SYNOPSYS®

WHITE PAPER

Coverity: Risk Mitigation for DO-178C

Gordon M. Uchenick, Lead Aerospace/Defense Sales Engineer

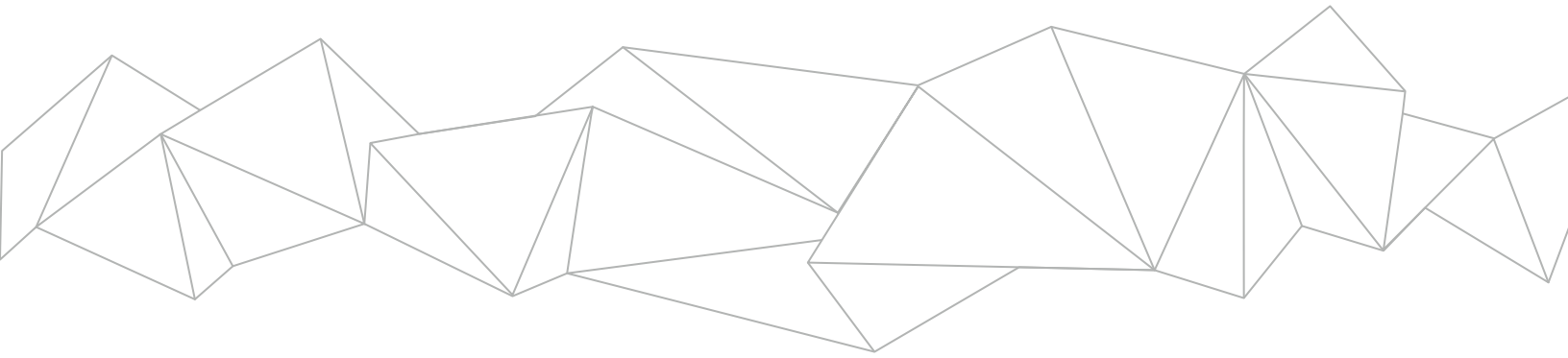


Table of contents

Mission accomplished, but at a cost

Airborne systems and equipment have become increasingly software-intensive since the early 1980s. Yet accidents and fatalities caused by software are almost never heard of in civil aviation. This safety record is particularly impressive when you consider that the latest generation of aircraft systems run millions of lines of code. Civilian airborne software has performed so reliably because it is required to meet the objectives defined in the DO-178 standard. Industry and government DO-178 practitioners deserve equal credit for this admirable track record, a result of their shared culture of "never compromise safety."

Historically, DO-178 has been successful at ensuring safety, but not without significant impact on budgets and schedules. A recent certification, for example, had approximately 1,300 software requirements. (For reference, a test procedure is typically generated for every 2–4 requirements.) The requirements, in addition to design, code, and test documents, must all be written and reviewed via checklists. In typical cases, the cost of DO-178 certification can range from $25 to $100 per line of code—that's $2.5 million to $10 million for 100,000 lines of code! The meticulous scrutiny with which auditors inspect the most critical code can double those costs and inject additional uncertainty into the release schedule. Fortunately, tools such as Coverity by Synopsys, a comprehensive static analysis solution, allow developers to increase productivity and reduce risk by finding and fixing software defects earlier in the development life cycle, thus simplifying the compliance process. Coverity is uniquely positioned as a leader in The Forrester Wave™: Static Application Security Testing, Q4 2017,* and in the 2018 Gartner Magic Quadrant for Application Security Testing.†

In typical cases, the cost of DO-178 certification can range from $25 to $100 per line of code—that's $2.5 million to $10 million for 100,000 lines of code!

DO-178 overview

DO-178, whose formal title is Software Considerations in Airborne System and Equipment Certification, was first published in 1981 by the Radio Technical Commission for Avionics (RTCA), a U.S. nonprofit public-private partnership that produces recommendations on a wide range of aviation issues. The purpose of DO-178 is "to provide guidance for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements."[1] The European Organisation for Civil Aviation Equipment (EUROCAE) contributed to DO-178, and the joint EUROCAE designation for the standard is ED-12C. To keep up with advances in technology, RTCA released Revision A in 1985, Revision B in 1992, and the current Revision C in 2011.‡

The standard has been adopted worldwide by government agencies responsible for civilian airborne system and equipment certification. Military programs can elect to use DO-178 and often do so for airborne software derived from commercial products. Partner nations in international joint programs may have requirements for DO-178/ED-12C certification, an important consideration in foreign military sales.

* Available at https://www.synopsys.com/software-integrity/resources/analyst-reports/2017-sast-forrester-wave.html.

† Available at https://www.synopsys.com/software-integrity/resources/analyst-reports/2018-gartner-magic-quadrant.html.

‡ DO-178 and other RTCA publications are available at https://my.rtca.org/nc__store.

DO-178C defines 10 processes of the software life cycle and categorizes several objectives within each process, as shown in Table 1.

Process	Number of objectives
Software planning	7
Software development	7
Software requirements	7
Software design	13
Software coding	9
Integration	5
Software verification	9
Software configuration management	6
Software quality assurance	5
Certification liaison	3

Table 1. DO-178C software life cycle processes and objectives.[2]

DO-178 also defines five software assurance levels, from the most rigorous, level A, used for the inspection of the most critical airborne code, to level E, which describes software whose failure would not have any effect on the safe operation of the aircraft (see Table 2). Levels are assigned to each software unit as the result of a system safety assessment process. The standard also defines for each level which objectives are applicable and which must be satisfied "with independence"—that is, by someone other than the developer.

Level	Definition
A	Software failure results in a catastrophic failure condition for the aircraft
B	Software failure results in a hazardous failure condition for the aircraft
C	Software failure results in a major failure condition for the aircraft
D	Software failure results in a minor failure condition for the aircraft
E	Software failure has no effect on operational capability or pilot workload

Table 2. DO-178 software levels.[3]

Cost and risk management through early defect detection

The economic benefits of early defect detection are well-known and well-documented. In commercial applications, it costs about 85% less to remediate a defect if it is found during the coding phase rather than after product release.[4] Post-release defects are far costlier to remediate in safety-critical applications than in commercial applications

(easily as much as 100× costlier) because safety-critical software failures can cause physical damage or even fatalities, which are likely to become the grounds for costly litigation and significant brand damage.

According to NIST, it costs about 85% less to remediate a defect if it is found during the coding phase rather than after product release.[5]

In Section 4.4, "Software Life Cycle Environment Planning," DO-178C states that the goal of planning objectives is to "avoid errors during the software development processes that might contribute to a failure condition"[6] and that it recognizes the benefit of early defect detection. It is noteworthy that DO-178C specifically mentions software development processes, because development must occur before integration or validation.

DO-178C does not prescribe how to meet any of its objectives. Neither static analysis nor any other tool or technology is required or recommended. However, DO-178C does recognize that tools are a necessary part of the software life cycle. Coverity has proven its value in early defect detection and risk reduction in all vertical marketplaces. In fact, developers of mission-critical, safety-critical, and security-critical software were among the earliest adopters of Coverity, especially for embedded systems, where resolving problems after product release is particularly challenging.

Software development life cycle integration

Coverity can be integrated into the software development life cycle (SDLC) in three ways, each one with increasing effectiveness and, therefore, return on investment:

Analysis after coding is complete

Coverity is used to perform a software audit before the code proceeds to the next phase of the SDLC. While this method will find software defects, its disadvantage is the length of time between defect creation and detection. By the time defects are triaged, the original developers likely will have already moved on to other tasks, which they must stop so they can remediate the issues. This upsets the schedule in two separate tasks, not just the one containing the defect.

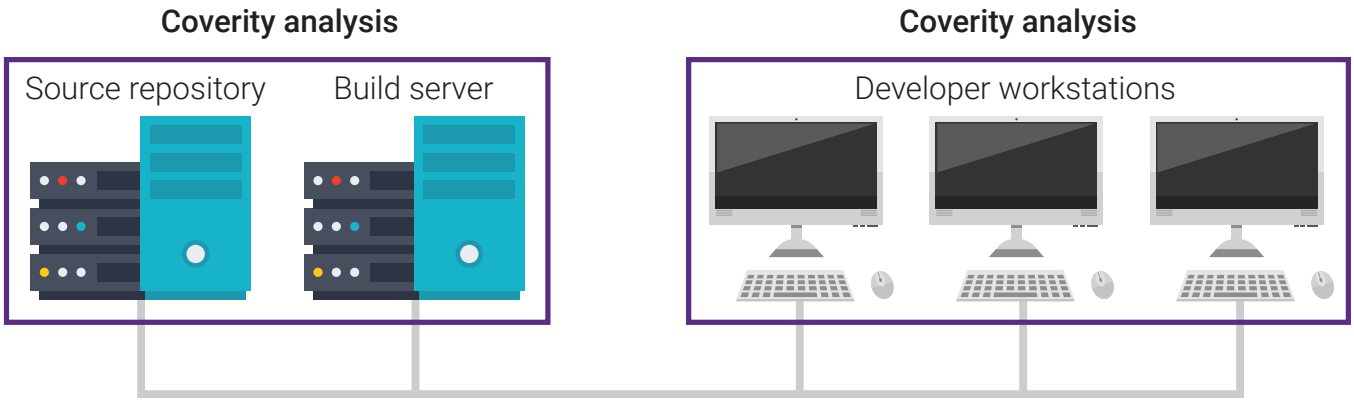
Analysis integrated into periodic builds

It is more productive to integrate Coverity into the project's periodic builds (typically nightly or weekly). Since Coverity analyzes the code every time it is built, it reports defects as soon as they are introduced into the codebase. Coverity can also automatically assign ownership of a defect and immediately dispatch a notification to the developer who created it. In this way, developers can address defects while the tasks are still fresh in their minds, with less effort than if they had to spend time recreating the thought processes and understandings used when first writing the code.

Ad hoc analysis on the developer's desktop

Running Coverity on an ad hoc basis on the code on the developer's desktop, before code review or commit, provides additional benefits:

- **More effective review.** Developers can ensure their code is as clean as possible before review by either their peers or the team's subject matter expert. Consequently, the reviewers can spend more time ensuring the code accurately and completely implements low-level software requirements, the real purpose of DO-178.
- **Fewer testing cycles.** Defects are mitigated to the greatest possible extent before any commit to the source code repository. This methodology often reduces the total number of testing cycles required because defects never enter the codebase; instead, they are addressed before each build.
- **Refined developer techniques.** When developers frequently review Coverity's findings, they improve their skills as they learn to recognize unanticipated gaps in both error handling and data edge conditions. As developers look at the defects Coverity finds, they refine their coding techniques and thus are less likely to produce recurrences of the same defect type, out of a sense of pride in craftsmanship and personal responsibility.[7]



Coverity in a safety-critical development workgroup

Enforcing a safe coding standard

The most common languages used in airborne software, C and C++, were not designed with safety as a primary goal of the syntax. DO-178C recognizes that some language features are inappropriate for use in safety-critical applications and states that meeting its objectives "may require limiting the use of some features of a language."[8]

A significant requirement of DO-178C is to create and enforce a software coding standard. It is sensible to begin with existing standards that are well-established for producing mission-critical and safety-critical code. We recommend you start with the following literature:

- **For C:** JPL Institutional Coding Standard for the C Programming Language (JPL, March 3, 2009).§
- **For C++:** Joint Strike Fighter Air Vehicle C++ Coding Standards (Lockheed Martin, December 2005).¶ The JSF standard's coding rules are aggregated from well-respected programming literature and the well-established Motor Industry Software Reliability Association (MISRA) standard. The JSF standard provides a rationale for each rule.

These standards have been used successfully to develop hundreds of millions of lines of code in mission-critical and safety-critical systems. You may use them with confidence as the basis for your own coding standard, and Coverity's analysis features can help you enforce the most important parts.

§ Available at https://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_C.pdf.

¶ Available at http://www.jsf.mil/downloads/down_documentation.htm.

We also recommend examining the MISRA C and C++ compliance standards** because they provide a hierarchy of rules and identify where deviations from the rules are allowed. While this compliance hierarchy is initially useful, we recommend that you choose only those MISRA rules that are relevant to the project at hand. Coverity has built-in and configurable features that detect and report MISRA violations, thus allowing you to automate that part of software coding standard enforcement, saving valuable time and personnel resources in review cycles.

Reverse-engineering artifacts

DO-178C outlines 22 documentation datasets that plan, direct, explain, define, record, or provide evidence of activities performed during the SDLC. It does not matter when the documentation is written if it is accurate and complete when the review is started. In a perfect waterfall world, all the planning, standards, requirements, design, and test case documentation would exist before the first line of code was written. Other documents, such as test results and configuration data, would then emerge from the software development life cycle itself.

In the real world, however, a significant portion of the required documentation typically must be reverse-engineered from the code, for the following reasons:

- **Documentation is not the developer's primary skill.** Developers are hired because they write good code, not because they write good documentation. Moreover, they typically despise documentation tasks. They would much rather spend their time developing, and will therefore put the minimally acceptable level of effort into any activity that gets in the way of writing code.
- **The codebase integrates previously developed code.** This type of code includes:
  - Previously developed code from a noncertified codebase
  - Previously developed code from a codebase certified at a lower level
  - Commercial off-the-shelf (COTS) code purchased from an external supplier
  - Custom code developed by a supplier
  - Open source components
- **The development baseline must be upgraded.** Regardless of any component's pedigree or provenance, all the code in a project must meet the objectives of DO-178C, including presentation of the required artifacts. If these artifacts do not exist, they must be reverse-engineered from the code to satisfy the guidance laid out in Section 12.1.4, "Upgrading a Development Baseline."

Document revision loops inject schedule risk and financial uncertainty into the certification effort.

Reverse-engineering artifacts is most effectively done by experienced DO-178C documentation specialists. As these specialists write artifacts, they often find issues in the code, which they report back to the developers. These issues may not be coding errors; they could be usages of the language that the experts know, from experience, will be questioned by reviewers. If the report leads to code changes, then parts of the SDLC must be repeated. Depending on the issue, documents that were thought to be complete might require revision. These revision loops inject schedule risk and financial uncertainty into the certification effort because in most cases, DO-178C artifact specialists are contractors and the contractor agreement specifies significant charges for exceeding a certain number of revision loops.

** Available at https://www.misra.org.uk/Publications/tabid/57/Default.aspx.

Coverity provides effective risk mitigation against the costly revision loops described above. When used during development, Coverity empowers developers to submit cleaner code to artifact writers. Consequently, code revisions are minimized because questionable or erroneous code constructs have already been identified by Coverity and addressed by the original developers. Coverity has even shown significant value when used at the eleventh hour, when the submitting company was getting dangerously close to the revision loop limit.

Tool qualification

DO-178C significantly expands on the concepts of tool qualification over previous revisions. Tools must be qualified when they are used to eliminate, reduce, or automate processes and the tool's output is not verified manually or by another tool.[9] The standard defines three criteria for distinguishing between types of tools:

Criterion 1. The output of the tool is part of the airborne software. The tool could introduce an error in the software.

Criterion 2. The tool automates part of the verification process. It replaces or reduces the use of other verification or development processes. It could fail to detect an error in the software.

Criterion 3. The tool automates part of the verification process but does not replace or reduce the use of other verification or development processes. It could fail to detect an error in the software.

DO-178C's companion document DO-330, Software Tool Qualification Considerations, outlines five tool qualification levels, from TQL-1 (the most rigorous) to TQL-5 (the least rigorous). The tool's criterion, combined with the software level A–D, determines the required tool qualification level. (Tool qualification is not needed for software level E.)

An important consideration is that when a tool is qualified, that qualification applies only for its use on the system being certified. If the same tool is to be used on another system, it must be requalified in the context of that other system. However, there are provisions in DO-330 for the reuse of previously qualified tools.[10]

Qualifying Coverity

Since Coverity is a verification tool, not a development tool, it meets criterion 2 or 3, depending on how it is used. If criterion 2 applies ***and*** the software level is A or B, then Coverity must be qualified at TQL-4 to comply with DO-178 guidance. Otherwise—that is, if criterion 3 applies and/or the software level is C or D—tool qualification at TQL-5 is sufficient.

It is possible to apply DO-330's objectives to Coverity to help claim credit for using an analysis tool. DO-330 defines tool qualification for Coverity, a commercial off-the-shelf (COTS) product, as a cooperative effort between Synopsys and the developer organization. Synopsys' contribution is tailored to the specific requirements of each qualification effort and is provided by the Software Integrity Group's services organization.

Coverity qualification at TQL-5

At TQL-5, Synopsys provides detailed documentation on the tool's operational requirements. The system developer then documents how the tool meets the software life cycle objectives defined in DO-178C's Plan for Software Aspects of Certification and other detailed planning documents.[11] Testing and verification of the tool as installed is also a cooperative effort. Synopsys provides the required test cases and the test execution procedure; then the developer organization runs the test suite in the specific development environment and records the results.

Coverity qualification at TQL-4

COTS tool qualification at TQL-4 is significantly more rigorous than qualification at TQL-5. The level of effort for both Synopsys and the developer organization is much higher, and therefore, qualification at TQL-4 should be

considered carefully on a case-by-case basis, starting with a discussion between the developer and the Synopsys services group.

Formal analysis as an alternative certification method

A significant update in DO-178C is that it discusses alternative methods for obtaining certification credit, such as the use of formal methods, which is typically employed when an extremely high level of confidence is required. Formal methods are a rigorous analysis of a mathematical model of system behaviors intended to prove the model is correct in all possible cases. What formal methods can't verify, however, is that the mathematical model correctly and completely corresponds both to the physical problem to be solved and to the systems implemented as the solution. Unless the code is automatically generated from the mathematical model by a formally verified development tool, correspondence between the mathematical model and the source code must be verified by manual methods.

Attempting to formally reason about the properties of a completely integrated large system is not practical today, especially as code sizes are growing ever more rapidly, for these reasons:

- Formal analysis of large code bodies isn't practical with respect to the computing resources required or the run time. Even on a very powerful computing platform, proving the correctness of a large mathematical system model can take days. So formal methods are often limited to proving only the most critical components of a complete model, such as key functional blocks of a microprocessor or cryptologic module.
- It is also important to understand that even if each component is highly assured, the combination of those components into a total system does not yield the same level of assurance. The principles of composability (i.e., after integration, do the properties of individual components persist, or do they interfere with one another?) and compositionality (i.e., are the properties of the emergent system determined only by the properties of its components?) are at the leading edge of formal software assurance.

Thus, formal methods should not be considered an alternative to static analysis; rather, formal analysis provides added insurance that a single critical module performs correctly under all possible conditions.

Putting it all together

Picking the right analysis tool is important because it will have a significant effect on development efficiency and the DO-178C certification schedule. In this paper we've discussed how using Coverity static analysis during code development and before reverse-engineering certification artifacts from the code has proven to increase productivity while simultaneously reducing budget and schedule risk. But it is typical for procurement policies to require the consideration of multiple suppliers for software tools. Synopsys welcomes competition and, in that spirit, provides the following vendor-neutral guidelines for establishing your selection criteria:

Do's

- Install, or have the vendor install, the candidate tool for a test run in your environment. (Synopsys does this regularly with Coverity, for free.)
  - Verify that the tool works in your development environment.
  - Verify that it interfaces with your software repository and defect tracking systems.
  - Verify that it is compatible with your software build procedures and other development tools, such as compilers, integrated development environments (IDEs), and so on.
  - Verify that managing and updating the tool will not impose an unacceptable workload on IT staff.

- Run the tool over your existing code.
  - Determine whether the defects reported are meaningful or insignificant. Allocate some time for your subject matter experts to perform this task, because a proper assessment requires a systemwide perspective.
  - Determine whether the tool presents defects in a manner useful to developers. There should be more information than "Problem type X in line Y of source file Z." The tool should disclose the reasoning behind each finding, because very often the fix for a defect found in a line of code is to change lines of code in the control flow preceding that defect.
- Verify that the tool is practical.
  - Verify that it runs fast enough to be invoked in every periodic build.
  - Determine whether you can run it only over modified code relative to the baseline while still retaining context or whether it is fast enough to analyze all the code all the time.
  - Determine whether you can implement and enforce a clean-before-review or clean-before-commit policy.
- Determine the false-positive (FP) rate.
  - Focus on your own code. Do not accept an FP rate based on generic code or an unsubstantiated vendor claim.
  - Decide an acceptable FP rate for your process. An unacceptable FP rate wastes resources and erodes developer confidence in the tool itself. A very significant finding can be obscured by meaningless noise.
- Investigate training, startup, and support options.
  - Inquire about the vendor's capability to provide on-site training relevant to your SDLC.
  - Verify that they can provide services to help you get started with the tool quickly and smoothly.
  - Verify that their support hours correspond to your workday.
  - Verify that they have field engineering staff if on-site support becomes necessary.

Don'ts

- Don't evaluate tools by comparing lists of vendor claims about the kinds of defects that their tools can find, and don't let a vendor push the evaluation in that direction. Comparing lists of claims regarding defect types isn't meaningful and leads to false equivalencies. Capabilities with the same name from different vendors won't have the same breadth, depth, or accuracy.
- Don't waste time purposely writing defective code to be used as the evaluation target. Purposely written bad code can contain only the kinds of defects that you already know of. The value of a static analysis tool is to find the kinds of defects that you don't already know of.
- Don't overestimate the limited value of standard test suites such as Juliet.†† These suites often exercise language features that are not appropriate for safety-critical code. Historically, the overlap between findings of different tools that were run over the same Juliet test suite has been surprisingly small.
- Don't base your evaluation on a "hunt for the golden bug." In other words, don't require that a static analysis tool find the defect in version ***n–1*** of your software that was the reason for creating version ***n***. Because you were recently burned by that defect, you're on the lookout for it. But once again, an important value of a static analysis tool is to find the kinds of defects that you aren't already looking for.

In summary, DO-178 certification provides assurance that the certified code meets its requirements with an appropriate level of confidence, because the cost of failure can be unthinkable. While the certification process is deliberately painstaking, the use of static analysis tools like Coverity eases much of the struggle. For more information on how Coverity has proven its value in other vertical marketplaces, visit www.synopsys.com/SAST.

†† Juliet Test Suites are available at https://samate.nist.gov/SRD/testsuite.php.

References

1. DO-178C, Software Considerations in Airborne Systems and Equipment Certification, RTCA, 2012, p. 11.
2. Table data adapted from ibid., Annex A, pp. 95–105.
3. Ibid, p. 14.
4. Planning Report 02-3, The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST, May 2002, p. 7-14.
5. Ibid.
6. DO-178C, p. 27.
7. Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira, An Overview on the Static Code Analysis Approach in Software Development, 2018, p. 4.
8. DO-178C, p. 75.
9. Ibid., p. 84.
10. DO-330, Software Tool Qualification Considerations, RTCA, 2011, p. 59.
11. Ibid., p. 62.

THE SYNOPSYS DIFFERENCE

Synopsys helps development teams build secure, high-quality software, minimizing risks while maximizing speed and productivity. Synopsys, a recognized leader in application security, provides static analysis, software composition analysis, and dynamic analysis solutions that enable teams to quickly find and fix vulnerabilities and defects in proprietary code, open source components, and application behavior. With a combination of industry-leading tools, services, and expertise, only Synopsys helps organizations optimize security and quality in DevSecOps and throughout the software development life cycle.

For more information, go to www.synopsys.com/software.

SYNOPSYS®

185 Berry Street, Suite 6500
San Francisco, CA 94107 USA

U.S. Sales: 800.873.8193
International Sales: +1 415.321.5237
Email: sig-info@synopsys.com