PARASOFT

Achieving DO-178C Compliance

DO-178C is a document that outlines best practices for developing software for airborne electronic systems. DO-178C is not a prescriptive standard with respect to providing details on how to specifically write software, neither is written by the FAA—the body responsible for regulating software in airborne systems. The FAA, however, does consider the guidance described in DO-178C acceptable for applying software development practices that may prevent defects that potentially have life-threatening consequences. By adopting the DO-178C, software makers ensure that their products are in compliance with FAA regulation.

DO-178C can be viewed as a how-to manual for complying with Title 14 Code of Federal Regulations (14 CFR), which is the document that outlines the requirements for FAA approval. Parsing 14 CRF and its multiple references to other regulations and turning the information into an actionable plan for making software is a tremendous feat. Fortunately, this process was undertaken by the standards consortium RTCA, which publishes DO-178C.

DO-178C describes nearly the entire process of developing software: from planning and gathering requirements, to producing the code, to testing the final product. Organizations that need additional guidance related to their particular development method, programming style, etc. can also purchase RTCA's supplemental documents:

- **DO-278A** *Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems*
- **DO-330** *Software Tool Qualification Considerations*
- **D0-331** *Model-Based Development and Verification Supplement to DO-178C and DO-278A*
- **D0-332** *Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A*
- **DO-333** *Formal Methods Supplement to DO-178C and DO-278A*

In this paper, we'll focus only on DO-178C and concentrate on addressing specific processes, such as verification and validation activities, traceability, testing methods, and test environments. We'll look in greater detail at the following chapters:

- 5.3.2 Software Coding Process Activities
- 6.3.3 Reviews and Analyses of the Software Architecture
- 6.3.4 Reviews and Analyses of the Source Code
- 6.4.1 Test Environment
- 6.4.3 Requirements-Based Testing Methods
- 6.4.4.2 Structural Coverage Analysis
- 6.4.4.3 Structural Coverage Analysis Resolution
- 12.2 Tool Qualification

A Word about DO-178B

It is worth mentioning that DO-178C is the latest version of the standard. Most organizations are likely familiar with its predecessor, DO-178B, and will find that the recommendations in the newer version are not dramatically different. DO-178C does, though, improve on the language in B that may have left openings for interpretation. The intents behind some of the processes are explained in greater detail. For example, in chapter 6.1, which defines the purpose for the software verification process, DO-178C adds the following purpose with regard to the Executable Object Code:

> *The Executable Object Code is robust with respect to the software requirements such that it can respond correctly to abnormal inputs and conditions.*

This is in addition to the statement regarding the verification of the Executable Object Code defined in DO-178B:

> *The Executable Object Code satisfies the software requirements (that is, intended function), and provides confidence in the absence of unintended functionality.*

The additional requirement further defines the role of the Executable Object Code, which is a gap that may affect the safe functionality of the system. This is in contrast to the previous guideline, which may have been taken to reduce the scope of the verification process to system failure. There are many instances where the language in DO-178C seeks to define concepts, functions, and process in more detail, but an exhaustive list of such updates is outside the scope of this paper. A more definitive summary of the differences is provided in Appendix A of DO-178C.

5.3.2 Software Coding Process Activities

This section states that developers should write code in a manner that fulfills the low-level requirements and conforms to the architecture and standards outlined in the development plan. Four requirements for producing source code are listed that should be observed in order to consider the code properly developed:

a. *The Source Code should implement the low-level requirements and conform to the software architecture.*

Verifying that the low-level requirements (LLR) were implemented (and were implemented according to the planned architecture) requires a mechanism to provide visibility into the software development process. A development testing platform, such as Parasoft's Development Testing Platform, coupled with a suite of testing tools, such as Parasoft C/C++test, enables bi-directional traceability from code to requirement. This ensures that the source code is implemented correctly.

b. *The Source Code should conform to the Software Code Standards.*

Organizations have a number of options for standardizing the style of their code. Language and coding style do not matter to DO-178C, but declaring a standard in the development plan and scrupulously following it is key. Pattern-based static analysis looks for deviations from a set structure to enforce coding standards. Parasoft C/C++test, for example, ships with a library of common C and C++ coding standards, such MISRA C, MISRA C++, and JSF++, which automates this process.

c. *Inadequate or incorrect inputs detected during the software coding process should be provided to the software requirements process, software design process, and/or software planning process as feedback for clarification or correction.*

Problematic inputs are usually found in the QA phase and sent back to the development team. The issue with finding and reporting any issue in QA is that technical debt, or the amount of work required to maintain the application over time, increases as the software development lifecycle (SDLC) progresses. This is opposed to detecting defects, problematic inputs, or any other anomaly as the code is developed.

Writing test cases and conducting unit tests ensures that issues are detected as the code evolves. Unit testing is frequently considered a cumbersome process that developers may

occasionally skip. Automating test case generation and test execution with a solution such as Parasoft C/C++test, reduces the risk that errors will remain invisible until it's too late or too expensive to fix.

d. *Use of autocode generators should conform to the constraints defined in the planning process.*

In many cases, manual adjustments are necessary before auto-generated code can be implemented, which leaves a potential opening for the introduction of errors. Additionally, the output may not conform to the constraints defined in the planning process. In any case, auto-generated code must still be tested in the same manner as manually written code. Static analysis, unit testing, and certainly code review, should be used to ensure that auto-generated code follows this requirement.

6.3.3 Reviews and Analyses of the Software Architecture

This section states that you must review and analyze the software architecture in order to detect and report errors that may have been introduced. There are six objectives that the review and analysis activities you deploy must achieve to confirm that the software architecture is acceptable:

a. ***Compatibility with the high-level requirements:*** *The objective is to ensure that the software architecture does not conflict with the high-level requirements, especially functions that ensure system integrity, for example, partitioning schemes.*

b. ***Consistency:*** *The objective is to ensure that a correct relationship exists between the components of the software architecture. This relationship exists via data flow and control flow. If the interface is to a component of a lower software level, it should also be confirmed that the higher software level component has appropriate protection mechanisms in place to protect itself from potential erroneous inputs from the lower software level component.*

c. ***Compatibility with the target computer:*** *The objective is to ensure that no conflicts exist, especially initialization, asynchronous operation, synchronization, and interrupts, between the software architecture and the hardware/software features of the target computer*

d. ***Verifiability:*** *The objective is to ensure that the software architecture can be verified, for example, there are no unbounded recursive algorithms.*

e. ***Conformance to standards:*** *The objective is to ensure that the Software Design Standards were followed during the software design process and that deviations to the standards are justified, for example, deviations to complexity restrictions and design construct rules.*

f. ***Partitioning integrity:*** *The objective is to ensure that partitioning breaches are prevented.*

The software verification activities you've identified in your software plan, such as static analysis, unit testing, code review, coverage analysis, etc. must be reported in the Software Verification

Results to satisfy the above objectives (see DO-178C chapter 11.14 "Software Verification Results" for details).

The Software Verification Results should include procedures for all reviews, analysis and tests, the configuration item or software version under test, and all test pass/fail results, including coverage analysis and traceability documentation. Discrepancies should be recorded and tracked in your problem reporting system.

6.3.4 Reviews and Analyses of the Source Code

This section states that you must review and analyze the source code in order to detect and report errors. There are six objectives that the review and analysis activities you deploy must achieve to confirm that the software architecture is acceptable.

a. ***Compliance with the low-level requirements:*** *The objective is to ensure that the Source Code is accurate and complete with respect to the low-level requirements and that no Source Code implements an undocumented function.*

b. ***Compliance with software architecture:*** *The objective is to ensure that the Source Code matches the data flow and control flow designed in the software architecture.*

c. ***Verifiability:*** *The objective is to ensure that the Source Code does not contain statements and structures that cannot be verified and that the code does not have to be altered to test it.*

d. ***Conformance to standards:*** *The objective is to ensure that the Software Design Standards were followed during the development of the code, for example, complexity restrictions and code constraints. Complexity includes the degree of coupling between software components, the nesting levels for control structures, and the complexity of logical or numeric expressions. This analysis also ensures that deviations to the standards are justified.*

e. ***Traceability:*** *The objective is to ensure that the low-level requirements were developed into Source Code.*

f. ***Accuracy and consistency:*** *The objective is to determine the correctness and consistency of the Source Code, including stack usage, memory usage, fixed point arithmetic overflow and resolution, floating-point arithmetic, resource contention and limitations, worst-case execution timing, exception handling, use of uninitialized variables, cache management, unused variables, and data corruption due to task or interrupt conflicts. The compiler (including its options), the linker (including its options), and some hardware features may have an impact on the worst-case execution timing and this impact should be assessed.*

As with the reviews and analysis of the software architecture described in section 6.3.3, the software verification activities you've identified in your software plan, such as static analysis, unit testing, code review, coverage analysis, etc. must be reported in the Software Verification Results to satisfy the above objectives (see DO-178C chapter 11.14 "Software Verification Results" for details).

The Software Verification Results should include procedures for all reviews, analysis and tests, the configuration item or software version under test, and all test pass/fail results, including coverage analysis and traceability documentation. Discrepancies should be recorded and tracked in your problem reporting system.

6.4.1 Test Environment

This section states that the test environment should be as close as possible to the actual environment for which the software is being developed and that more than one environment may be required to achieve the best results. Emulated or simulated environments are acceptable. The standard also says that "selected tests should be performed in the integrated target computer environment, since some errors are only detected in this environment."

C/C++test can be used in host-based, simulator, and target-based code analysis and test flows, which ensures accurate test results. Additionally, Parasoft C/C++test, Jtest and dotTEST users can define test environment variables.

6.4.3 Requirements-Based Testing Methods

This section describes requirements-based testing methods and the errors they detect. Three methods are discussed, but only the Hardware/Software Integration Testing method is prescribed.

Requirements-Based Hardware/Software Integration Testing

The software must be able to fulfill high-level requirements after it is integrated with the target hardware to meet this guideline. DO-178C identifies numerous types of errors that are usually found during integration testing. The following errors are listed in this section of DO-178C:

- Failure to satisfy execution time requirements
- Inability of built-in test to detect failures
- Errors in hardware/software interfaces
- Stack overflow

C/C++test can execute tests on the target, as well as in the test environment so that hardware integration can be verified. Software engineers can adjust the unit testing scope to include functions as they are completed and run regression tests to ensure proper software integration. Additionally, the runtime error detection capabilities expose integration defects and other errors as the application is exercised.

Requirements-Based Software Integration Testing

The focus in this method is on implementing testing activities which ensure that software components integrate with each other, integrate with the architecture, and function according to the defined requirements. According to the standard, "this method may be performed by expanding the scope of the requirements through successive integration of code components with a corresponding expansion of the test cases."

Examples of software integration errors include incorrect initialization of variables and constants, parameter passing errors, data corruption, inadequate end-to-end numerical resolution, and incorrect sequencing of events and operations.

Users can adjust static analysis rules in Parasoft C/C++test to analyze as much or as little of the code as needed. Files, as well as lines of code, can be tested based on when they were added, who added them, by path, etc. Software engineers can adjust the unit testing scope to include functions as they are completed and run regression tests to ensure proper software integration. Additionally, the runtime error detection capabilities expose integration defects and other errors as the application is exercised

Requirements-Based Low-Level Testing

This method focuses on ensuring that the software complies with its low-level requirements. Examples of the errors this method detects include, failure of an algorithm to satisfy a software requirement, incorrect loop operations, incorrect logic decisions, etc.

Low-level testing usually means testing single functions with given inputs described in the requirements. Users can parameterize and automatically generate unit test cases to functions as they are completed. Tests can be saved and ran as regression suites to ensure proper software integration.

6.4.4.2 Structural Coverage Analysis

Requirements-based testing is not sufficient to analyze the complete code structure or component interfaces, so DO-178C prescribes structural coverage analysis and identifies the following activities, in particular:

a. *Analysis of the structural coverage information collected during requirements-based testing to confirm that the degree of structural coverage is appropriate to the software level.*

The multi-metric test coverage analyzer reports code coverage, including line, statement, block, decision/branch, path, and MC/DC, and condition coverage. Coverage can be tracked at all levels of testing—unit testing to integration to application—to deliver integrated reports.

6.4.4.3 Structural Coverage Analysis Resolution

If unexercised code is discovered during structural coverage analysis testing, it must be remediated. DO-178C lists four possible causes and provides guidance on resolution activities.

Shortcomings in Requirements-based Test Cases or Procedures

In these instances, the test cases are simply inadequate and should be altered to provide the missing coverage. The requirements-based method used should also be reviewed (see “6.4.3 Requirements-Based Testing Methods” above).

Parasoft C/C++test enables users to adjust coverage as necessary to ensure that all required code has been verified. Test cases are traceable to requirements via Parasoft’s Development Testing platform, so users can determine how to resolve shortcomings.

Inadequacies in Software Requirements

If requirements were written in such a way that results in unverifiable code, then software engineers may need to modify the software requirements. In Parasoft's Development Testing platform, users can revise requirements and maintain full bi-directional traceability.

Extraneous Code, Including Dead Code

According to DO-178C, this code should be removed and a verification process should be enacted to ensure its removal. Code may be allowed to remain, however, as long as it does not exist in the Executable Object Code as a result of compiling, for example. Users can run C/C++test static analysis to initially detect unexercised code, as well as verify its removal.

Deactivated Code

DO-178C identifies two categories of deactivated code, each with a different resolution process:

1. Deactivated code that is not intended to be executed requires a combination of analysis and testing that identifies the process of prevention, isolation or elimination.
2. Deactivated code that is only executed in certain configurations of the target computer environment requires additional test cases and procedures to verify the required coverage objective.

To verify either category of deactivated code, users can deploy Parasoft C/C++test, which includes flow analysis and runtime analysis tools.

12.2 Tool Qualification

An automated defect prevention tool, such as Parasoft C/C++test, must be qualified in order to ensure that it provides consistent and accurate software verification and validation results (see RTCA document DO-330 for more information). Moreover, the tool must be verified for each system, version, and configuration on which the tool is used.

Only the functions of the tool or tools used during the execution of DO-178C need to be qualified, provided that "tool protection" can be demonstrated. Protection refers to the ability of different functions within the tool to operate without adversely impacting other tool functions. Additionally, tools can only be qualified for the use specifically described in the Plan for Software Aspects of Certification (see DO-178C chapter 4.0 SOFTWARE PLANNING PROCESS for more information).

There are three criteria for determining if a tool needs to be qualified:

a. *Criteria 1: A tool whose output is part of the airborne software and thus could insert an error*
b. *Criteria 2: A tool that automates verification process(es) and thus could fail to detect an error, and whose output is used to justify the elimination of reduction of:*
    1. *Verification process(es) other than that automated by the tool, or*
    2. *Development process(es) that could have an impact on the airborne software.*
c. *Criteria 3: A tool that, within the scope of its intended use, could fail to detect an error.*

Moreover, DO-178C identifies different qualification levels, TQL-1 to TQL-5, that tools must be assigned. The TQL depends on the Software Level (see chapter 2.3.3 "Software Level Definition" for more information). Table 12-1 from DO-178C describes the relationships between the qualification criteria, TQL, and Software Level:

Software Level	Criteria		
	1	2	3
A	TQL-1	TQL-4	TQL-5
B	TQL-2	TQL-4	TQL-5
C	TQL-3	TQL-5	TQL-5
D	TQL-4	TQL-5	TQL-5

Users automating their defect prevention activities with Parasoft C/C++test (or any other tool) will have to qualify their tools under criterion b and/or c. The complete tool qualification process, including the objectives, activities and life cycle data, is provided in a separate RTCA document, DO-330 . . . for an additional cost.

At the most basic level, the software qualification means using the automation tool on a control codebase and verifying that the tool produces consistent, predictable outcomes. The process must be documented in the software plan. Parasoft offers a qualification kit for C/C++test. Contact Parasoft Professional Services for additional information regarding the test qualification kit.

Conclusion

Our goal in this paper was to identify and parse the particular sections of DO-178C that an automated defect prevention system can support. The standard provides guidance on software planning and software development lifecycle processes that require human intelligence. A Development Testing Platform can greatly assist with these activities, but fully exploring the benefits of a Development Testing Platform is beyond the scope of this paper. For details, read our Development Testing Platform paper.

About Parasoft

Parasoft researches and develops software solutions that help organizations deliver defect-free software efficiently. To combat the risk of software failure while accelerating the SDLC, Parasoft offers a Development Testing Platform and Continuous Testing Platform. Parasoft's enterprise and embedded development solutions are the industry's most comprehensive—including static analysis, unit testing, requirements traceability, coverage analysis, API testing, dev/test environment management, service virtualization and more. The majority of Fortune 500 companies rely on Parasoft in order to produce top-quality software consistently and efficiently as they pursue agile, lean, DevOps, compliance, and safety-critical development initiatives.

Contacting Parasoft

Headquarters

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016

Toll Free: (888) 305-0041
Tel: (626) 305-0041
Email: info@parasoft.com

Global Offices

Visit www.parasoft.com/contact for contacting Parasoft in EMEAI, APAC, and LATAM.

About the Authors

Adam Trujillo, Technical Writer, Parasoft
Michal Rozenau, Developer, Parasoft