guidance rules for developers and software safety reviewers in their tasks. Development of such procedures is an ongoing evolutionary task for several reasons, including:

- No one can think of every safety issue in advance;
- Inevitable disputes arise about the applicability of many guidance statements;
- Changes to safety assessment details occur as the design evolves;
- Certification authority expectations are unpredictable;
- Customers often take issue with many conclusions of the safety assessment details.

Developing and applying these procedures and lessons learned also applies to the design of many devices in other fields, such as medical, automotive and household consumer devices.

## The Software System Safety Plan (SSSP)

The basic document for setting up and running a software safety program is MIL-STD-882, now in Revision E. The SSSP is the highest-level plan for developing software with operational hazards at acceptable levels of risk. In the subject development, no plan guidance was provided and the plan was assembled in contractor format. Along with the usual front matter and introductory material, the plan included a brief description of the software to be developed, a high-level schedule summary, a list of deliverable documents associated with the safety program and a summary of other tasks (reviews, assessments, resolution of hazards, etc.) that were described as team responsibilities. The plan included a list of definitions of terms used within, taken from MIL-STD-882's list if applicable to the program, but modified for program requirements. Next, an organizational chart and descriptions explained the assignment of ongoing project responsibilities based on title (e.g., the general responsibilities of the program manager, lead safety engineer and quality control engineer), but this information had little practical value to the development team — other than perhaps being useful as a one-time introduction of what the program expected you to do — because specific assignments were based on the master schedule and availability of personnel at any one time. The SSSP then cited the master schedule, maintained as a separate project document, followed by a lengthy section that described the software safety process as set forth in MIL-STD-882, but with focus on the primary safety issues of the specific system.

The next SSSP section described hazard analysis processes to be performed, including many tasks that were *never* performed (e.g., reliance on lessons learned, dissemination of historical lessons learned to various engineering disciplines and possibly others). This section described processes for functional hazard analysis – probably the most important part of the safety analysis process, followed by descriptions of requirements analysis, architectural design, detail designs, implementation and testing. Subsequent sections addressed deliverables, approach to certification by safety authorities, and testing.

While safety analysis for hardware systems and software systems follows the same basic procedures for each life cycle phase, the biggest difference is probably the recognition that failure rates cannot be realistically assigned to software failures. Whereas some sources suggest assigning hardware-based failure rates based on human experience and expectations — e.g., ARP5580 Table 8 [Ref. 4] provides examples of failure likelihood ranging from "very high" to "remote," and assigns corresponding

*Table 1 — Section Titles in AOP-52 and JSSSEH.*

| AOP-52 | | JSSSEH | |
|---|---|---|---|
| 4.2.2 | Failure in the Computing Environment | E.5.5.1 | Same title |
| 4.2.3 | CPU Selection | E.5.2 | Same title |
| 4.4 | Safety-Related Events and Safety-Related Functions | — | — |
| 4.7.2 | Computer/Human Interface Issues | E.9.1.1 | CHI Issues |
| 4.12.1 | General Testing Guidelines | E.12.1 | Same title |
| 4.12.2 | Trajectory Testing for Embedded Systems | E.13.2 | Same title |

failure rates to these descriptions (MIL-STD-882 Table II uses "frequent" to "improbable" or "eliminated," but does not provide corresponding numerical figures). MIL-STD-882's approach is to use a software module's autonomy as the basis for predicting risk — software that can fire a missile without human intervention ("autonomous") has the most potential risk, while software that cannot do anything potentially hazardous on its own ("no safety impact") has the least risk. MIL-STD-882 provides a table with five such levels of autonomy.

To avoid confusion, MIL-STD-882 defines three safety adjectives applied to hardware and software, with practical meanings used in this paper as follows:

- **Safety-critical** means associated with hazards that can cause serious harm to people, things, or the environment.
- **Safety-related** means associated with minor hazards of minimal consequence.
- **Safety-significant** means either of the two previous terms.

When developing an SSSP, contractors usually put together government-required plans in accordance with Statement of Work (SOW) requirements with one goal in mind: getting customer acceptance. Yet, a plan's real purpose is to guide the development team to achieving goals in a clear, step-by-step fashion; the plan should be written for the *staff*, not the *customer*. There's no need for hype about the contractor's capabilities and experience, or about how much they've thought about things such as customer benefits. That information belongs in the proposal. Rather, the plan should be relatively short, concise, and clear — and aimed at the engineering staff. A common exception in high-level plans is providing an up-front design description summary — principally diagrams — for the benefit of customer reviewers whose introduction to the project is the plan itself. Such descriptions should be at high levels because reviewers will not spend much time with these to get any deep understanding.

**Lessons:**

- Keep it simple and minimize the effort.
- Know that plans will change due to both customer comments and design evolution — don't try to make each revision perfect.
- Avoid duplicating what is in other documents; cite references where it makes sense to do so.

## Guidance Documents: Extracting Essentials

Specific safety requirements of the JSSSEH and AOP-52, the two software guidance documents cited in MIL-STD-882, apply to all development phases. These requirements appear in AOP-52 Section 4 ("Generic Software Safety Design Requirements") and in JSSSEH Appendix E ("Generic Software Safety Requirements And Guidelines"). Requirements in the two documents are largely identical for practical purposes.

**Lesson:** We put each AOP-52 requirement and each JSSSEH requirement side by side on a spreadsheet to identify differences and to highlight where further discussion was required to determine applicability and the need for tailoring.

One exception is AOP-52's section 4.4, which does not provide requirements; rather, it is a 600-word essay on how to identify safety-related hazards. The JSSSEH omits this.

The JSSSEH is newer (August 2010) and longer (334 pages) than the AOP-52 (March 2009, 205 pages). It is very clear that JSSSEH Appendix E ("Generic Software Safety Requirements and Guidelines") heavily borrows from — and is based on — AOP-52 Section 4 ("Generic Software Safety Design Requirements"). The JSSSEH made many modifications to the borrowed statements to clarify and expand upon them. Many other JSSSEH statements are identical to AOP-52 statements (aside from grammatical corrections), except where the JSSSEH replaced "must" by "shall" and replaced "safety-related" by "safety-critical."

Nearly all AOP-52 Section 4 requirements containing "must" or "shall" are included in JSSSEH Appendix E. A few "should" statements are not.

Aside from requirement statements, both the JSSSEH and AOP-52 include some lengthy discussions and checklist-type review questions to consider (see Table 1). In general, JSSSEH text in these sections is an update of AOP-52 text.

**Lessons:** These sections generally do not contain "must" or "shall" but it is helpful for Software Development Plan (SDP) authors to review the JSSSEH versions because some of the material might be applicable to the SDP. Peer review checklist authors should do the same for development of peer review checklists.

MIL-STD-882, the governing safety document, refers to the JSSSEH and AOP-52 simply as guidance sources, so the safety reviewers need to identify and extract project-specific guidance statements and incorporate them — tailored as appropriate — in the SDP and in the safety review checklists. There is a natural conflict here, however: Reviewers need to understand design architecture and details to do the extractions, but the safety assessments should also be done before the design goes too far. For practical purposes, we have found that the architectural design should first be laid out (subject to evolutionary change, of course), then the guidance statements should be extracted.

**Lesson:** At the code level, to be thorough, we found it best to extract all statements. Those that are not applicable should be identified as such in the SDP and in the review checklists.

**Lesson:** To provide traceability from project requirements back to the generic requirements, and as a means to review all tailoring changes, list all generic requirements in a spreadsheet along with tailored versions for comparison, and include the rationale for tailoring.

**Lesson:** Some generic requirements in guidance documents contain multiple statements, whereas well-written requirements should be limited to a single observation for test purposes and for tracking in a requirements database. Example (JSSSEH E.3.12, System Errors Log): *"The software shall make provisions for logging all system errors. The operator shall have the capability to review logged system errors. Errors in safety-critical routines shall be highlighted and shall be brought to the operator's attention as soon after occurrence as practical."* We broke that into three requirements, and added a project-specific fourth and fifth, putting "shall" in upper case to highlight that each is a mandatory, testable requirement:

- The software SHALL make provisions for logging all system errors.
- The operator SHALL have the capability to review logged system errors.
- Detected errors in routines affecting system safety SHALL be brought to the operator's attention as soon after occurrence as practical.
- Detected errors in routines affecting system safety SHALL be highlighted in the system error log.
- Detected errors in safety-critical routines SHALL cause automatic system transition to a safe state as soon as practical.

The spreadsheet included requirement-specific comments where additional discussion was necessary to further tailor requirements — in this case, notably that "soon as practical" isn't testable and that corresponding requirements would need to be modified to specify reaction times for specific kinds of errors.

## Identifying Top-Level hazards

**Lesson:** MIL-STD-882 does not require a list of top-level hazards, but we found this listing useful. The idea is to maintain a list of all possible safety hazards — those caused by failures of hardware or software, or unexpected human interactions — to assure that (1) each software hazard in the required functional hazard analysis (FHA) can be traced up to one or more top-level hazards, and (2) each top-level hazard potentially caused by software can be traced down to at least one underlying software failure in the FHA. Since this was a software project for the author's development team, top-level failures that could not be caused by software (or human activities monitored by software) were identified as such, and these were to be addressed independently by the hardware development team. To complicate matters, the hardware team included its own software developers for certain hardware-specific tasks.

**Lesson:** Identifying and assigning responsibility for resolution of the one top-level hazard list among multiple teams requires good coordination among teams.

Identification of top-level hazards is a group effort. Generic hazards include those potentially causing injury from mechanical devices or components, electrical shock, excessive levels of acoustic noise, excessive levels of electromagnetic radiation, generation of toxic substances, excessive heat or cold, fire and others. Generic hazards also include damage to things, including the project's own equipment. The team should think carefully about whether software could cause a generic hazard and, if so, include the software function in the hazard tracking system.

Many project-specific hazards on the top-level hazard list might occur while the system does what it's supposed to do — but with unexpected or incorrect timing, output levels, aim, indications, output messages, human interaction, etc. The starting point is to list everything the subject system is supposed to do (i.e., each system software function in the Functional Hazard Analysis), then hypothesize failure modes for each function and identify whether each failure mode can cause a hazardous condition.

## Software Requirements and Safety Issues

This author once believed that software safety issues were just a subset of software reliability issues — a subset of failures that just happened to affect safety. This is emphatically not the case, principally because software-based safety hazards can occur while software is doing exactly what it's required to do. Obviously, the problem is that the requirements may not fully address project-specific safety issues. For example, it may be that software creates an unexpected operational hazard when system prime power is lost for a fraction of a second (perhaps a full restart may not be triggered), software may do the wrong thing when given a bad input from failed hardware or (more likely) when a human operator does a combination of things during a certain system state that no one ever considered. The last example would be a sneak circuit problem in the hardware world — a hazard arising from combinations or sequences of unexpected (and unanalyzed) inputs, given the condition that there are no failed components. However, the challenge is not to exhaustively consider unexpected input combinations or sequences that can cause hazards in the top hazard list; rather, it is to determine whether each hazard affected by software in the top-level hazard list can be caused by unexpected inputs. A software module involved in this way should be subject to one or more functional requirements, added to the requirements list as necessary, to prevent unexpected inputs from causing outputs with potentially hazardous consequences to the extent that can be implemented feasibly. (AOP-52 section 4.12.10, "Operator Interface Testing," states "*Operator interface testing must include operator errors during safety-related operations to verify safe system response to these errors.*" That's okay, but inadequate, because it requires only that some scripted errors be tested.)

"The design team should consider lessons learned from previous developments in their collective experiences, from histories of related developments, from news reports of tragedies caused by software failures (and why they failed) and from one's imagination. What are the possible ways this particular system could cause harm, and how could the system's software unexpectedly cause harm during development, factory tests, installation set-up, proper operation, improper operation, self-test, upgrade, fault isolation procedures and normal maintenance?"

There are several sources to consider in the effort to develop project-specific safety requirements. The design team should consider lessons learned from previous developments in their collective experiences, from histories of related developments, from news reports of tragedies caused by software failures (and why they failed) and from one's imagination. What are the possible ways this particular system could cause harm, and how could the system's software unexpectedly cause harm during development, factory tests, installation set-up, proper operation, improper operation, self-test, upgrade, fault isolation procedures and normal maintenance? Each possible hazard belongs on the top-level hazard list.

## Using Functional Hazard Analysis (FHA)

A Functional Hazard Analysis (FHA) is a failure modes and effects analysis of software functions capable of causing hazards [Refs. 5 and 6]. We developed a detailed FHA on a Microsoft Excel spreadsheet to consider each safety-significant software functional requirement (many software functional requirements had no effect on safety and didn't belong there), how it could fail, and what to do about it. The FHA listed each software-critical and software-related system function by ID, name, and description, then possible failure modes (e.g., fails to _____, runs too soon, too late, intermittently, etc.), effects at the next higher level and at the system level, then values for parameters in MIL-STD-882 Tables I through V (Severity, Probability [placeholding guesses], Risk Assessment, Software Control Category [degree of autonomy], and Software Criticality).

**Lesson:** For each software function, we supplied only Probability and Control Category, and used Excel's VLOOKUP function to automatically supply the other three from tables copied from MIL-STD-882 to another worksheet in the same workbook. We also used Conditional Formatting so that cell colors in VLOOKUP cells automatically corresponded to cell colors in the MIL-STD-882 tables. Finally, the FHA included a Mitigation column to describe how the hazard could be controlled.

**Lesson:** It is desirable to limit the quantity of system-level effects to a meaningful minimum set (it always happens that different team members create many unnecessary entries by describing the same effect in different ways, and often declaring different severity values to the same effect). To limit the number of system-level effects and multiple severity values, we listed each effect, arranged in logical groups, on a separate worksheet and identified each with an ID number and a severity value (1 through 4 per MIL-STD-882 Table I). We entered these IDs in the FHA sheet and used VLOOKUP in the System Effects column and in the Severity column to copy the corresponding descriptions and severities from the system effects sheet to the FHA sheet.

**Lesson:** Many failure modes were mitigated by the same mitigating features or techniques. To save effort and provide consistency, we listed each mitigation description on a separate worksheet and identified each with a mitigation ID number. We entered these IDs in the FHA sheet (with some failure modes listing multiple mitigation IDs) then ran a macro that read the IDs and copied the corresponding mitigation descriptions from the mitigation sheet to the FHA sheet.

**Lesson:** Interpreting the terms "safety-critical" and "safety-related" can be difficult. The guidance documents are riddled with these two terms — with both terms usually appearing together. How to determine whether a software function is a safety-critical or safety-related one? And is the distinction useful? MIL-STD-882 defines these terms and uses them to clarify definitions of software autonomy in Table IV, but makes no statements as to how safety analyses should treat them differently. JSSSEH also defines the terms (same as MIL-STD-882), but it sometimes uses "safety-related" to mean either term (e.g., section 4.4.3) and it also makes no statements on how safety analyses should treat them differently. While a safety-critical function, by definition, can lead to catastrophic or critical hazards, it's the distinction between "Catastrophic" and "Critical" that leads to different Level of Rigor (LOR) efforts (MIL-STD-882 Table V). Similarly, a safety-related function, by definition, can lead to "Marginal" or "Negligible" hazards, but it's the distinction between these two levels that makes a difference in the safety effort. So, while the design team can decide whether each software function in the FHA is either safety-critical or safety-related, what is needed in the FHA is classification of software function's severity as Catastrophic, Critical, Marginal, or Negligible.

The project required delivery of a Software Hazard Analysis (SHA), a text document to expand on key findings of the FHA and discuss proposed mitigations and resulting effects at the top level. The SHA included a description of the system and its operations, a list of its software components, a table of all software requirements with corresponding safety significance and corresponding JSSSEH requirement IDs, a table of all AOP-52 requirements with statements of compliance and rationale for instances of noncompliance, a similar table of JSSSEH requirements, analysis of particular hazards from the FHA and proposed mitigations, checklists for design reviews for each phase of development, and a table of functional requirements and expected methods of verification.

## Considering Safety-Significant Effects of Partitioning and Redundancy

Early high-level designs are typically shown as block diagrams in which software is divided into modules. Part of the design process at this stage is identifying the safety-significant modules so that safety analyses will be applied only where needed. Safety-Significant Modules (SSM) must be partitioned from other modules in the design phase and it must be shown that non-SSM functions cannot affect safety-significant functions in SSM (the SHA is a good place for capture). The selected operating system must be designed to work with a partitioned system, allocating separate memory spaces for SSM and non-SSM, and providing means for messaging between them.

**Lesson:** Highlight proposed safety features on the diagram (e.g., health monitoring of critical modules, hazard detection and automatic shutdown capabilities, manual override interfaces). A good approach to be sure (and able to convince certification authorities) that all safety-significant software has been identified is to perform a Functional Hazard Analysis (FHA) — which would probably be done anyway — to determine each functional failure that can cause a system-level hazard (each of which appears in the top-level hazard list), then use Fault Tree Analysis (FTA) to determine the software parts that could contribute to each such functional failure.

Selection of an operating system to best support system safety (e.g., providing memory partitioning if relevant to the design) isn't trivial and requires research.

**Lesson:** A paper trail is important for eventual certification. Make a separate report that describes the pros and cons of each operating system under consideration, along with system application history and engineering experience (if possible) and explain your conclusions. Provide data sheets as appendices. Also, be sure to attribute manufacturers' claims to the manufacturers, rather than as statements of fact. Don't expect certification authorities to help you select an operating system, but expect that they may want to know how you reached your conclusions.

It's impractical to provide redundancy in all subsystems and operations, but as a goal sensors that affect software safety should be redundant. Dependence on a single sensor without a readily available override capability can be catastrophic [Ref. 7].

## Understanding MIL-STD-882 Level of Rigor

The idea of Level of Rigor (LOR) is to be sure that parts of software that underlie safety hazards receive the level of attention during development that is commensurate with the hazard's criticality, where criticality is a combination of operational autonomy and hazard severity. For each critical software module (or whatever unit of software makes sense; MIL-STD-882 refers to "exact software contributors to hazards and mishaps"), you determine the software's autonomy and the hazard's severity. Then you use MIL-STD-882 tables to determine a "software criticality index" (SwCI) value, which in turn defines risk as a final hazard rating. This rating ranges from SwCI 1 to SwCI 5 (corresponding to High, Serious, Medium, Low, and "Not Safety," respectively). The SwCI value is the basis for determining LOR. An uncaptioned table following Table V in MIL-STD-882 states the "minimum" efforts required for each SwCI; the development team is responsible for appropriate tailoring. Results may go into the SSSP — typically most clearly represented as displayed in Table 2.

*Table 2 — Safety-Specific Analysis and Test Requirements.*

| | Analysis | | | | Testing | |
|---|---|---|---|---|---|---|
| SwCI | Requirements | Architectural Design | Detailed Design | Implementation | In-Depth | Functional |
| SwCI 1 | X | X | X | X | X | — |
| SwCI 2 | X | X | X | — | X | — |
| SwCI 3 | X | X | — | — | X | — |
| SwCI 4 | — | — | — | — | — | X |
| SwCI 5 | — | — | — | — | — | — |

In this table, the SSSP must define carefully what "Analysis" and "In-Depth" (versus "Functional") safety testing means so developers know exactly what to do. Details may be provided in the SDP detailed design checklists for each development phase.

While this schedule of efforts is the MIL-STD-882 minimum, realistically the design process for any kind of structured development will include reviews (analysis) for all parts of the system software and all four development phases — simply because the non-safety parts of the software system are also expected to operate reliably. There's no substitute for reviews using checklists and testing using procedures devised by independent test engineers. One exception is that SwCI 5 software needn't be reviewed for safety issue.

## Understanding Practical Limitations to Testing Safety-Significant Requirements

Typically, a good chunk of critical code — 25% is a good estimate — is devoted to the detection and handling of unexpected events (not just built-in-test [BIT] code, but events such as local detection of bad message data, out-of-range values, timing anomalies, plus error logging, etc.). Unfortunately, not all of that code can be tested in feasible ways (e.g., it is usually not feasible to simulate memory errors to test code that looks for such errors, or interrupts from external sources that occur in an unexpected sequence). Where functions are not testable, analysis must provide a convincing explanation of why code will perform as expected. This analysis typically consists of careful code review by independent reviewers and, where possible, should include a description of unit testing and results.

JSSSEH 4.4.1.3 lists the types of robustness tests that *should* be performed. While not strictly required, all safety-significant software should indeed be subject to both analysis and robustness tests to determine software and system behavior when operating in conditions beyond those specified — abnormal inputs, excessive data traffic, unexpected operator actions, arithmetic exceptions and overflow, and timing variations (such as unexpected message and interrupt sequences [Ref. 8]). Some development programs allow time for an operator to try to break a system rendered harmless for this purpose (if you can be sure that it is truly harmless) by intentionally applying unscripted abnormal commands, or just playing with it. Video recording of operator actions during such free play may be useful for capturing actions that really cause a hazardous event.

## Setting Up a Hazard Tracking System (HTS) to Resolve Safety Issues

An HTS is required by MIL-STD-882, and the SSSP describes the project's specific HTS details. The customer initially wanted an HTS using a relational database with links among hazards, requirements, mitigations, resolutions and changes. However, the HTS was implemented on a spreadsheet that proved to be perfectly adequate. The HTS grouped together multiple FHA hazards resulting in the same system-level hazard and listed them on one row of the spreadsheet. For example, there were several functional failures in which software failed to detect specific operational faults and force the system to a safe state. These functions were therefore listed on the same row, with a new summary hazard ID and a description of the common system-level hazard. Other headings included identification of the subsystem owning the failed software, causal factors, effects, a list of top-level failures that might result, system operating modes during which the hazard might occur, associated safety requirements, MIL-STD-882 table information (see the previous FHA discussion), and status information (working group results, resolutions, etc.).

## Dealing With Certification Authorities

The author was tasked with the development of a Software Safety Certification Plan that "details the Safety Certification process and requirements to meet Safety Certification requirements" of several certification authorities. Unfortunately, there were no guidance documents that outlined the plan contents.

**Lesson:** Knowing that the well-known software certification plan called *Plan for Software Aspects of Certification* (PSAC), part of DO-178B that is required for

Federal Aviation Administration (FAA) certification of airborne software [Ref. 9]), provides clear and thorough certification guidelines, the author used the PSAC outline as a starting point, tailoring details as appropriate for the ground-based subject system software. Certification authorities were generally pleased with the results.

Part of the certification process is presenting materials to certification authorities to show that all safety considerations have been made and that all safety processes are being followed, with a paper trail to document these activities. In other words, their job is to assure that you've thoroughly addressed all development requirements for software safety. Contractors may also ask questions that further explain what authorities are looking for.

**Lesson:** While authorities will have a keen interest in your design, don't expect them to bless any part of it, or to advise that one approach is better than another, because they obviously don't want to share responsibility for future problems. As with any review board, different people have different expectations, so you should plan to perform additional safety efforts if requested.

**Lesson:** Presentation materials shouldn't have a marketing slant (highlighting company capabilities, experience and cleverness) and shouldn't provide more material than necessary. Too much effort is often spent on unnecessary background material, overly elaborate slides and detailed tabular information that is unreadable as a slide. Authorities may politely ignore all this material, but its development and the internal review process with many reviewers can take a lot of expensive time.

## Value of Independent Review

For the program described in this paper, the government apparently required an independent review of the software development. The author was one of the reviewers.

**Lesson:** Independent reviewers with appropriate backgrounds (in this case, reviewers employed by another company) can make a significant difference in the software safety process, in part because they bring additional experience and insights from their own software development backgrounds. They can also be more objective than development team members because they can scrutinize guidance document requirements and check for design compliance without taking time from the design and development schedule.

A potential conflict is that reviewers may assert that it is necessary to review additional design aspects, or request that the development team perform additional safety efforts beyond what is planned (or budgeted). The solution is to follow the SSSP carefully and augment it with additional tasks if it is required by the governing documents. Any proposed changes should also be cleared with certification authorities.

## About the Author

Nathaniel Ozarin served as a senior engineering consultant at Omnicon (www.omnicongroup.com), a company specializing in reliability and safety analysis for the military, medical, industrial and transportation industries. His background includes hardware engineering, software engineering, systems engineering, programming and reliability engineering. He received a bachelor of science degree in Electrical Engineering from Lehigh University, a master's degree in electrical engineering from Polytechnic University of New York, and an MBA from Long Island University. He is a member of the Institute of Electrical and Electronics Engineers (IEEE) and was named Reliability Engineer of the Year by the IEEE Reliability Society in 2009. ⬣

## References

1. MIL-STD-882E. "Department Of Defense Standard Practice, System Safety," 2012.
2. *Joint Software Systems Safety Engineering Handbook* (JSSSEH), Naval Ordnance Safety and Security Activity, Indian Head, MD, 2010.
3. *Guidance on Software Safety Design and Assessment of Munition-Related Computing Systems* (AOP-52), North Atlantic Treaty Organization, 2009.
4. SAE ARP5580. "Recommended Failure Modes and Effects Analysis (FMEA) Practices for Non-Automobile Applications," Society of Automotive Engineers, Inc. 2000.
5. Ozarin, N. W. "Failure Modes and Effects Analysis during Design of Computer Software," *Proceedings of the Annual Reliability and Maintainability Symposium*, 2004.
6. Ozarin, N. W. "Applying Software Failure Modes and Effects Analysis to Interfaces," *Proc. Ann. Reliability & Maintainability Symposium*, 2009.
7. Travis, Gregory. "How the Boeing 737 Max Disaster Looks to a Software Developer," *IEEE Spectrum*, April 18, 2019.
8. Beatty, Sean M. "Improving Software Safety: Finding the Defects that Testing and Inspection Miss," *Proceedings of the 22nd International System Safety Conference*, 2004.
9. DO-178B, "Software Considerations in Airborne Systems and Equipment Certification," RTCA, 2013.