Article development led by **acmqueue**
queue.acm.org

Avionics software safety certification is achieved through objective-based standards.

BY B. SCOTT ANDERSEN AND GEORGE ROMANSKI

Verification of Safety-Critical Software

AVIONICS SOFTWARE HAS become a keystone in today's aircraft design. Advances in avionics systems have reduced aircraft weight thereby reducing fuel consumption, enabled precision navigation, improved engine performance, and provided a host of other benefits. These advances have turned modern aircraft into flying data centers with computers controlling or monitoring many of the critical systems onboard. The software that runs these aircraft systems must be as safe as we can make it.

The Federal Aviation Administration (FAA) and its European counterparts, along with the major airframe, engine, and avionics manufacturers worked together to produce guidance for avionics software developers culminating in the document *Software Considerations in Airborne Systems and Equipment Certification*⁶ published in the United States by the nonprofit organization RTCA as DO-178B and in Europe by EUROCAE as ED-12B. The guidance

in DO-178B is in the form of objectives and activities that must be met or performed to earn certification for the software product.

A safety assessment and hazard analysis helps determine the Design Assurance Level (DAL) for the software by characterizing the effects of its failure on the aircraft, crew, and passengers. There are five DALs (quoted directly from DO-178B and FAA Advisory Circular AC 25.1309-1A):¹

- **Catastrophic:** Failure conditions that would prevent continued safe flight and landing.

- **Hazardous/Severe-Major:** Failure conditions that would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be: (1) a large reduction in safety margins or functional capabilities, (2) physical distress or higher workload such that the flight crew could not be relied on to perform their tasks accurately or completely, or (3) adverse effects on occupants including serious or potentially fatal injuries to a small number of those occupants.

- **Major:** Failure conditions that would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a significant reduction in safety margins or functional capabilities, a significant increase in crew workload or in conditions impairing crew efficiency, or discomfort to occupants, possibly including injuries.

- **Minor:** Failure conditions that would not significantly reduce aircraft safety, and that would involve crew actions that are well within their capabilities. Minor failure conditions may include, for example, a slight reduction in safety margins or functional capabilities, a slight increase in crew workload, such as routine flight plan changes, or some inconvenience to occupants.

- **No Effect:** Failure conditions that do not affect the operational capability of the aircraft or increase crew workload.

DO-178B identifies a set of Software Levels Definitions A through E that roughly correspond to the DALs, with level A software the highest criticality and level E the lowest. The objectives to be met depend upon the software level assigned to the project. This article discusses activities and objectives associated with level A.

Software Engineering 101

In the preface of Nancy Leveson's *Safe-ware: System Safety and Computers*, the author cites "One obvious lesson is that most accidents are not the result of unknown scientific principles but rather of a failure to apply well-known, standard engineering practices. A second lesson is that accidents will not be prevented by technological fixes alone, but will require control of all aspects of the development and operation of the system."⁴ Definition and control of the software development process is the first key to creating safety-critical systems.

The objectives and activities identified in DO-178B should be familiar to anyone who was attentive during software engineering courses in college. The key points are that activities and work-products are defined and repeatable. A software life cycle model must be identified, and transition criteria between processes must be documented. In short, things are written down.

There are plans and standards used by software development and verification teams, and those activities are audited to ensure compliance with those plans and standards. These plans include:

- **Plan for Software Aspects of Certification (PSAC).** The primary means of communicating the overall project plan to the certification authority.

- **Software Development Plan (SDP).** Defines the software life cycle and development environment.

- **Software Verification Plan (SVP).** Describes how the software verification process objectives will be satisfied.

- **Software Configuration Management Plan (SCMP).** Describes how artifacts will be managed.

- **Software Quality Assurance Plan (SQAP).** Describes how the SQA (Software Quality Assurance) function will ensure plans are followed and standards are met.

DO-178B demands three standards be present: requirements, design, and source code.

A software configuration management (SCM) system consistent with the SCMP must be provided. Additionally, an issue-tracking or bug-tracking system must be provided to document issues with either the software or compliance to standards and processes. All of these activities, plans, and standards (with the exception of the PSAC) are

things that a well-run organization at SEI³ level 2 or 3 would have in place.

No Surprises

What does it mean to be *safe*? If complete safety were defined as the complete absence of hazards, then it would be difficult to imagine any practical and useful real-world system ever being completely safe. The safeness of a particular system is, therefore, a relative notion defined by the identified potential hazards within the system, the likelihood of each hazard's occurrence, and the effectiveness of the mitigations put in place for each of those hazards.

In order to inventory and understand the hazards for a system, you must first understand what the system is supposed to do. Otherwise, how can you tell if it is doing it wrong (or what might result)? A strong set of requirements is necessary for the foundation of any safety-critical system. Further, the requirements will appear at various levels of abstraction beginning as high as the requirements for the airframe and descending into systems, line-replaceable units, CSCIs (computer software configuration items) for a particular subsystem, high-level requirements for that subsystem, and low-level requirements. The particular levels of abstraction present will depend on the particulars

of the aircraft and the project, but the presence of multiple levels of requirements and abstraction is common.

Returning for a moment to the question of what it means to be *safe*, a portion of that answer could be summed up as “no surprises.” The software should perform such that all of the requirements are fulfilled. Further, there should be an *absence of unintended function*. That is, the software should do only what is specified in the requirements: no more, no less. You do not want to be learning about a hidden, extra feature of the software during an emergency at 30,000 feet.

Traceability

To say that “the system should have an absence of unintended function,” prompts the question: unintended by whom? As requirements are developed at lower levels, the requirements statement must include more details and specifics. Lower-level requirements should and must provide additional value to the implementers, or they would just be restatements of their parent requirements. How can you differentiate between something that has been added to a lower-level requirement and something that is just a decomposition or elaboration of a higher-level requirement?

Managing the relationships between levels of requirements (and other project artifacts) is done through a system called *traceability*. Traceability is a mapping between requirements or other project artifacts that provides a navigable relationship between two or more items. For example, a high-level software requirement can be traced to one or more low-level requirements. Where such a mapping shows a decomposition and elaboration of the higher-level item (and no new behavior), no additional safety analysis is necessary. If a lower-level item cannot be directly traced to a higher-level item, then the possibility exists that this lower-level item has introduced an unintended function. In such a case, the lower-level (unmapped) item should be subjected to a safety assessment.

Traceability provides a means of following the mappings from the highest level of requirements down through each level of abstraction to the lowest level. From the lowest levels

of software requirements traceability continues to software design artifacts, source code, verification methods and data, and other related artifacts. You should be able to start with a system-level requirement and follow each related requirement in turn through the traceability mapping down to the related code and verification data.

Reviews and Independence

Each requirement and other project artifact must be subject to review, and the review should be based on criteria identified in the project planning documents. Additionally, depending on the criticality of the software as defined by its Software Level Definition, the review of a specific artifact might need to be done with *independence*, defined in DO-178B as the “separation of responsibilities, which ensures the accomplishment of objective evaluation. For software verification process activities, independence is achieved when the verification activity is performed by a person(s) other than the developer of the item being verified, and a tool(s) may be used to achieve an equivalence to the human verification activity. For the software quality assurance process, independence also includes the authority to ensure corrective action.”

The DO-178B guidance does not specify how reviews must be performed. Some organizations hold meetings with several (or many) reviewers, collect minutes, and sign the review as a group. Anyone who has witnessed or participated in a group review will likely attest that the efficacy of such a review is highly dependent upon the acumen of those doing the review and the culture of the organization holding the review. In short, the danger here is that if everybody is responsible, then nobody is responsible.

Verocel, a company that provides software verification services, takes a different approach to organizing reviews. Instead of assigning groups of engineers to a given review, it assigns a single engineer who is solely responsible for the completeness and correctness of the artifact and its review. It is not unusual for the assigned engineer to solicit assistance from other members of the group or even the originator of the artifact to answer questions, provide further analysis or insights, or

obtain clarifications. In the end, however, only one signature appears on the review checklist, and responsibility for the quality of the artifact and its review rests with the one person who signed.

Mechanics of Artifact Organization and Traceability

There are practical considerations in the organization of project artifacts and the traceability between them. Often, projects have hundreds or even thousands of requirements. How should these items be managed? How can the traceability between them be maintained? Again, DO-178B provides a set of objectives but is silent on how these objectives should be met. It is up to the development team to provide details for how each objective will be met in the project planning documents, including artifact organization and traceability. The development team and the certification authority (or its representative) must agree on these plans.

Several commercial offerings are available for requirements management. Some organizations use word processors and spreadsheets (along with some specific procedures) to meet these objectives. Verocel found all of these approaches wanting and developed its own requirements and artifact management system with a relational database as its backing store. This system, called VeroTrace, manages requirements text directly and holds references to other artifacts such as design components, source files, test procedures, and test results, which are maintained in the CM (configuration management) system. VeroTrace fulfills the navigable traceability objective. Once again, DO-178B is not prescriptive on these matters. A valid organizational and traceability system could conceivably be created with little more than a stack of paper index cards. As a practical matter, large software development projects require automation of some type to manage the project artifacts, their review state, and their associated traceability.

A Good Requirement

The DO-178B objectives for a good requirement are related to: (a) compliance with system requirements, (b) accuracy and consistency, (c) compatibility with the target computer, (d) verifiability, (e) conformance to standards, (f) traceabil-

ity, and (g) algorithm aspects. DO-178B provides specifics for each of these topics. The “conformance to standards” objective suggests that there are standards with which to conform. Indeed, project planning documents will likely include standards and development guidelines for requirements, design components, coding standards, test development standards, and other aspects of software development. Having these guidelines and standards documented provides a means for SQA to assess whether the applicable processes are being followed and to demand corrective action if they are not.

The requirement development standards contain additional criteria. For example, a requirement should have a unique identifier so that it can be unambiguously referenced both in its review and by other requirements or artifacts through traceability. A requirement should have some version identifier so that a change can be recognized and impacted relationships can be assessed. The requirement author must be identified or independence of review cannot be guaranteed. The review of a requirement must identify the reviewer for the same reason. A DO-178B objective often generates a secondary group of implied activities and objectives that must be met to fulfill the original objective.

This might sound like an excessive amount of work, and it certainly is a lot of work to do correctly. The intent of this system is to provide a set of solid requirements with traceability between those of the same level of abstraction (where necessary) and between levels of abstraction to ensure the absence of unintended function. Only then can you begin to assert the software is safe.

Waterfall

A software development process known commonly as “the waterfall model” generally follows these steps: identify all the requirements, complete the architecture and design documents, then write the code. Virtually no system of significant size can be built this way. The guidance in DO-178B acknowledges this and states the following in Section 3: “The guidelines of this document do not prescribe a preferred software life cycle, but describe the separate processes that comprise most life cycles and the interaction between them. The separa-

Software should do only what is specified in the requirements: no more, no less. You do not want to be learning about a hidden, extra feature of the software during an emergency at 30,000 feet.

tion of the processes is not intended to imply a structure for the organization(s) that perform them. For each software product, the software life cycle(s) is constructed that includes these processes.” Further, DO-178B Section 12.1.4d states, “Reverse engineering may be used to regenerate software life cycle data that is inadequate or missing in satisfying the objectives of this document.” In short, it is important to meet all the objectives specified by DO-178B, but the order in which those objectives are met is not dictated.

That said, the FAA had certain concerns about the practice of reverse engineering and commissioned George Romanski (Verocel, Inc.) and Mike DeWalt (then of Certification Services Inc.) to research the problem. (DeWalt has since returned to the FAA as chief scientific and technical advisor for aircraft computer software.) The study, *Reverse Engineering Software and Digital Systems*, determined that 68% of those surveyed had used some sort of reverse engineering on a project. The report uncovered a number of other interesting facts, but the upshot for this discussion is that the software development process need not be “waterfall” to be successful.²

Validation and Verification

Validation: “Are we building the right product?” *Verification*: “Are we building the product right?”⁵ The FAA commissioned the reverse engineering study because of the obvious concern that a reverse engineering effort might simply restate what the code was doing rather than determine what the code *should* be doing. Validation—determining if we are building the right product—is still an important component of the safety process, and there are no shortcuts to achieving this goal.

The reverse engineering study used data provided by Verocel from 13 projects with a total of 250,000 e-LOC (effective lines of code). (For C programs, for example, an e-LOC excludes blank lines, comment lines, lines with only a single bracket, else, or other keyword.) Problems reported within these projects were apportioned among the following categories: design error, comment error, documentation error, error-handling problems, test errors, structural coverage problems, modified functionality, requirements errors, and

code errors. Details on *how* these problems were found were also included: review, analysis (manual inspection used to reverse engineer artifacts), observation (manual inspection not related to the specific artifact being worked on), beta test/functional test, structural coverage analysis, or system test.

Of all the problems found, 77% were discovered by engineers using engineering judgment, including 54% found by analysis, 8% by observation, and 15% by review. That is, three-quarters of problems with the software were found by engineers looking hard to determine if the software is doing what it *should* do. A successful verification effort can be done only on a set of project artifacts that has been through a successful validation effort.

Software Verification

Software verification can be accomplished by any of several means or a combination. The most common means is by test. Requirements-based testing uses a set of requirements as the basis for the test criteria and produces results that conclusively report whether the software under test fulfills those requirements.

A second common verification method is *analysis*. Certain requirements are difficult or impossible to verify by test. For example, if a requirement demands that interrupts be locked and a critical section formed during a particular operation, then interrupt locking may be impossible to detect from the outside. In this case, it is appropriate to produce a small analysis document that provides evidence that this requirement is fulfilled.

Test and analysis are by no means the only verification options available. For example, *formal methods* can be used to prove correctness of an algorithm or fidelity to the software requirements. For small systems such as a pressure sensor, *exhaustive input* testing can be used to fully cover the input space for the software. Even product service history can be used in certain limited situations.

Structural Coverage Analysis

DO-178B says the following about SCA (structural coverage analysis): “The objective of this analysis is to determine which code structure was not exercised by the requirements-based test procedures...The structural coverage analysis may be performed on the Source Code, unless the software level is A and the compiler generates object code that is not directly traceable to Source Code statements. Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences. A compiler-generated array-bound check in the object code is an example of object code that is not directly traceable to the Source Code.”

The idea behind SCA is simple: if testing is complete and the software fulfills the requirements completely, then one would expect 100% of the code would be executed during the test. There are reasons why this might not be true, however. For example, robustness checks in the software might have execution paths that are unreachable because it is impossible under normal testing conditions to create the exceptions they guard against.

Gaps in the SCA might also indicate other problems: shortcomings in the requirements-based test cases or procedures, inadequacies in the software requirements, or *dead code*, which is code that cannot be traced to any requirements. Dead code should be removed from the system or the requirements should be updated if that function is necessary for the software.

Deactivated code is not intended to be executed during flight, although it has corresponding requirements and should be present in the software. (Perhaps this code is used only for ground maintenance activities.) It is necessary to show that deactivated code cannot be inadvertently executed in a mode where the code is not intended to run.

In addition to the activities showing coverage of source or object code in the software under test, further analysis must be performed to confirm the data coupling and control coupling between the code components. The output of the compiler is not trusted. Testing and structural coverage analysis is used to confirm that source code and object code are fully covered and any discrepancies are remedied or documented. Similarly, the linker (or any other reference fix-up mechanism) is also not trusted. Proper coupling and fix-up must be shown to be correct. SCA data along with control coupling data are reviewable artifacts.

The accompanying figure shows the web of traceability from HLR (high-level requirement) to SCA report and its review checklist. Dashed lines indicate traceability. Note that this diagram is only representative (and not complete). For example, documents used for verification by analysis are not shown. Even so, this gives a good general overview of a typical web of traceability.

Development and Verification Tools

As previously mentioned, neither the compiler nor the linker is trusted fully. These are considered *development tools* because they create artifacts that fly. It is possible to create *qualified development* tools whose output need not be checked. The guidance for the creation of such a tool (and its verification) is the same as the guidance for the creation and verification for software that flies. At times, however, the extraordinary burden of creating that full set of

Sample traceability.

verification artifacts is worth it. The output of a qualified development tool does not need to be subject to further verification efforts.

A less intense effort is required for the creation of a *qualified verification* tool. Such a tool can be substituted for a human reviewer on the targeted artifacts. Qualified verification tools require only requirements, requirements-based tests, and appropriate documentation as outlined by DO-178B. As an example, Verocel has a qualified verification tool called VerO-Link that helps with the control coupling analysis.

The Role of the DER

In the U.S. the certification authority is the FAA, but the FAA does not supply engineering resources to each project developing software for aircraft. Instead a system of DERs (designated engineering representatives) has been developed that allows the FAA to delegate responsibility for project oversight to specially trained and properly certified engineers who work within the industry, either for an independent firm or even for the manufacturers themselves. The FAA always does the final sign-off for a project, but the DERs do the lion's share of the work following the progress of the project and evaluating the materials produced by the project.

DERs assigned to the project will hold a series of four stages of involvement (SOI) meetings at various stages of the project. The first of the meetings, SOI-1, covers the plan for certification and reviews materials that are key to ensure that the project will proceed properly. This involves verifying that the processes and procedures necessary for a successful certification effort are in place, that there is a good and solid plan for meeting each of the identified DO-178B objectives, and that there is good agreement between the DER and the software developer on all important aspects of the project. The point of the first meeting is to ensure that a plan for certification is in place and that everyone agrees that if the plan is followed and the artifacts are produced, then there should be no impediments for achieving certification.

A second meeting, SOI-2, reviews any open issues from the first meeting, assesses the impact of any variations from

the plan, and explores the quality of the materials developed thus far in the project. The minimum project state for having a successful SOI-2 is 50% of the requirements developed and reviewed, 50% of the design artifacts completed and reviewed, 50% of the source code completed and reviewed, and traceability between all of these artifacts. The purpose of this meeting is not to assess or review the completed set of requirements or other artifacts but to identify any problems early in the process so that corrective action can be taken while changes are relatively inexpensive and minimally disruptive. The volume of materials required to be ready for such a meeting may be more or less than the 50% identified here, as it is completely between the DER and the development team to set goals and milestones.

A third meeting, SOI-3, takes place when development is complete and the verification effort is approximately 50% complete. Verification can be done by any number of mechanisms including test, analysis, formal methods, or any other approach described and approved within the project plans for aspects of certification.

The final meeting, SOI-4, happens when all certification evidence has been produced for the completed project, and the review is primarily to assess the readiness of the package for final certification assessment.

A DER is in an adversarial role. He or she is also in a position to be an asset, assisting the project team in identifying flaws in their plans, defects in their processes, or even as a facilitator who can help the project get out of a bind by making suggestions or identifying options that the team has not identified themselves. A DER should not be placed in a position of developing process plans; the DER's job is to evaluate and not to develop materials that he or she would later be evaluating. A good DER is tough as nails and helps ensure the project produces software of high integrity.

Wrapping Up

DO-178B is not prescriptive, but it is comprehensive in its criteria and objectives. Not all of what DO-178B demands can be captured in a relatively short article, but we have highlighted the main points about how DO-178B provides

guidance to help produce safe software. Good requirements, vetted to ensure they match the intent of what the software should do, must be developed. These requirements and all software development artifacts (including designs, code, tests, test results, and structural coverage data) are connected through a web of navigable traceability that helps ensure each of the requirements is fulfilled by the software and that there is no unintended function. On the other end, structural coverage analysis identifies code not covered by requirements-based tests, and dead code (code not associated with any requirements) can be flagged for removal.

All of this is presented with a backdrop of solid and documented engineering practices and a mature software development environment. With proper procedures and controls in place, it is possible to create software upon which one would trust their safety and the safety of others.

Related articles on queue.acm.org

Interview: A Conversation with Jim Ready
<http://queue.acm.org/detail.cfm?id=644261>

There's Still Some Life Left in Ada
Alexander Wolfe
<http://queue.acm.org/detail.cfm?id=1035608>

Best Practice (BPM)
Derek Miers
<http://queue.acm.org/detail.cfm?id=1122688>

References

1. Federal Aviation Administration. *System Design and Analysis*, Advisory Circular AC 25.1309-1A (June 21, 1998).
2. Federal Aviation Administration. *Reverse Engineering Software and Digital Systems*, April 2011 [Draft Report]. To be published by the FAA William J. Hughes Technical Center.
3. Humphrey, W.S. *Managing the Software Process*. SEI Series in Software Engineering. Addison-Wesley Publishing Co., Reading, MA, 1990.
4. Leveson, N. *Safeware: System Safety and Computers*. Addison-Wesley Publishing, Reading, MA, 1995.
5. Rakitin, S.R. *Software Verification and Validation for Practitioners and Managers*, second ed. Artech House, Norwood, MA, 2001.
6. RTCA. *Software Considerations in Airborne Systems and Equipment Certification* (DO-178B), 1992.

B. Scott Andersen is a principal software engineer for Verocel, Inc. He has more than 30 years of experience in the software industry, most recently as a member of the Jini development team at Sun Microsystems helping to build a distributed computing model for Java. .

George Romanski is a co-founder and president of Verocel, Inc. He has specialized in the production of software development environments for the past 40 years. His work has focused on compilers, cross-compilers, runtime systems, and tools for embedded realtime applications.