BLACK DUCK®

WHITE PAPER

COVERITY RISK MITIGATION FOR DO-178C

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

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-178C software levels

MISSION ACCOMPLISHED, BUT AT A COST

Since the early 1980s, airborne systems and equipment have become increasingly software intensive. Yet accidents and fatalities caused by software are almost unheard of. This safety record is particularly impressive when you consider that the latest generation of combat and commercial aircraft systems run over 6 million lines of code. One reason for this success is that airborne software must meet the rigorous 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 compromising safety.

The Radio Technical Commission for Aeronautics (RTCA) DO-178 is the de facto standard for certifying software used in safety-critical airborne systems and equipment. It defines objectives, processes, and criteria to ensure software reliability, compliance, and traceability throughout the development life cycle. DO-178 helps organizations mitigate risks, improve software quality, and meet regulatory requirements.

While DO-178 has been successful at ensuring safety, certification is expensive and time-consuming. 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 increases costs and injects additional uncertainty into release schedules.

DO-178C OVERVIEW

DO-178, whose formal title is Software Considerations in Airborne System and Equipment Certification, was first published in 1981 by the 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 creation of software for airborne systems and equipment that complies with safety requirements. The European Organization for Civil Aviation Equipment (EUROCAE) contributed to DO-178, and the joint EUROCAE designation for the standard is ED-12C.

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 foreign military sales certification.

DO-178C, the current version of the document, was published in January 2012. DO-178C defines 10 processes of the software life cycle and categorizes several objectives within each process, as shown in Table 1.

DO-178C 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 which objectives are applicable for each level, and which must be satisfied "with independence"—that is, by someone other than the developer.

COST AND RISK MANAGEMENT THROUGH EARLY DEFECT DETECTION

In safety-critical software, postrelease defects are often far costlier to remediate than in commercial products. That's because safety-critical software failures can cause physical damage or even fatalities, which can result in costly litigation and significant brand damage.

DO-178C 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" 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.

Coverity® Static Analysis from Black Duck 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 provides comprehensive static analysis that empowers development teams to deliver safe and reliable software at scale. Scans can be performed throughout the early stages of the software development life cycle (SDLC) to uncover defects when they're least disruptive and easiest to resolve.

Real-time analysis as developers code

Coverity plugs into developer IDEs to identify defects as code is being written, so issues can be resolved before they're committed. It identifies code quality and security issues using fast scanning techniques, without requiring developers to switch tools or manually invoke a scan. Detailed defect descriptions and actionable remediation guidance are included with each issue, enabling developers to resolve issues quickly.

Automated scans triggered by source code management (SCM) events

For programming languages that don't require a full build, static analysis scans can be triggered to run automatically on all pull requests. This identifies defects that are introduced by any new or changed code. Issue details and remediation guidance are included as comments on the pull request, enabling development teams to quickly assess the criticality of the issue without needing to switch tools.

Periodic comprehensive analysis

Comprehensive static analysis scans should be run periodically to identify more-complex issues that haven't been discovered by previous testing. A full project scan provides deep analysis of the entire codebase and can uncover even obscure, hard-to-find issues that span multiple files. Scan results can be integrated into quality gates to fail the build or trigger other actions if critical defects or policy violations exist. Analysis can be configured according to the risk profile of the software and the organization's coding standards to produce highly accurate results that align with the appropriate threshold for each project.

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. 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."

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:** Jet Propulsion Laboratory (JPL) Institutional Coding Standard for the C Programming Language (JPL, March 3, 2009).
- **For C++:** Joint Strike Fighter (JSF) 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.

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.

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. 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, a significant portion of the required documentation 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.

The codebase integrates previously developed code, such as

- 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 in Section 12.1.4, "Upgrading a Development Baseline."

Coverity helps developers submit cleaner code to artifact writers, minimizing time-consuming revision loops. 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 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. 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 (TQL), 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.

QUALIFYING COVERITY

Coverity is certified by TÜV SÜD Product Service GmbH as meeting the requirements for support tools according to IEC 61508-3. It is qualified for use in safety-related software development according to DO-178C, ISO 26262, IEC 61508, EN 50128, and EN 50657. And it is classified for use up to Level A in accordance with DO-178C.

The documentation pack for Coverity includes the necessary functional safety manual, which describes tool operation and failure modes, including the risk of misconfiguration and of false positives and false negatives.

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, 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.

DO-330, also known as Software Tool Qualification Considerations, is a standard published by RTCA that provides guidance for the qualification of software tools used in the development and verification of airborne systems. The goal of DO-330 is to replace the software tool qualification guidance of DO-178B/ED-12B and encourage its use outside the airborne domain. DO-330 defines tool qualification for Coverity, a commercial off-the-shelf (COTS) product, as a cooperative effort between Black Duck and the developer organization. Black Duck's contribution is tailored to the specific requirements of each qualification effort and is provided by Black Duck's services organization.

Coverity qualification at TQL-5

At TQL-5, Black Duck provides detailed documentation on Coverity's operational requirements. The system developer also documents how it meets the software life cycle objectives defined in DO-178C's Plan for Software Aspects of Certification and other detailed planning documents. Testing and verification of Coverity as installed is also a cooperative effort. Black Duck provides the required test cases and the test execution procedure; 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 at TQL-5. The level of effort for both Black Duck 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 Black Duck services group.

Formal methods as an alternative to static analysis

DO-178C discusses alternative ways to obtain certification, such as the use of "formal methods," which are 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. 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.

DO'S AND DON'TS

Do's

- Install, or have the vendor install, the candidate tool for a test run in your environment.
- 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, 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.
- Choose 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.
- Determine how effectively you're able to configure analysis based on your organization's, or the software's, risk profile. This may be critical to ensuring the appropriate level of results for your software.
- Investigate training, startup, and support options.
- Inquire about the vendor's capability to provide on-site training relevant to your SDLC.
- Verify that it can provide services to help you get started with the tool quickly and smoothly.
- Verify that its support hours correspond to your workday.
- Verify that it has 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 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. In many cases, critical defects can be very complex, span multiple files and libraries, and require in-depth analysis in order to be identified.
- 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 from 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 consider a static analysis tool to be sufficient because it's able to 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. Again, an important value of a static analysis tool is to find the kinds of defects that you aren't already looking for.

CONCLUSION

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 Coverity Static Analysis Software | Black Duck.

ABOUT BLACK DUCK

Black Duck® meets the board-level risks of modern software with True Scale Application Security, ensuring uncompromised trust in software for the regulated, AI-powered world. Only Black Duck solutions free organizations from tradeoffs between speed, accuracy, and compliance at scale while eliminating security, regulatory, and licensing risks. Whether in the cloud or on premises, Black Duck is the only choice for securing mission-critical software everywhere code happens. With Black Duck, security leaders can make smarter decisions and unleash business innovation with confidence. Learn more at www.blackduck.com.