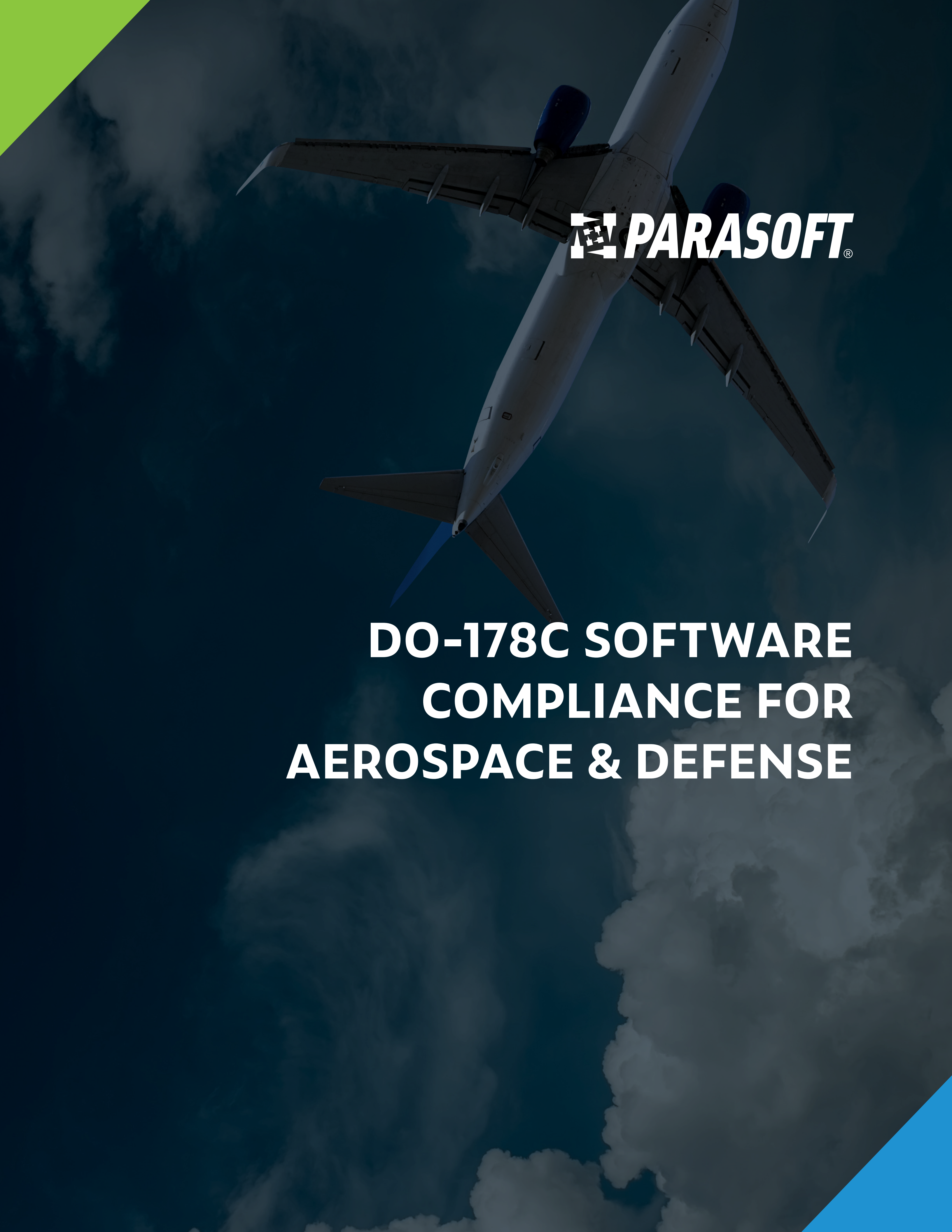


PARASOFT®

DO-178C SOFTWARE COMPLIANCE FOR AEROSPACE & DEFENSE

Contents

Overview

Aerospace Industry Outlook for Commercial & Defense

The aerospace industry is one of, if not the most technically complex and sophisticated, industries that exist. Much of it has to do with the diversity of the aircraft that are created for commercial as well as defense purposes. There's a large overlap in the latest trends in technology used by the aerospace industry, but there are also interesting areas that differ and are worth mentioning. However, one of the main contributors to changes and trends in the aerospace industry is cost.

The average passenger airliner costs between $82 and $350 million, and based on the type of military aircraft, it can cost between $82 and $2.1 billion. A Boeing 787-10 goes for $340 million and a Northrop Grumman B-2 Spirit Stealth Bomber will set you back $2.1 billion.

Commercial aircraft cost a significant amount due to factors like extensive research, development, production, and operations. Developing a new commercial aircraft involves substantial R&D efforts, including designing and testing new technologies, aerodynamics, materials, and safety features. This phase often spans several years and requires a substantial investment in skilled engineers, scientists, and facilities. This is the same for military aircraft, but in addition, they often pioneer new technologies and innovations that lead to higher R&D costs as well as the need for very specialized engineering talent.

Another factor in the cost of commercial aircraft is testing and certification. Extensive testing and certification processes are required to ensure that an aircraft meets safety, performance, and environmental standards. Similarly, military aircraft must undergo rigorous testing and certification processes to ensure their performance, safety, and compliance with military standards are achieved.

These and other influences, like supply chain complexities, material used (advanced composites and titanium), commercial use customizations (cabin layout, in-flight entertainment system, galley arrangements, and so on), or military customizations like weaponry, avionics, stealth, survivability, and other mission-specific equipment, can drive up costs.

Open Systems Architectures

One of the approaches being used by the aerospace industry to mitigate costs is the adoption of open architectures and interoperability. Open systems architecture is a system design approach that aims to produce systems, such as software and hardware, that are inherently interoperable and connectable without recourse to retrofit and redesign. The Future Airborne Capability Environment (FACE™) Consortium has established an open procurement environment that facilitates reuse to meet four core goals:

1. Improve affordability.
2. Increase speed.
3. Improve agility
4. Deliver excellence.

The FACE™ Consortium is a government and industry partnership dedicated to accomplishing the four core goals using open industry standards, advanced integration, and maintenance technologies. Military and commercial organizations can purchase FACE-certified products found in the FACE registry.

Artificial Intelligence & Machine Learning

The use of artificial intelligence (AI) and machine learning (ML) comes up at aerospace events, and one thought is to replace the commercial airline copilot with an AI copilot. There are some hefty safety hurdles to overcome before this scenario can be realized. Nonetheless, analytical AI can be applied in aerospace to predict when a part is going to fail through anomaly detection or by tracking, scheduling, and managing maintenance based on historical data and predictive analytics. However, this is completely the opposite for defense.

The U.S. is developing AI capabilities for a broad range of military functions that will have a significant impact on the defense sector. AI technologies are rapidly evolving. Defense primes are advancing their AI capabilities organically and through acquisitions.

AI is being applied in operations like intelligence, surveillance, reconnaissance (ISR), logistics, cyber, command and control, and drone swarms. Perhaps the most publicized and controversial AI application in defense concerns autonomous vehicles and weapon systems. AI technology will make military operations more efficient, accurate, and powerful while also offering long-term cost-cutting potential.

Urban Air Mobility

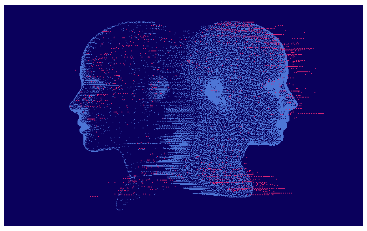
In the commercial space, one of the major trends is the push towards more sustainable and environmentally friendly aviation. This refers to the development of electric and hybrid electric propulsion systems.

The FAA has put out the Urban Air Mobility (UAM) Concept of Operations in support of developing air transportation for a wide range of passenger, cargo, and other operations within and between urban and rural environments using new and innovative aircraft. Vehicles such as electric vertical takeoff and landing (eVTOL) types of aircraft are currently under development. Nevertheless, the U.S. military is also embracing eVTOL for military missions.

Development & Design

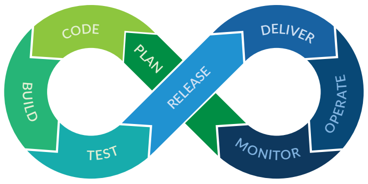
Advancements in software solutions and practices are also making improvements in productivity, quality, time to market, and costs. Other technologies, like cybersecurity, have become of paramount concern. Here are a few that are having a powerful impact on development and need mentioning.

Digital Twin



The use of a virtual representation or virtual model of a physical system that mimics the functionalities of the actual hardware and software is referred to as a "digital twin." Digital twins for an aircraft, jet engine, or even a semiconductor subsystem offer the unique capability of a shift-left approach to enable earlier design, analysis, and verification.

Agile Methodologies



Agile methodologies such as DevOps and DevSecOps are being adopted to improve the efficiency of software development. These approaches emphasized iterative development, collaboration, and continuous integration and delivery (CI/CD), enabling faster and more reliable software delivery.

Adopting these agile development methodologies does not conflict with DO-178C recommended software development processes. DO-178C is a descriptive standard that informs and recommends what should be done to ensure safety. The "how" is left up to the organization to decide on evolving best practices and solutions.

Model-Based Systems Engineering

Aerospace companies have been increasingly adopting model-based engineering (MBSE), which involves creating digital models that can represent the entire system, including hardware, software, and interactions. MBSE helps improve communication among multidisciplinary teams and allows for better system understanding and integration.

Cybersecurity

With the increasing connectivity of aerospace systems and the reliance on software for critical functions, cybersecurity is a key concern. The military and aerospace companies are focusing on implementing robust cybersecurity measures to protect against cyber threats and ensure the safety and security of aviation systems. RTCA DO-326A and DO-355A are the de facto cybersecurity standards.

Mil/Def Aerospace

Specific to the aerospace and aviation sectors within the military/defense (Mil/Def) industry, they are responsible for designing, developing, and manufacturing a wide range of military aircraft, helicopters, and unmanned aerial vehicles (UAVs). These vehicles serve various purposes, including reconnaissance, surveillance, combat, and transport.

Military aircraft are equipped with advanced avionics systems, high-performance engines, and cutting-edge weapon systems to ensure air superiority and the effective deployment of military operations. This sector is also involved in space exploration and satellite technologies. Military satellites are critical for communication, intelligence gathering, and navigation. They facilitate secure and real-time communication between ground troops, aircraft, and command centers.

Additionally, military space technology contributes to early warning systems, weather monitoring, and global positioning capabilities. The military is not required to adapt commercial aviation safety certification guidelines, but they do so because such guidelines enable a more robust, safe, and secure aircraft for the warfighter.

The Role of Standards & Regulations

DO-178C, which is also published in Europe as EUROCAE ED-12C, is the standard for "Software Considerations in Airborne Systems and Equipment Certification." It's a core standard for all avionics or airborne systems and a document by which certification authorities such as the Federal Aviation Administration (FAA), European Union Safety Agency (EASA), and Transport Canada approve and certify all commercial software-based aerospace systems.

Avionics is an assembly of electronics subsystems integrated onboard freighter aircraft, military aircraft, business jets, and other private-owned, chartered, and unscheduled aircraft. These systems include engine controls, flight control systems, navigation, communications, flight recorders, lighting systems, fuel systems, electro-optic (EO/IR) systems, weather radar, and performance monitoring systems.

Without certification, commercial airborne software systems cannot be deployed. The military is not required to adapt commercial aviation safety certification guidelines, but they do so because such guidelines enable a more robust, safe, and secure aircraft for the warfighter.

As safety and security concerns grow due to advances in technology and their application in avionic systems, one standard cannot address all solutions and best practices. Therefore, there are supplemental RTCA guidance documents that contain clarifications, frequently asked questions, discussion papers, and rationale to DO-178C. Here are just a few:

- DO-278A, Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems
- DO-248C, Supporting Information for DO-178C and DO-278A
- DO-333, Formal Methods Supplement to DO-178C and DO-278A
- DO-326A, Airworthiness Security Process Specification
- DO-355A, Information Security Guidance for Continuing Airworthiness
- DO-330, Software Tool Qualification Considerations
- DO-331, Model Based Development and Verification
- DO-332, Object Oriented Technology and Related Techniques
- DO-254, Design Assurance Guidance for Airborne Electronic Hardware

Though not part of the RTCA library, an important standard to include is SAE AS9100D: Quality Management Systems - Requirements for Aviation, Space, and Defense Organizations. It's the international quality standard used by the aerospace industry for applying best practices in product safety, security, and performance that help run your organization efficiently and effectively.

Organizational best practices and processes aid teams in getting organized, reduce costs, mitigate risks, boost productivity, and drive continuous improvement. Organizations certified to this standard demonstrate a commitment to excellence and the delivery of quality. It provides your customers with a way of determining whether you are a viable and attractive alternative to other suppliers.

In addition, to stay up-to-date on FAA regulations, the FAA Dynamic Regulatory System (DRS) is a knowledge center that includes all regulatory guidance material and is continuously updated.

What Is RTCA DO-178C?

The Radio Technical Committee for Aeronautics (RTCA) DO-178C is a functional safety standard that provides guidance and considerations for the production of software for airborne systems and equipment. The aim is to ensure that the system performs its intended function with a level of confidence in safety that complies with airworthiness requirements. If an aircraft is to fly over commercial U.S. airspace, compliance with the standard is required.

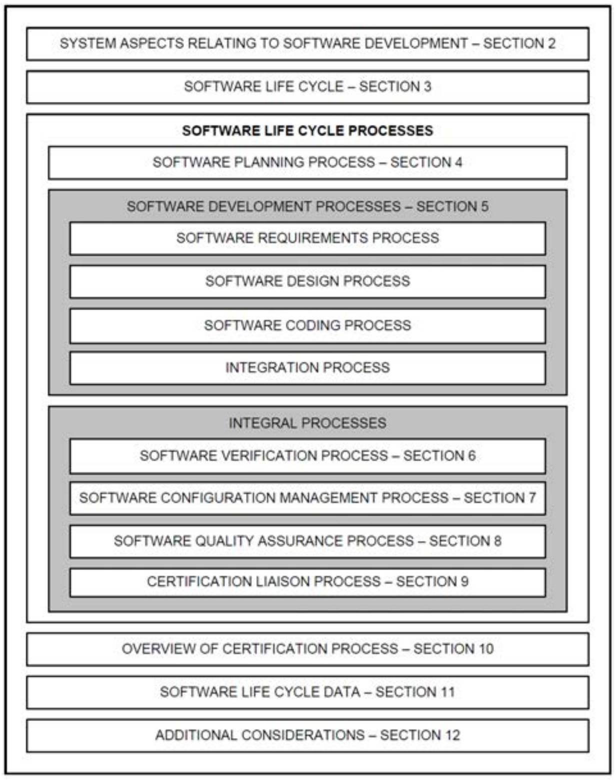

SYSTEM ASPECTS RELATING TO SOFTWARE DEVELOPMENT – SECTION 2
SOFTWARE LIFE CYCLE – SECTION 3
SOFTWARE LIFE CYCLE PROCESSES
SOFTWARE PLANNING PROCESS – SECTION 4
SOFTWARE DEVELOPMENT PROCESSES – SECTION 5
SOFTWARE REQUIREMENTS PROCESS
SOFTWARE DESIGN PROCESS
SOFTWARE CODING PROCESS
INTEGRATION PROCESS
INTEGRAL PROCESSES
SOFTWARE VERIFICATION PROCESS – SECTION 6
SOFTWARE CONFIGURATION MANAGEMENT PROCESS – SECTION 7
SOFTWARE QUALITY ASSURANCE PROCESS – SECTION 8
CERTIFICATION LIAISON PROCESS – SECTION 9
OVERVIEW OF CERTIFICATION PROCESS – SECTION 10
SOFTWARE LIFE CYCLE DATA – SECTION 11
ADDITIONAL CONSIDERATIONS – SECTION 12

Figure 2-1: The sections that make up the DO-178C standard

DO-178C provides the following guidance:

- Objectives for software life cycle processes
- Activities that provide a means for satisfying those objectives
- Descriptions of the evidence in the form of software life cycle data that indicate that the objectives have been satisfied
- Variations in the objectives, independence, software life cycle data, and control categories by software level
- Additional considerations (for example, previously developed software) that are applicable to certain applications
- Definition of terms provided in the glossary

DO-178C covers the full engineering life cycle. From planning, development, verification, quality assurance, liaison, and certification. It is subdivided into 12 sections. **Section 1, not shown expresses the purpose, scope, and how to use the document.**

RTCA was founded back in 1935. They are an independent standards development organization and serve as the basis for government certification of equipment used by the tens of thousands of aircraft flying daily through the world's airspace.

RTCA is a private, not-for-profit corporation, which works closely with the Federal Aviation Administration (FAA) and industry experts from the U.S. and around the world, such as the European Organization for Civil Aviation Equipment (EUROCAE) working group to help develop this comprehensive, contemporary aviation standard. The EUROCAE is a non-profit organization with the objective of developing standards for European civil aviation.

The original DO-178 standard was released back in 1982. However, it was not considered useful. As a result, the DO-178A revision followed, published in 1985. This revision focused more on modern software engineering principles and verification practices. It introduced a correlation between critical failure conditions with level numbers 1, 2, and 3. Level 1, which you may know better as Development Assurance Level (DAL) was the strictest.

In December 1992, revision DO-178B was released, which shifted from a “how to” type of document to a “what to do” type of document. A big focus was put on objectives that your software process needs to satisfy in order to reach compliance and ultimately certification.

Another noticeable change was to the number of possible critical failure conditions defined in DAL. They grew to five software levels and changed from numbers to letters A through E. Level A was the most stringent and Level E meant no safety requirement. Also, testing your requirements was strongly emphasized. It advised not to look at the code to create test cases, but to look at your requirements. It was backed by structural code coverage to ensure that you have covered everything.

DO-178B also incorporated bidirectional traceability between systems, high- and low-level requirements, including test cases, and down to the code to show that all the requirements have been implemented. The idea of having tools qualified for use was introduced.

Today, we're at revision C. Released in January 2012, DO-178C removed imprecise wording found in DO-178B for clarification. It also became a joint effort between RTCA and EUROCAE. But the major difference between DO-178B and DO-178C is the adoption of a modular approach to supplemental guidance documents. You now have supplemental standards, including the following.

» DO-330 addresses software tool qualification.

» DO-331 addresses model-based development.

» DO-332 addresses object-oriented software.

» DO-333 addresses formal methods to complement your testing.

This ebook provides a condensed overview of each of the DO-178C sections, highlighting the key takeaways.

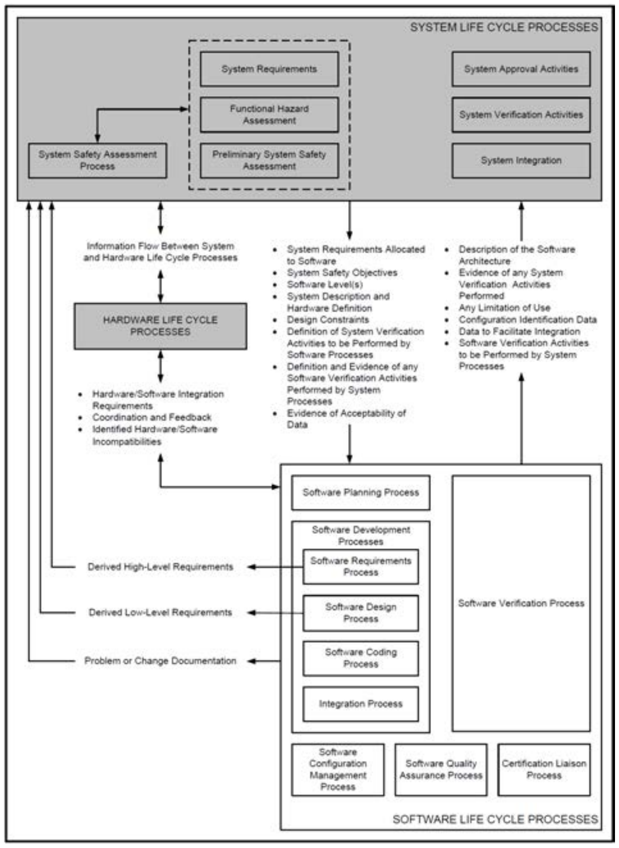
System Aspects Relating to Software Development, Section 2

Section 2 discusses the system life cycle processes, the artifacts produced, how they flow down into the software life cycle and the information flow between these processes. A big part of this is requirements analysis, where the software system requirements are initially developed from the system operational requirements or customer requirements, and how these artifacts flow into the software life cycle.

In the software life cycle, requirements decomposition continues, software verification takes place, and ultimately certification.

Though DO-178C captures the flow between system and software life cycles in the diagram above, the topic is well defined in the SAE ARP4754A standard, Guidelines for Development of Civil Aircraft and Systems.

Figure 2-2: Information flow between system and software life cycle processesThe Guideline Enforcement Plan demonstrates how each MISRA guideline is verified.


SYSTEM LIFE CYCLE PROCESSES
System Requirements
Functional Hazard Assessment
Preliminary System Safety Assessment
System Safety Assessment Process
System Approval Activities
System Verification Activities
System Integration
Information Flow Between System and Hardware Life Cycle Processes
HARDWARE LIFE CYCLE PROCESSES
• Hardware/Software Integration Requirements
• Coordination and Feedback
• Identified Hardware/Software Incompatibilities
• System Requirements Allocated to Software
• System Safety Objectives
• Software Level(s)
• System Description and Hardware Definition
• Design Constraints
• Definition of System Verification Activities to be Performed by Software Processes
• Definition and Evidence of any Software Verification Activities Performed by System Processes
• Evidence of Acceptability of Data
• Description of the Software Architecture
• Evidence of any System Verification Activities Performed
• Any Limitation of Use
• Configuration Identification Data
• Data to Facilitate Integration
• Software Verification Activities to be Performed by System Processes
Derived High-Level Requirements
Derived Low-Level Requirements
Problem or Change Documentation
Software Planning Process
Software Development Processes
Software Requirements Process
Software Design Process
Software Coding Process
Integration Process
Software Verification Process
Software Configuration Management Process
Software Quality Assurance Process
Certification Liaison Process
SOFTWARE LIFE CYCLE PROCESSES

Section 2 discusses the following topics:

» System requirements allocation to software

» Information flow between the system and software life cycle processes and between the software and hardware life cycle processes

» System safety assessment process, failure conditions, software level definitions, and software level determination

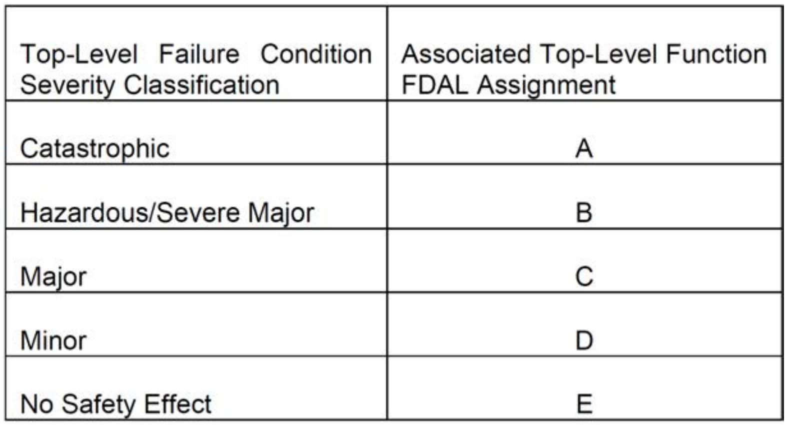
» Architectural considerations

» Software considerations in system life cycle processes

» System considerations in software life cycle processes

One other important part of section 2 is determining the software level classification or DAL. Catastrophic results equate to failure of flight control software where an aircraft would go down and many lives would be lost. This would be classified as software level A.

Hazardous is a step down, so serious or fatal injury to a relatively small number of the occupants other than the flight crew would be software level B. The classification continues to go down to software level E where there's no safety concern if failure were to occur.



Top-Level Failure Condition Severity Classification	Associated Top-Level Function FDAL Assignment
Catastrophic	A
Hazardous/Severe Major	B
Major	C
Minor	D
No Safety Effect	E

Table 2-1: DO-178C Development Assurance Levels (DAL)

Another perspective or side of this classification is quality assurance. With each increased level from level E to level A, there's an increased number of objectives that need to be met. For example, there's an increase in traceability between artifacts produced during product development. Also, there's an increase in software testing. The software may need to satisfy assembly or object code coverage instead of just statement, branch, and MC/DC coverage.

To share a best practice, if your software is classified at level B or lower, you may want to try to achieve some or all of the next higher software level objectives. The additional effort between some of the development assurance levels may not be too substantial and the benefits could very well pay off if customer requirements become more stringent.

Below is the well-known V-model. The right side captures the system and software design phases while the left side captures the verification phases. Standard ARP4754 is your go-to document on the development of aircraft systems considering the overall aircraft operating environment and functions. This includes validation of requirements and verification of the design implementation for certification and product assurance.

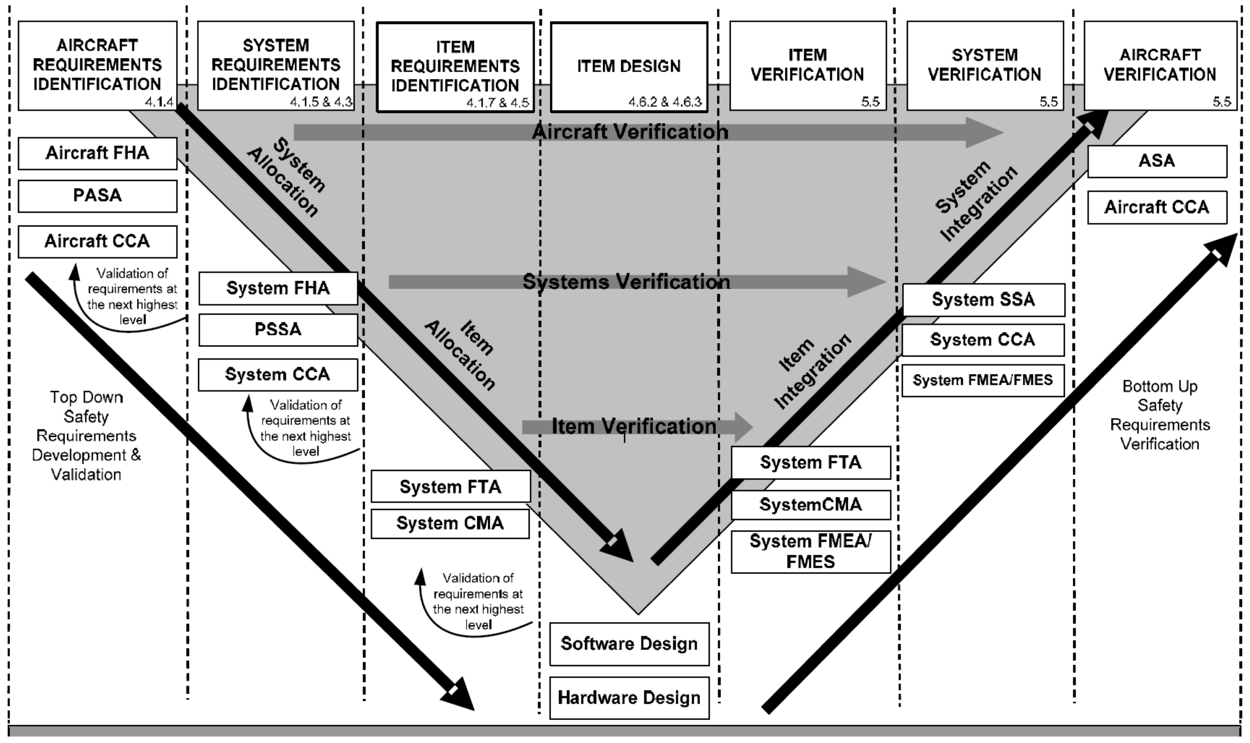


Figure 2-3: ARP4754A V-model development process

Software Life Cycle, Section 3

Section 3 discusses the aspects of the software life cycle process. The well-known sequence through the SDLC is requirements management, design, coding, and integration. DO-178C does not recommend a development process to use. It's left up to organizations to make that decision based on their own experience and factors like current technology, such as Agile, DevSecOps, CI/CD, or customer requirements. Whatever process you choose, the standard's objectives that must be met are not obstructed by the process.

DO-178C software life cycle processes include the following:

» **Software planning process.** Defines and coordinates the activities of the software development and integral processes for a project.

» **Software development processes.** Produce the software product. This process is comprised of processes for requirements, design, coding, and integration.

» **Integral software processes.** Ensure the correctness, control, and confidence in the software life cycle processes and their outputs. These include verification, configuration management, quality assurance, and certification liaison.

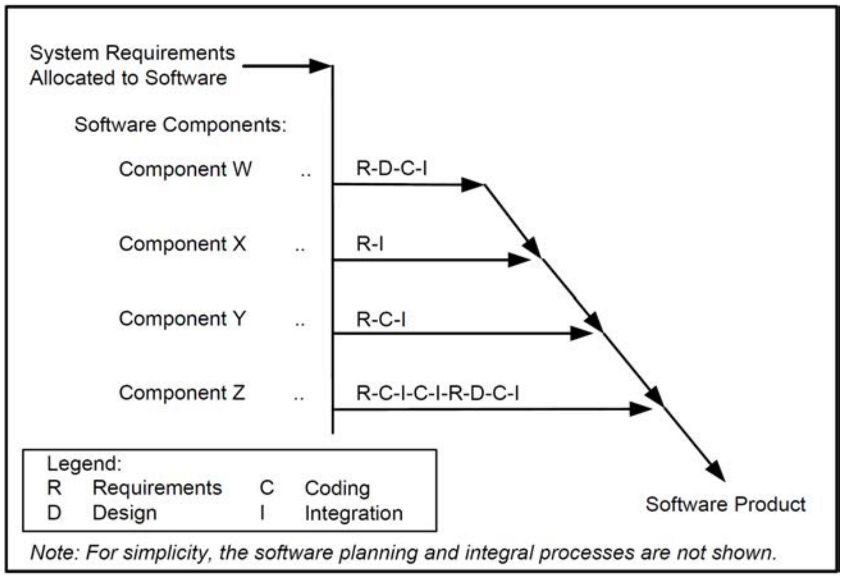

System Requirements Allocated to Software
Software Components:
Component W .. R-D-C-I
Component X .. R-I
Component Y .. R-C-I
Component Z .. R-C-I-C-I-R-D-C-I
Legend:
R Requirements C Coding
D Design I Integration
Software Product
Note: For simplicity, the software planning and integral processes are not shown.

Figure 2-4: DO-178C example of a software project using development sequences.

Software Planning Process, Section 4

Section 4 discusses the objectives and activities of the software planning process. The objectives are clearly defined and captured in Table A-1 of the standard. There are seven objectives that must be satisfied based on the software level (A-D). These objectives include defining the following:

- Software life cycle process
- Inter-relationships between processes
- Methods and tools to use
- Development standards to use for ensuring safety
- Verification that the software satisfies development requirements
- Verification that the organizations that will perform those activities

There are also many considerations to the software planning process, like the intent to use previously developed software or commercial off the shelf software (COTS), tool qualification, and many more described in section 12.

Table A-1 of the standard captures the objectives, the software levels that apply, and the expected output from these activities, which are a set of documents with reporting information about the organization, industry standard, software development, tools, verification results and certification.

- Plan for Software Aspects of Certification (PSAC)
- Software Development Plan (SDP)
- Software Verification Plan (SVP)
- Software Configuration Management Plan (SCM Plan)
- Software Quality Assurance Plan (SQM Plan)
- Software Requirements Standards
- Software Design Standards
- Software Code Standards
- Software Verification Results

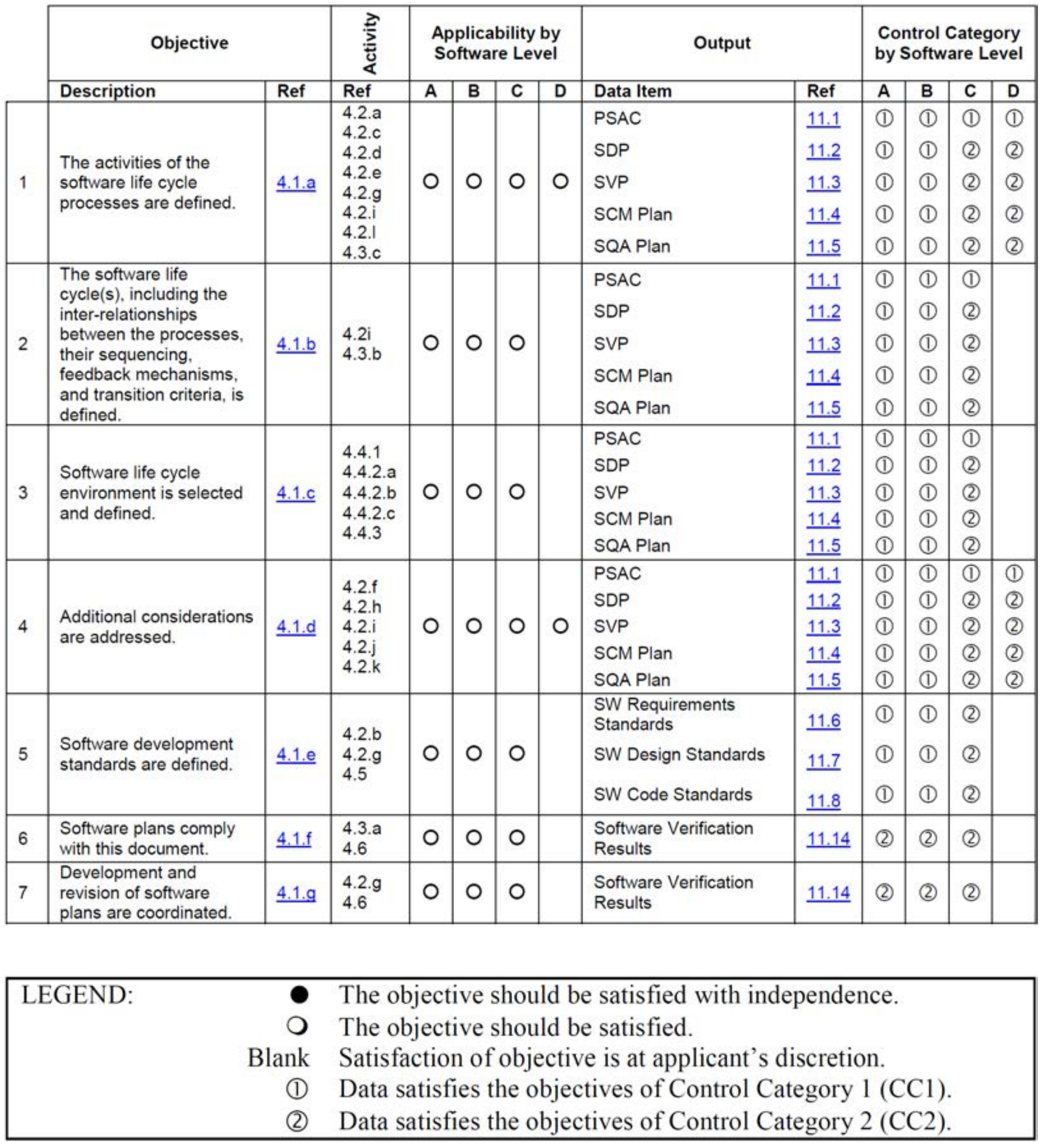


Table 2-2: Table A-1 Software planning process

	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	The activities of the software life cycle processes are defined.	4.1.a	4.2.a 4.2.c 4.2.d 4.2.e 4.2.g 4.2.i 4.2.l 4.3.c	○	○	○	○	PSAC	11.1	①	①	①	①
								SDP	11.2	①	①	②	②
								SVP	11.3	①	①	②	②
								SCM Plan	11.4	①	①	②	②
								SQA Plan	11.5	①	①	②	②
2	The software life cycle(s), including the inter-relationships between the processes, their sequencing, feedback mechanisms, and transition criteria, is defined.	4.1.b	4.2i 4.3.b	○	○	○		PSAC	11.1	①	①	①	
								SDP	11.2	①	①	②	
								SVP	11.3	①	①	②	
								SCM Plan	11.4	①	①	②	
								SQA Plan	11.5	①	①	②	
3	Software life cycle environment is selected and defined.	4.1.c	4.4.1 4.4.2.a 4.4.2.b 4.4.2.c 4.4.3	○	○	○		PSAC	11.1	①	①	①	
								SDP	11.2	①	①	②	
								SVP	11.3	①	①	②	
								SCM Plan	11.4	①	①	②	
								SQA Plan	11.5	①	①	②	
4	Additional considerations are addressed.	4.1.d	4.2.f 4.2.h 4.2.i 4.2.j 4.2.k	○	○	○	○	PSAC	11.1	①	①	①	①
								SDP	11.2	①	①	②	②
								SVP	11.3	①	①	②	②
								SCM Plan	11.4	①	①	②	②
								SQA Plan	11.5	①	①	②	②
5	Software development standards are defined.	4.1.e	4.2.b 4.2.g 4.5	○	○	○		SW Requirements Standards	11.6	①	①	②	
								SW Design Standards	11.7	①	①	②	
								SW Code Standards	11.8	①	①	②	
6	Software plans comply with this document.	4.1.f	4.3.a 4.6	○	○	○		Software Verification Results	11.14	②	②	②	
7	Development and revision of software plans are coordinated.	4.1.g	4.2.g 4.6	○	○	○		Software Verification Results	11.14	②	②	②	

LEGEND:		
	●	The objective should be satisfied with independence.
	○	The objective should be satisfied.
	Blank	Satisfaction of objective is at applicant's discretion.
	①	Data satisfies the objectives of Control Category 1 (CC1).
	②	Data satisfies the objectives of Control Category 2 (CC2).

Software Development Process, Section 5

The software development process is applied as defined by the software planning process and the software development plan. Whether teams or organizations choose a software development methodology like DevOps, Spiral, Waterfall, or another, the following four listed processes must be performed.

» Software requirements process

» Software design process

» Software coding process

» Integration process

The software requirements process begins by gathering all requirements from the stakeholder, regulatory bodies, standards, and more. These requirements are organized into domains such as hardware, software, mechanical, chemical, electrical, and so on, and then become your system-level requirements.

High-level requirements are derived from top-level system requirements. They decompose a system requirement into various high-level functional and nonfunctional requirements. This phase of the requirements decomposition helps in the architectural design of the system under development.

High-level requirements clarify and help define expected behavior as well as safety tolerances, security expectations, reliability, performance, portability, availability, scalability, and more. Each high-level requirement links up to the system requirement that it satisfies. In addition, high-level test cases are created and linked to each high-level requirement for the purpose of its verification and validation. This is part of the software design process, as each high-level requirement is further decomposed into low-level requirements.

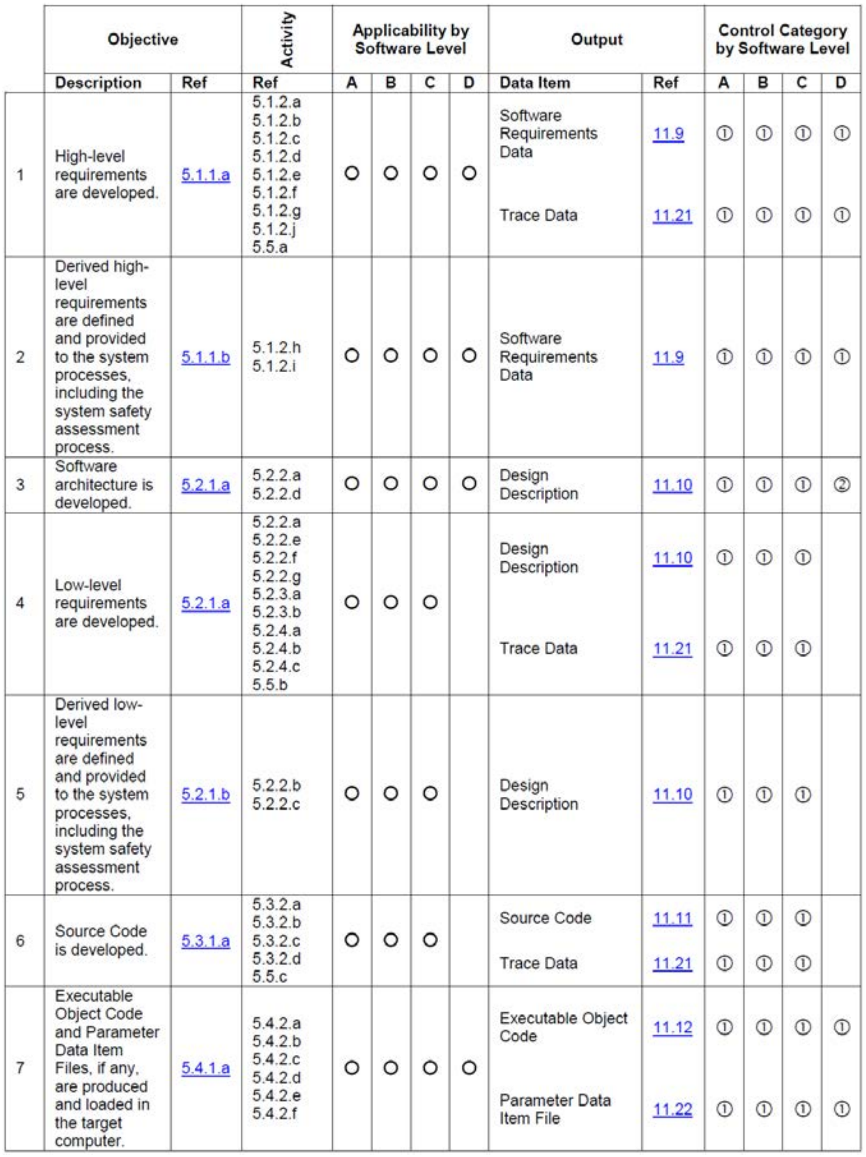
Low-level requirements are software requirements derived from high-level requirements. They further decompose and refine the specifications of the software's behavior and quality of service. They drill down to another level of abstraction and map it to individual software units. The coding process begins as code units are written to facilitate the software's detailed design and implementation. The inputs to the coding process are the low-level requirements and software architecture from the software design process, the software development plan, and the software code standards.

After the coding process is complete, the integration process consists of the following:

- Compiling
- Linking
- Loading software onto system or target hardware
- Executing

Coding defects need to be identified and fixed. Inadequate or incorrect inputs detected during the integration process should be provided to the following software processes as feedback for clarification or correction:

- Requirements
- Design
- Coding
- Planning



	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	High-level requirements are developed.	5.1.1.a	5.1.2.a 5.1.2.b 5.1.2.c 5.1.2.d 5.1.2.e 5.1.2.f 5.1.2.g 5.1.2.j 5.5.a	○	○	○	○	Software Requirements Data	11.9	①	①	①	①
								Trace Data	11.21	①	①	①	①
2	Derived high-level requirements are defined and provided to the system processes, including the system safety assessment process.	5.1.1.b	5.1.2.h 5.1.2.i	○	○	○	○	Software Requirements Data	11.9	①	①	①	①
3	Software architecture is developed.	5.2.1.a	5.2.2.a 5.2.2.d	○	○	○	○	Design Description	11.10	①	①	①	②
4	Low-level requirements are developed.	5.2.1.a	5.2.2.a 5.2.2.e 5.2.2.f 5.2.2.g 5.2.3.a 5.2.3.b 5.2.4.a 5.2.4.b 5.2.4.c 5.5.b	○	○	○		Design Description	11.10	①	①	①	
								Trace Data	11.21	①	①	①	
5	Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process.	5.2.1.b	5.2.2.b 5.2.2.c	○	○	○		Design Description	11.10	①	①	①	
6	Source Code is developed.	5.3.1.a	5.3.2.a 5.3.2.b 5.3.2.c 5.3.2.d 5.5.c	○	○	○		Source Code	11.11	①	①	①	
								Trace Data	11.21	①	①	①	
7	Executable Object Code and Parameter Data Item Files, if any, are produced and loaded in the target computer.	5.4.1.a	5.4.2.a 5.4.2.b 5.4.2.c 5.4.2.d 5.4.2.e 5.4.2.f	○	○	○	○	Executable Object Code	11.12	①	①	①	①
								Parameter Data Item File	11.22	①	①	①	①

Table 2-3: DO-178C Table A-2 Software development process

Bidirectional traceability that is established from each low-level requirement up to its high-level requirement and down to the low-level tests or unit test cases that verify and validate it helps in this endeavor.

Traceability is crucial to DO-178C. The depth of traceability varies based on the software level. Looking at the traceability that's required for DO-178C level D, organizations need not care about how the software has been developed, and as such, there's no need to have any traceability down to low-level requirements, the source code, or software architecture. Teams just need to trace from the system software requirements to the high-level requirements and then to the test cases, test procedures, and test results.

For levels B and C, how the source code has been developed becomes important. Teams need to expand traceability by adding bidirectional links from the high-level requirements to the low-level requirements and to the source code.

For level A projects, the requirements are to expand the traceability not just down to the source code, but to the assembly/object code. This is because compilers are known to expand and translate higher level languages to assembly code that does not map back to the originating code.

Parasoft has an assembly code coverage solution called ASMTools that automates code coverage at the assembly language level. Automating this effort alleviates much labor if code coverage at the assembly level is required.

For requirements traceability, Parasoft automates linking between requirements, test cases, and down to the source file, if required. Integrations with ALM tools like Jama, Codebeamer, and Polarion exist to help achieve this bidirectional traceability and building a traceability matrix for verification requirements.

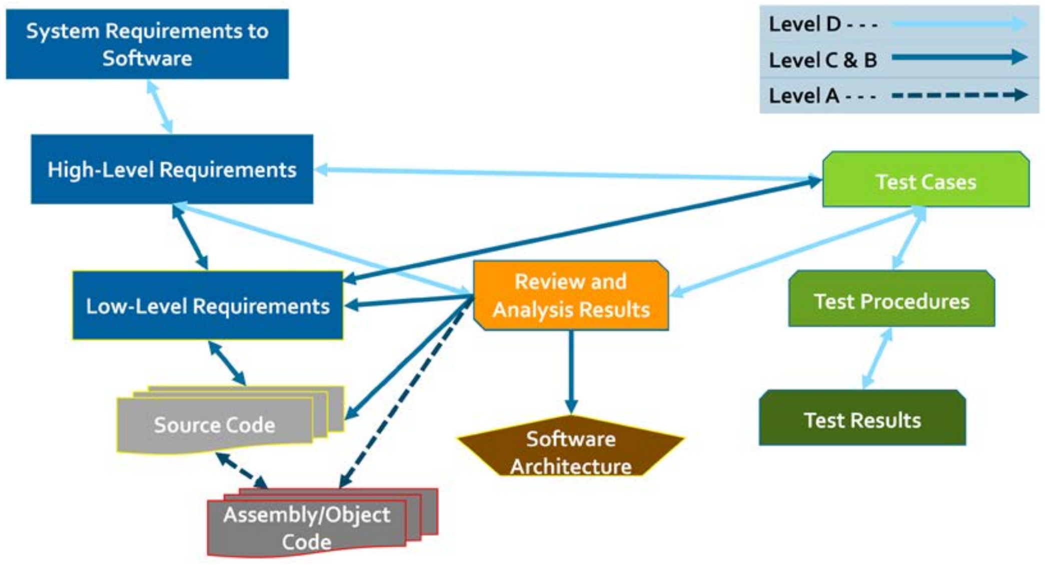


Figure 2-5: Requirements traceability through DO-178C software levels (D-A)

Software Verification Process, Section 6

The purpose of the software verification process is to detect, report, and remove the errors that may have been introduced during the software development process. The standard uses the term “verification” instead of “test” because testing alone cannot show the absence of errors. Verification is a combination of reviews, analysis, tests cases, and test procedures.

Tests provide internal consistency and completeness of the requirements, while test executions provide a demonstration of compliance with requirements.

DO-178C software verification process enables the following:

» The system requirements allocated to software shall be decomposed into high-level requirements that satisfy system requirements.

» High-level requirements shall be developed into software architecture and low-level requirements that satisfy high-level requirements.

» If one or more levels of software requirements are decomposed into high-level and low-level requirements, each successively lower level satisfies its higher-level requirements. If code is generated directly from high-level requirements, this does not apply.

» The software architecture and low-level requirements shall be developed into source code that satisfies low-level requirements and software architecture.

» The executable object code must satisfy software requirements and provide confidence in fulfilling its intended functionality.

» The executable object code shall be robust and respond correctly to abnormal inputs and conditions.

» The means used to perform the verification to be technically correct and complete for every determined software level.

Software testing demonstrates or “validates” that the software satisfies its requirements and reveals with a high degree of confidence that errors that could lead to unacceptable failure conditions, as determined by the system safety and security assessment process, have been removed. The following diagram shows software testing activities with subsections.

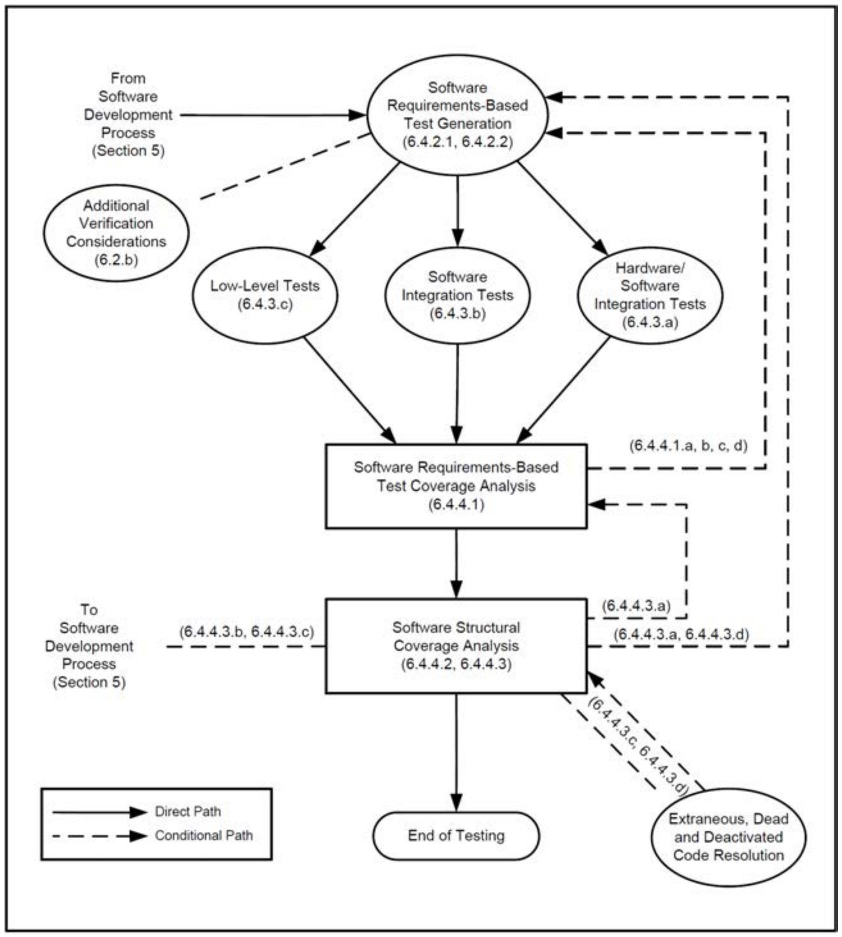

From Software Development Process (Section 5)
Software Requirements-Based Test Generation (6.4.2.1, 6.4.2.2)
Additional Verification Considerations (6.2.b)
Low-Level Tests (6.4.3.c)
Software Integration Tests (6.4.3.b)
Hardware/ Software Integration Tests (6.4.3.a)
(6.4.4.1.a, b, c, d)
Software Requirements-Based Test Coverage Analysis (6.4.4.1)
(6.4.4.3.a)
(6.4.4.3.a, 6.4.4.3.d)
To Software Development Process (Section 5)
(6.4.4.3.b, 6.4.4.3.c)
Software Structural Coverage Analysis (6.4.4.2, 6.4.4.3)
(6.4.4.3.c, 6.4.4.3.d)
Direct Path
Conditional Path
End of Testing
Extraneous, Dead and Deactivated Code Resolution

Figure 2-6: Software testing activities

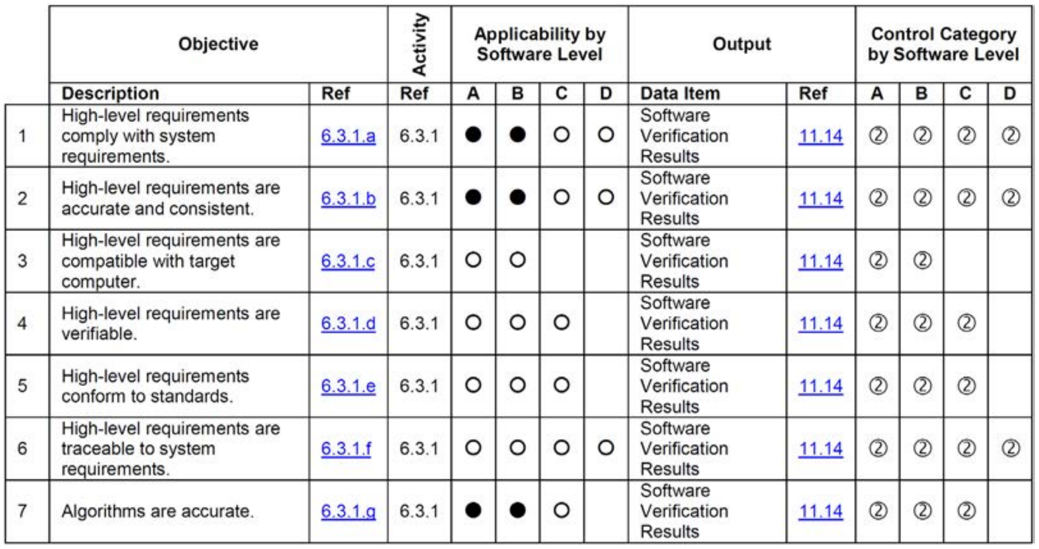
To further detail each testing activity, the standard provides a set of tables with well-defined objectives and outputs or artifacts needed to demonstrate compliance. These objectives are achieved by way of software testing and may include the following:

- Performing static analysis
- Unit testing
- Integration testing
- System testing
- Structural code coverage (statement, branch, MC/DC, assembly)
- On-target hardware
- Data and control coupling

Integrating hardware and software is crucial to ensuring safety, security, and reliability.

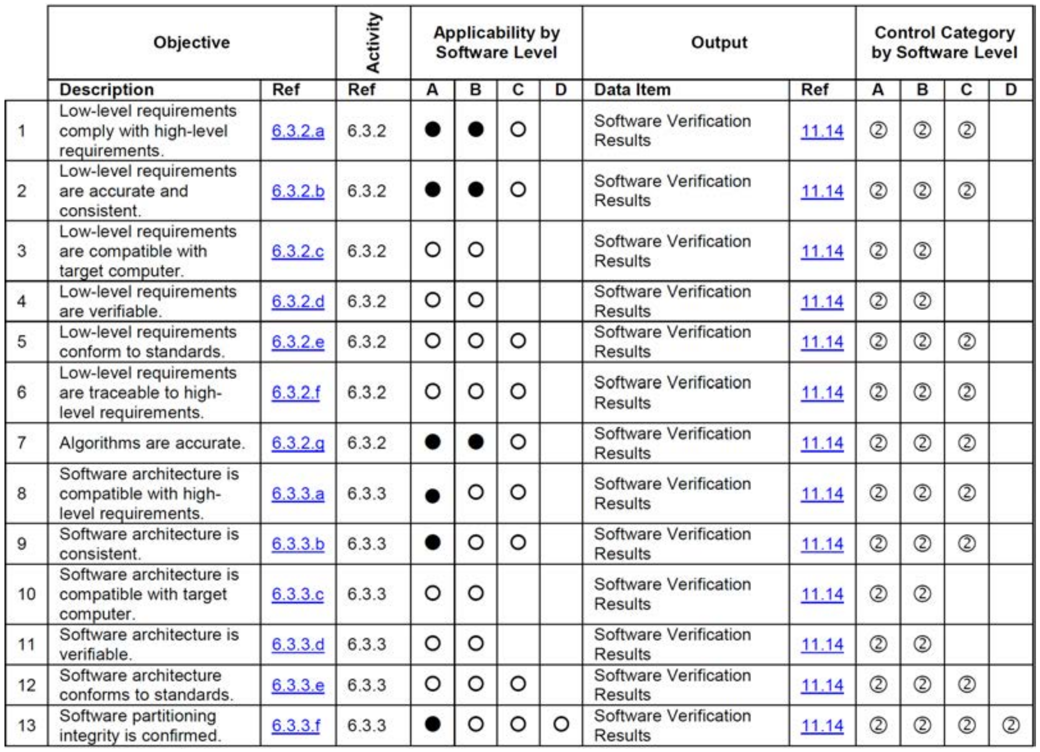
Be aware that all of these testing methods are automated by Parasoft's tool suite. You can get a glimpse of our C/C++ solution by taking a tour of Parasoft C/C++test.

The following tables list the set of objectives and expected outputs based on each software design assurance level in order to ensure airworthiness.



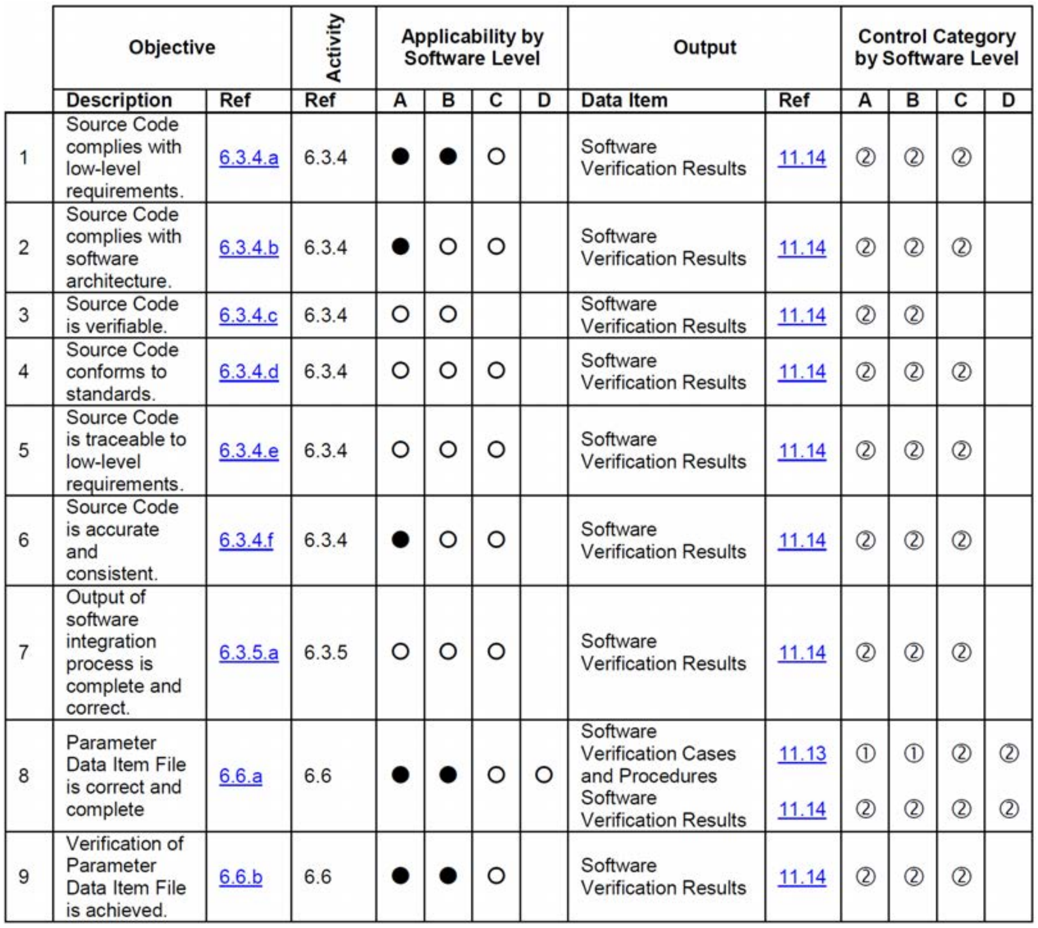
	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	High-level requirements comply with system requirements.	6.3.1.a	6.3.1	●	●	○	○	Software Verification Results	11.14	②	②	②	②
2	High-level requirements are accurate and consistent.	6.3.1.b	6.3.1	●	●	○	○	Software Verification Results	11.14	②	②	②	②
3	High-level requirements are compatible with target computer.	6.3.1.c	6.3.1	○	○			Software Verification Results	11.14	②	②		
4	High-level requirements are verifiable.	6.3.1.d	6.3.1	○	○	○		Software Verification Results	11.14	②	②	②	
5	High-level requirements conform to standards.	6.3.1.e	6.3.1	○	○	○		Software Verification Results	11.14	②	②	②	
6	High-level requirements are traceable to system requirements.	6.3.1.f	6.3.1	○	○	○	○	Software Verification Results	11.14	②	②	②	②
7	Algorithms are accurate.	6.3.1.g	6.3.1	●	●	○		Software Verification Results	11.14	②	②	②	

Table 2-4: DO-178C Table A-3 Verification of outputs of software requirements process



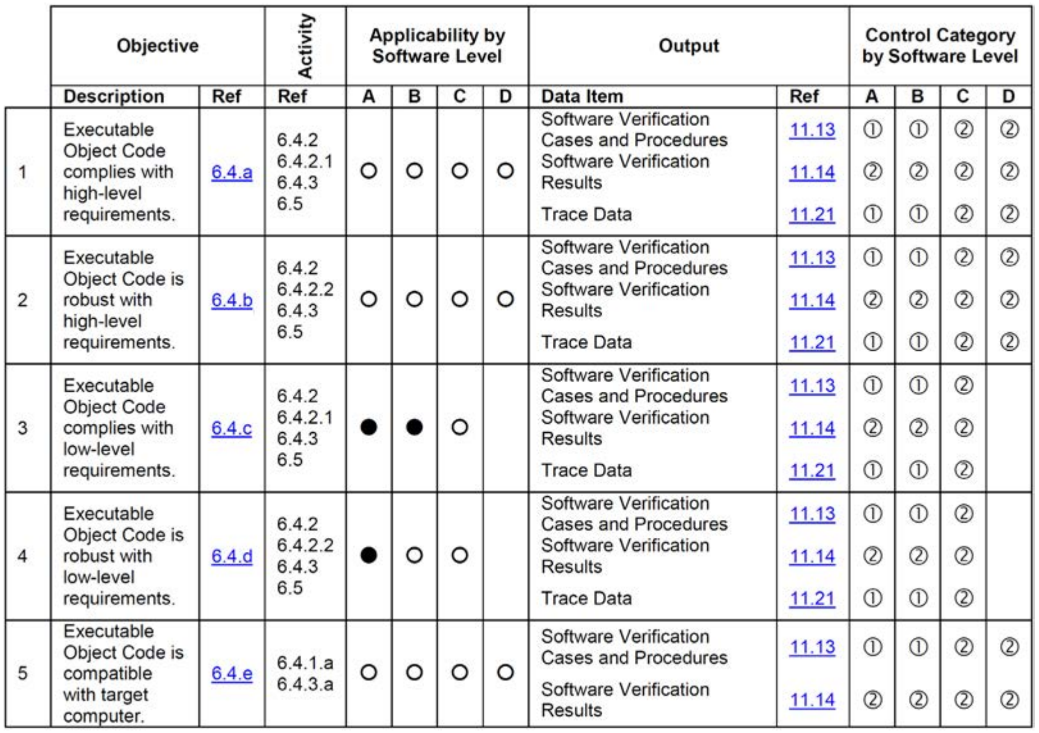
	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Low-level requirements comply with high-level requirements.	6.3.2.a	6.3.2	●	●	○		Software Verification Results	11.14	②	②	②	
2	Low-level requirements are accurate and consistent.	6.3.2.b	6.3.2	●	●	○		Software Verification Results	11.14	②	②	②	
3	Low-level requirements are compatible with target computer.	6.3.2.c	6.3.2	○	○			Software Verification Results	11.14	②	②		
4	Low-level requirements are verifiable.	6.3.2.d	6.3.2	○	○			Software Verification Results	11.14	②	②		
5	Low-level requirements conform to standards.	6.3.2.e	6.3.2	○	○	○		Software Verification Results	11.14	②	②	②	
6	Low-level requirements are traceable to high-level requirements.	6.3.2.f	6.3.2	○	○	○		Software Verification Results	11.14	②	②	②	
7	Algorithms are accurate.	6.3.2.g	6.3.2	●	●	○		Software Verification Results	11.14	②	②	②	
8	Software architecture is compatible with high-level requirements.	6.3.3.a	6.3.3	●	○	○		Software Verification Results	11.14	②	②	②	
9	Software architecture is consistent.	6.3.3.b	6.3.3	●	○	○		Software Verification Results	11.14	②	②	②	
10	Software architecture is compatible with target computer.	6.3.3.c	6.3.3	○	○			Software Verification Results	11.14	②	②		
11	Software architecture is verifiable.	6.3.3.d	6.3.3	○	○			Software Verification Results	11.14	②	②		
12	Software architecture conforms to standards.	6.3.3.e	6.3.3	○	○	○		Software Verification Results	11.14	②	②	②	
13	Software partitioning integrity is confirmed.	6.3.3.f	6.3.3	●	○	○	○	Software Verification Results	11.14	②	②	②	②

Table 2-5: DO-178C Table A-4 Verification of outputs of software design process



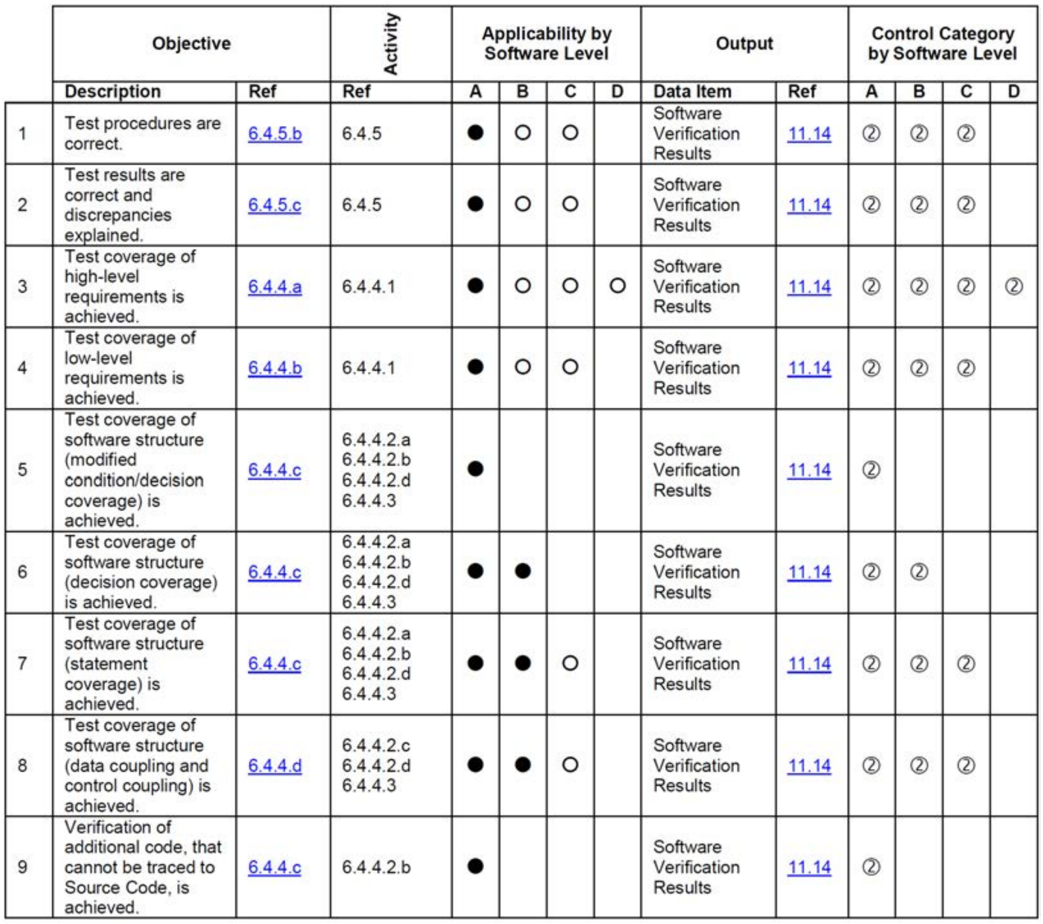
	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Source Code complies with low-level requirements.	6.3.4.a	6.3.4	●	●	○		Software Verification Results	11.14	②	②	②	
2	Source Code complies with software architecture.	6.3.4.b	6.3.4	●	○	○		Software Verification Results	11.14	②	②	②	
3	Source Code is verifiable.	6.3.4.c	6.3.4	○	○			Software Verification Results	11.14	②	②		
4	Source Code conforms to standards.	6.3.4.d	6.3.4	○	○	○		Software Verification Results	11.14	②	②	②	
5	Source Code is traceable to low-level requirements.	6.3.4.e	6.3.4	○	○	○		Software Verification Results	11.14	②	②	②	
6	Source Code is accurate and consistent.	6.3.4.f	6.3.4	●	○	○		Software Verification Results	11.14	②	②	②	
7	Output of software integration process is complete and correct.	6.3.5.a	6.3.5	○	○	○		Software Verification Results	11.14	②	②	②	
8	Parameter Data Item File is correct and complete	6.6.a	6.6	●	●	○	○	Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	② ②
9	Verification of Parameter Data Item File is achieved.	6.6.b	6.6	●	●	○		Software Verification Results	11.14	②	②	②	

Table 2-6: DO-178C Table A-5 Verification of outputs of software coding and integration processes



	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Executable Object Code complies with high-level requirements.	6.4.a	6.4.2 6.4.2.1 6.4.3 6.5	○	○	○	○	Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	② ② ②
2	Executable Object Code is robust with high-level requirements.	6.4.b	6.4.2 6.4.2.2 6.4.3 6.5	○	○	○	○	Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	② ② ②
3	Executable Object Code complies with low-level requirements.	6.4.c	6.4.2 6.4.2.1 6.4.3 6.5	●	●	○		Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	
4	Executable Object Code is robust with low-level requirements.	6.4.d	6.4.2 6.4.2.2 6.4.3 6.5	●	○	○		Software Verification Cases and Procedures Software Verification Results Trace Data	11.13 11.14 11.21	① ② ①	① ② ①	② ② ②	
5	Executable Object Code is compatible with target computer.	6.4.e	6.4.1.a 6.4.3.a	○	○	○	○	Software Verification Cases and Procedures Software Verification Results	11.13 11.14	① ②	① ②	② ②	② ②

Table 2-7: DO-178C Table A-6 Testing of outputs of integration process



	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Test procedures are correct.	6.4.5.b	6.4.5	●	○	○		Software Verification Results	11.14	②	②	②	
2	Test results are correct and discrepancies explained.	6.4.5.c	6.4.5	●	○	○		Software Verification Results	11.14	②	②	②	
3	Test coverage of high-level requirements is achieved.	6.4.4.a	6.4.4.1	●	○	○	○	Software Verification Results	11.14	②	②	②	②
4	Test coverage of low-level requirements is achieved.	6.4.4.b	6.4.4.1	●	○	○		Software Verification Results	11.14	②	②	②	
5	Test coverage of software structure (modified condition/decision coverage) is achieved.	6.4.4.c	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3	●				Software Verification Results	11.14	②			
6	Test coverage of software structure (decision coverage) is achieved.	6.4.4.c	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3	●	●			Software Verification Results	11.14	②	②		
7	Test coverage of software structure (statement coverage) is achieved.	6.4.4.c	6.4.4.2.a 6.4.4.2.b 6.4.4.2.d 6.4.4.3	●	●	○		Software Verification Results	11.14	②	②	②	
8	Test coverage of software structure (data coupling and control coupling) is achieved.	6.4.4.d	6.4.4.2.c 6.4.4.2.d 6.4.4.3	●	●	○		Software Verification Results	11.14	②	②	②	
9	Verification of additional code, that cannot be traced to Source Code, is achieved.	6.4.4.c	6.4.4.2.b	●				Software Verification Results	11.14	②			

Table 2-8: DO-178C Table A-7 Verification of process results

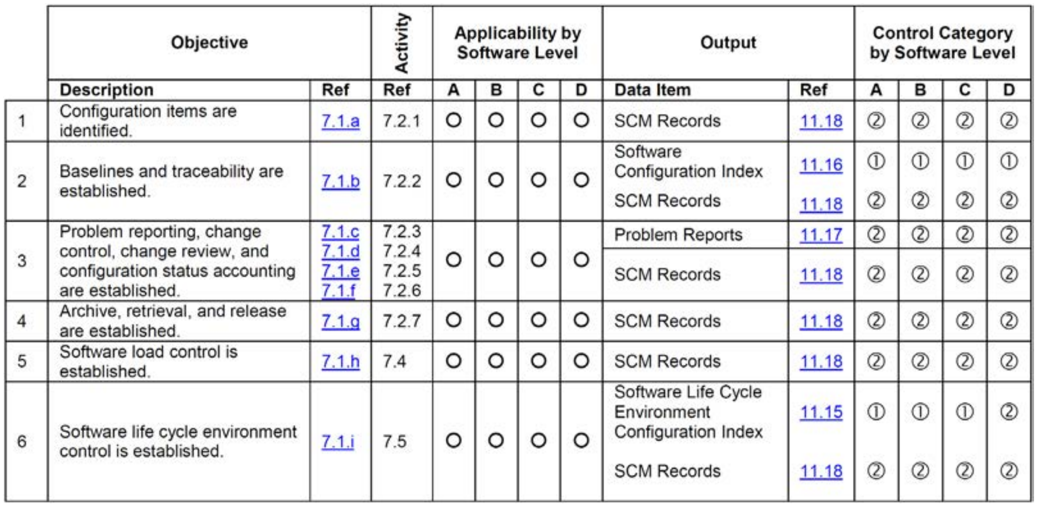
Software Configuration Management Process, Section 7

Section 7 discusses the objectives and activities of the software configuration management process. You need to be able to define and control configurations of the software throughout the software life cycle. Organizations or teams need to have source baselines, versioning, change control, change review, protection against unauthorized changes, problem reporting, and much more.

These are the software configuration management process activities:

1. Configuration identification
2. Baselines and traceability
3. Problem reporting, tracking, and corrective action
4. Change control
5. Change review
6. Configuration status accounting
7. Archive, retrieval, and release

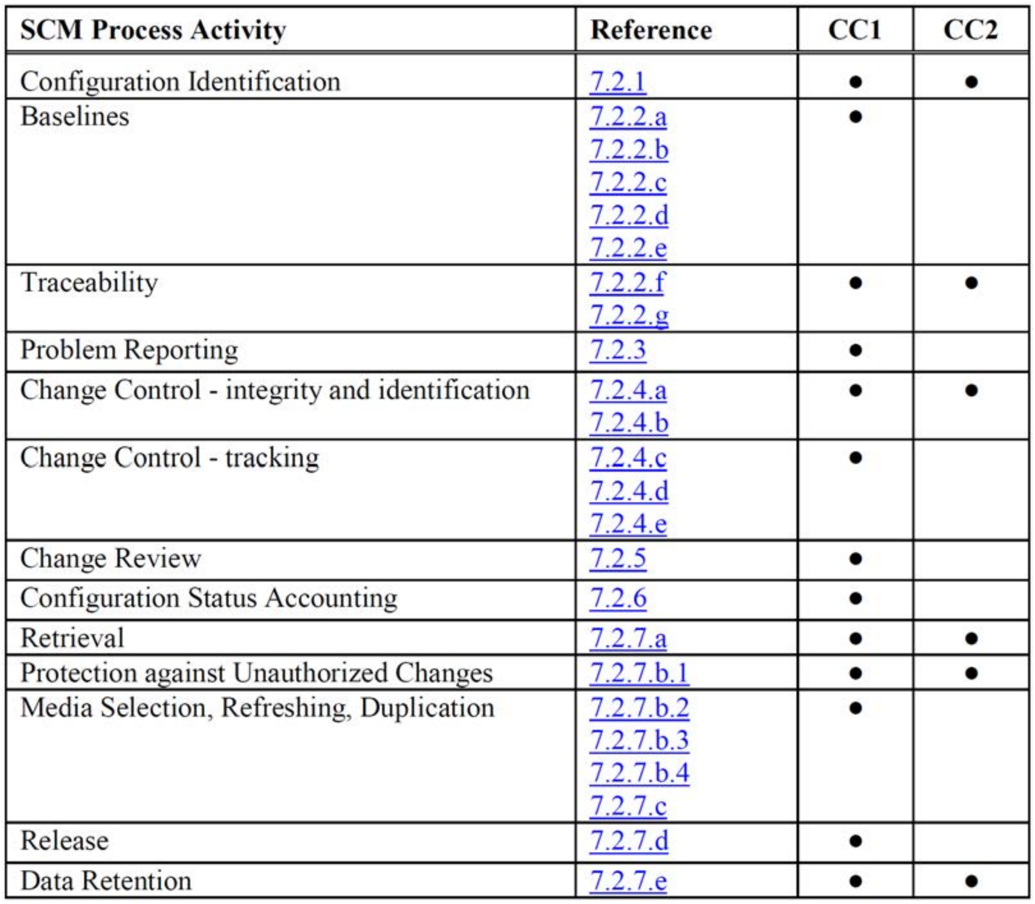
These activities are further detailed as objectives and their output. The objectives include being able to control item characteristic changes, record and report change control processing, and implementation status.



	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Configuration items are identified.	7.1.a	7.2.1	○	○	○	○	SCM Records	11.18	②	②	②	②
2	Baselines and traceability are established.	7.1.b	7.2.2	○	○	○	○	Software Configuration Index	11.16	①	①	①	①
								SCM Records	11.18	②	②	②	②
3	Problem reporting, change control, change review, and configuration status accounting are established.	7.1.c 7.1.d 7.1.e 7.1.f	7.2.3 7.2.4 7.2.5 7.2.6	○	○	○	○	Problem Reports	11.17	②	②	②	②
								SCM Records	11.18	②	②	②	②
4	Archive, retrieval, and release are established.	7.1.g	7.2.7	○	○	○	○	SCM Records	11.18	②	②	②	②
5	Software load control is established.	7.1.h	7.4	○	○	○	○	SCM Records	11.18	②	②	②	②
6	Software life cycle environment control is established.	7.1.i	7.5	○	○	○	○	Software Life Cycle Environment Configuration Index	11.15	①	①	①	②
								SCM Records	11.18	②	②	②	②

Table 2-9: DO-178C Table A-8 Software configuration management process

In Table A-8, notice the "Control Category by Software Level" column. DO-178C specifies which items must be treated as Control Category 1 or 2 based on the project's DAL. Items treated as Control Category 1 (CC1) must undergo full problem reporting processes, formal change review, and release processes. CC2 items do not need to undergo these more formal processes, but they must still comply with configuration identification and traceability needs, be protected against unauthorized changes, and satisfy applicable data retention requirements. The map between CC1 and CC2 data is found in the following table.



SCM Process Activity	Reference	CC1	CC2
Configuration Identification	7.2.1	●	●
Baselines	7.2.2.a 7.2.2.b 7.2.2.c 7.2.2.d 7.2.2.e	●	
Traceability	7.2.2.f 7.2.2.g	●	●
Problem Reporting	7.2.3	●	
Change Control - integrity and identification	7.2.4.a 7.2.4.b	●	●
Change Control - tracking	7.2.4.c 7.2.4.d 7.2.4.e	●	
Change Review	7.2.5	●	
Configuration Status Accounting	7.2.6	●	
Retrieval	7.2.7.a	●	●
Protection against Unauthorized Changes	7.2.7.b.1	●	●
Media Selection, Refreshing, Duplication	7.2.7.b.2 7.2.7.b.3 7.2.7.b.4 7.2.7.c	●	
Release	7.2.7.d	●	
Data Retention	7.2.7.e	●	●

Table 2-10: DO-178C SCM process activities associated with CC1 and CC2 data

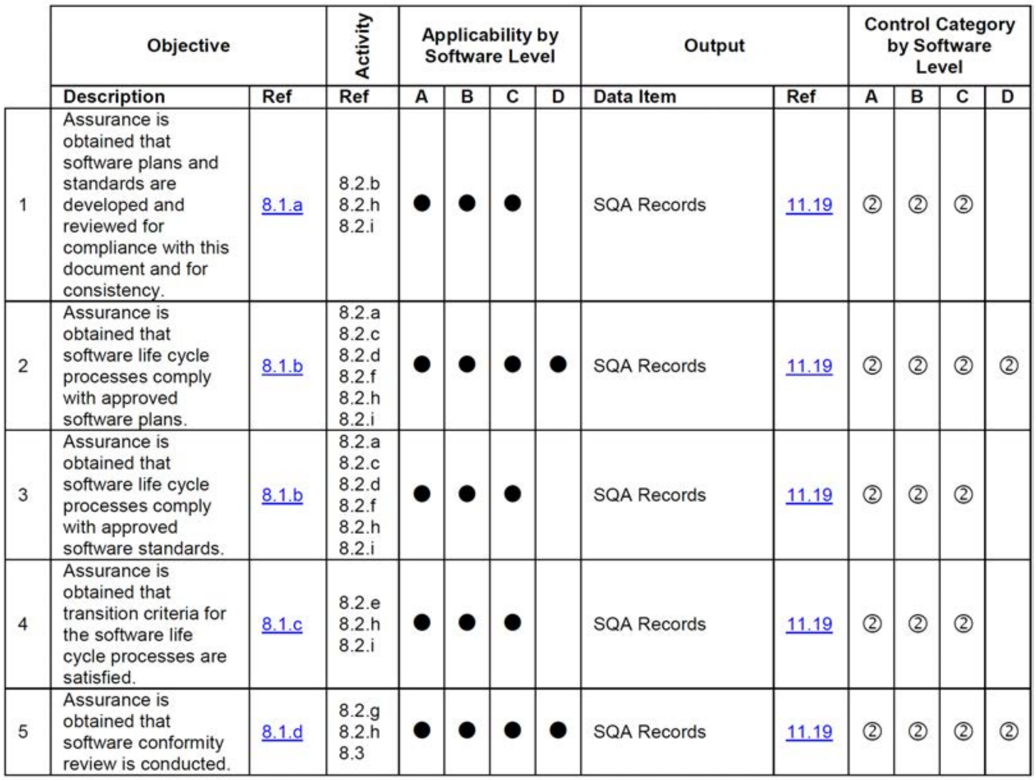
Software Quality Assurance Process, Section 8

The SQA process is captured in the Software Quality Assurance Plan, which is built during the software planning process. Outputs of the SQA process activities need to be recorded, evaluated, and tracked. Audits need to be performed and any deviations from the standards be resolved. The process entails providing assurance that:

» Software plans and standards are developed, reviewed, and will meet compliance.

» Artifacts, reports, and evidence are in place with approvals.

» Software product and software life cycle data conform to certification requirements.



	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Assurance is obtained that software plans and standards are developed and reviewed for compliance with this document and for consistency.	8.1.a	8.2.b 8.2.h 8.2.i	●	●	●		SQA Records	11.19	②	②	②	
2	Assurance is obtained that software life cycle processes comply with approved software plans.	8.1.b	8.2.a 8.2.c 8.2.d 8.2.f 8.2.h 8.2.i	●	●	●	●	SQA Records	11.19	②	②	②	②
3	Assurance is obtained that software life cycle processes comply with approved software standards.	8.1.b	8.2.a 8.2.c 8.2.d 8.2.f 8.2.h 8.2.i	●	●	●		SQA Records	11.19	②	②	②	
4	Assurance is obtained that transition criteria for the software life cycle processes are satisfied.	8.1.c	8.2.e 8.2.h 8.2.i	●	●	●		SQA Records	11.19	②	②	②	
5	Assurance is obtained that software conformity review is conducted.	8.1.d	8.2.g 8.2.h 8.3	●	●	●	●	SQA Records	11.19	②	②	②	②

Table 2-11: DO-178C Table A-9 Software Quality Assurance Process

Certification Liaison Process, Section 9

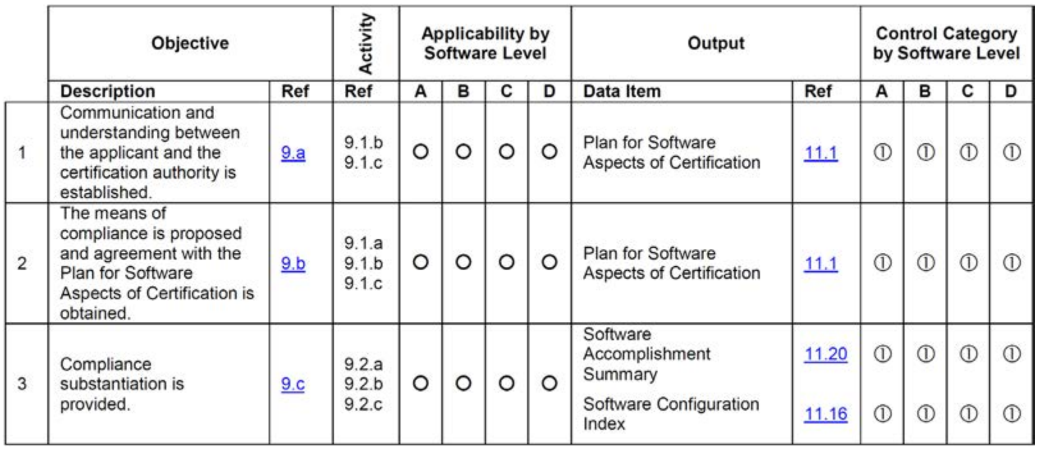
Section 9 discusses the certification liaison process and its objectives, which include the following:

» Establish communication and understanding between the applicant and the certification authority throughout the software life cycle to assist the certification process.

» Gain agreement on the means of compliance through approval of the Plan for Software Aspects of Certification.

» Provide compliance substantiation.

Your organization will need to produce the Plan for Software Aspects of Certification (PSAC), which will contain the certification liaison process. The PSAC will include plans on resolving issues identified by the certification liaison and obtaining agreement on the plan. The table below lists the set of objectives and expected output artifacts.



	Objective		Activity	Applicability by Software Level				Output		Control Category by Software Level			
	Description	Ref	Ref	A	B	C	D	Data Item	Ref	A	B	C	D
1	Communication and understanding between the applicant and the certification authority is established.	9.a	9.1.b 9.1.c	○	○	○	○	Plan for Software Aspects of Certification	11.1	①	①	①	①
2	The means of compliance is proposed and agreement with the Plan for Software Aspects of Certification is obtained.	9.b	9.1.a 9.1.b 9.1.c	○	○	○	○	Plan for Software Aspects of Certification	11.1	①	①	①	①
3	Compliance substantiation is provided.	9.c	9.2.a 9.2.b 9.2.c	○	○	○	○	Software Accomplishment Summary	11.20	①	①	①	①
								Software Configuration Index	11.16	①	①	①	①

Table 2-12: DO-178C Table A-10 Certification liaison process

Best practices for obtaining certification boil down to closely working with your certification liaison, who may be better known as your Designated Engineering Representative (DER), to evaluate for compliance, act on your behalf toward approval, and recommend that the FAA approve your certification.

Overview of Certification Process, Section 10

Section 10 is for informational purposes only regarding the certification process. It mentions the types of systems and equipment to which certification applies. It specifies that certification authorities do not certify software as a unique stand-alone product. It must be part of the airborne system or equipment.

"'Certification' applies to aircraft, engines, or propellers; and, in respect of some certification authorities, auxiliary power units. The certification authorities consider the software as part of the airborne system or equipment installed on the certified product; that is, the certification authorities do not certify the software as a unique, stand-alone product."

Approval also depends upon a successful demonstration or review of the products produced.

Software Life Cycle Data, Section 11

Section 11 discusses artifacts like the data and documentation produced during the software life cycle. The data needs to be unambiguous, complete, verifiable, consistent, modifiable, and traceable. It also must be in various forms like electronic and printed. Parasoft's automated report generation and analytics web dashboard provide much of the information needed within various artifacts and documents.

The artifacts to be produced during the software life cycle include the source code, object code, test cases, results, problem reports, and, of course, the plans. Here's the full list.

- Plan for software aspects of certification
- Software development plan
- Software verification plan
- Software configuration management plan
- Software quality assurance plan
- Software requirements standards
- Software design standards
- Software code standards
- Software requirements data
- Design description
- Source code
- Executable object code
- Software verification cases and procedures
- Software verification results
- Software life cycle environment configuration index
- Software configuration index
- Problem reports
- Software configuration management records
- Software quality assurance records
- Software accomplishment summary
- Trace data
- Parameter data item file

Additional Considerations, Section 12

Section 12 provides additional guidance and consideration on topics that can have an impact on objectives and activities in the software life cycle. For example, the use of or modifications to previously developed software. Section 12 provides additional clarification and activities to perform that help ensure safety and recertification. Here are just some other considerations include:

- » Changes to the development environment such as processor, programming language, auto code generator, development tools, and the like.
- » Upgrading a development baseline.
- » Use of already certified software on an alternate type of aircraft.
- » Use of certified software where there's a change in the compiler or processor.

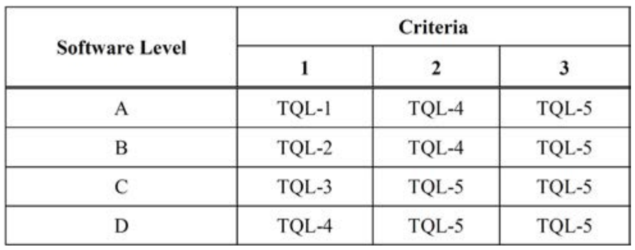
Based on the consideration, section 12 provides additional objectives in software configuration management, software quality assurance, development tool qualification, and more.

Section 12 covers the importance of "Tool Qualification" and determining if its needed. This is because if a tool is used that eliminates, reduces, or automates processes, teams need to take into consideration whether the tool might introduce errors into the life cycle.

The following criteria should be used to determine the impact of the tool:

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

There are five levels of tool qualification, TQL-1 through TQL-5, that are determined by the tool use and its potential impact on the software life cycle. TQL-1 is the most rigorous level. The tool qualification level needs to be coordinated with the certification authority.



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

Table 2-13: DO-178C Tool qualification level determination

The objectives, activities, guidance, and life cycle data required for each tool qualification level are described in DO-330, "Software Tool Qualification Considerations."

Parasoft supports DO-178C and DO-330 conformant tool qualification processes with an automated tool qualification kit. The Tool Qualification Kit automates the process of creating the supporting documentation required in using C/C++test for static analysis, unit testing, and coverage requirements.

Parasoft's Tool Qualification Kit reduces the time taken to perform the tool qualification and the potential for human error by leveraging automation to guide users through the following workflow:

1. Specify the use cases and capabilities to be used on the project.
2. Quickly map known issues in the tool you're qualifying to the features of the tool you're using in development.
3. Plan and capture the results of manual testing efforts.
4. Execute automated tests.
5. Bring all the data together and generate the critical documents.

Requirements for Compliance in Testing

Static Analysis

Static code analysis is the analysis of code without actual code execution. Static analysis exposes safety and security vulnerabilities in the code by applying a comprehensive set of code analysis techniques including:

- Pattern-based analysis
- Data flow analysis
- Control flow analysis
- Abstract interpretation
- Code metrics and more

These methods identify memory buffer overflows, divide by zero, use of insecure libraries, organization coding rules, directive violations, and so forth.

In DO-178C, the objectives for static analysis fall under Section 6 related to software verification processes. The objectives of static analysis focus on ensuring that the software code is free from certain types of defects and follow good coding practices.

For example, Section 6.3.4 Review and Analysis of Source Code, provides an overview of the software verification activities required to review code in terms of compliance, verifiability, and traceability. However, this section also specifies the need to inspect the code for conformance to standards, accuracy, and consistency, all of which are good applications for static analysis.

While DO-178C does not have a specific requirement for static analysis, the guidelines and objectives related to static analysis are spread across sections within Chapter 6. It's crucial to interpret and apply these guidelines appropriately in the context of the project to ensure compliance with DO-178C for the certification of airborne software.

Some of the typical requirements for static analysis in DO-178C may include the following.

1. **Tools**. Selecting and using appropriate static analysis tools to analyze the source code for defects and compliance with coding standards.
2. **Coding standards.** Ensuring that the software code follows a set of predefined coding standards or guidelines to improve readability, maintainability, and safety.
3. **Verification of software requirements.** Using static analysis to verify that the software code correctly implements the software requirements and that there are no discrepancies between the requirements and the code.
4. **Defect identification and removal.** Identifying and removing defects such as coding errors, potential runtime issues, and other flaws through static analysis.
5. **Traceability.** Ensuring that the static analysis results are appropriately documented and traced back to the specific requirements, source code, and any corrective actions taken.
6. **Tool qualification.** If static analysis tools are used for safety-critical code, ensure that these tools are qualified appropriately according to DO-330 Software Tool Qualification Considerations and that their usage is documented.

Most of these verification activities are supported through the automation of static analysis using modern advanced tools like Parasoft C/C++test. In addition, Parasoft provides code metrics on maintainability, clarity, testability, portability, robustness, reusability, complexity, and support for team code peer reviews. Dynamic analysis, unit testing, and other runtime error detection is also provided.

Early Defect Detection

Early defect detection with static analysis tools can significantly improve compliance with DO-178C by addressing potential coding issues and vulnerabilities in the software development process. Static analysis analyzes source code without executing it, identifying defects and potential issues based on predefined rules.

Static analysis tools can detect coding errors and bugs in the source code early in the development process. By identifying and fixing these errors early on, the development team can prevent such defects from propagating into later stages of development, where they might be more difficult and costly to fix.

Safety-critical software used in airborne systems must be protected from potential security vulnerabilities. Static analysis tools can identify potential security weaknesses in the code, such as buffer overflows, input validation issues, and other security-related defects. Addressing these vulnerabilities early in the development process enhances the security posture of the software.

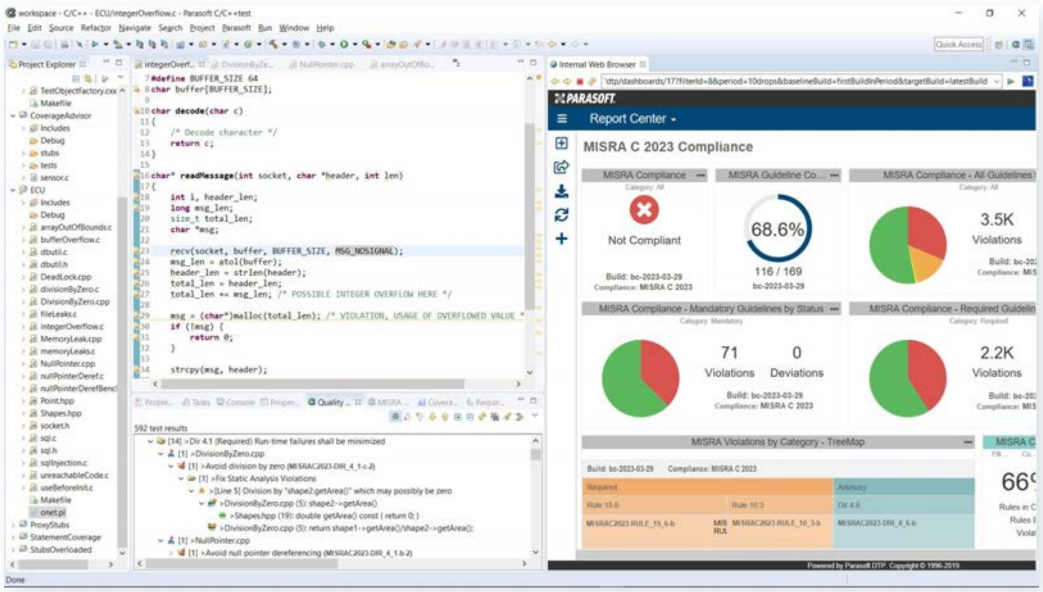
DO-178C requires comprehensive verification activities throughout the software development life cycle (Chapter 6). Static analysis, being a form of static verification, allows for early verification of the source code. By finding and addressing defects early on, the software can progress through subsequent verification stages with greater confidence, saving time and effort in the long run.

By adopting static analysis early in the software development process, in conjunction with other verification and validation methods, teams can proactively address defects and security vulnerabilities. This leads to a more streamlined certification process and a higher likelihood of producing reliable and safe software for use in airborne systems.

Some of the common types of defects that Parasoft C++test static analysis can detect include:

- Null pointer dereference
- Memory leaks
- Buffer overflows and underflows
- Uninitialized variables
- Dead code
- Resource management issues
- Concurrency issues
- Security vulnerabilities
- Performance issues
- Complexity metrics

These are just some examples of the types of defects that Parasoft C++test static analysis can detect. Additionally, static analysis tools like Parasoft C++test can be customized to include or exclude certain types of checks based on the project's specific requirements and coding standards.


workspace - C/C++ - ECU/integerOverflow.c - Parasoft C/C++test
File Edit Source Refactor Navigate Search Project Parasoft Run Window Help
Project Explorer
integerOverf... DivisionByZe... NullPointer.cpp arrayOutOfBo...
Internal Web Browser
PARASOFT
Report Center
MISRA C 2023 Compliance
MISRA Compliance
Not Compliant
MISRA Guideline Co...
68.6%
116 / 169
MISRA Compliance - All Guidelines
3.5K Violations
MISRA Compliance - Mandatory Guidelines by Status
71 Violations
0 Deviations
MISRA Compliance - Required Guidelin
2.2K Violations
MISRA Violations by Category - TreeMap
Powered by Parasoft DTP. Copyright © 1996-2019

```
#define BUFFER_SIZE 64
char buffer[BUFFER_SIZE];

char decode(char c)
{
    /* Decode character */
    return c;
}

char* readMessage(int socket, char *header, int len)
{
    int i, header_len;
    long msg_len;
    size_t total_len;
    char *msg;

    recv(socket, buffer, BUFFER_SIZE, MSG_NOSIGNAL);
    msg_len = atol(buffer);
    header_len = strlen(header);
    total_len = header_len;
    total_len += msg_len; /* POSSIBLE INTEGER OVERFLOW HERE */

    msg = (char*)malloc(total_len); /* VIOLATION, USAGE OF OVERFLOWED VALUE *
    if (!msg) {
        return 0;
    }

    strcpy(msg, header);
```

```
592 test results
[14] >Dir 4.1 (Required) Run-time failures shall be minimized
  [1] >DivisionByZero.cpp
    [1] >Avoid division by zero (MISRAC2023-DIR_4_1-c-2)
      [1] >Fix Static Analysis Violations
        >[Line 5] Division by "shape2.getArea()" which may possibly be zero
          >DivisionByZero.cpp (5): shape2->getArea()
            >Shapes.hpp (19): double getArea() const { return 0; }
          >DivisionByZero.cpp (5): return shape1->getArea()/shape2->getArea();
  [1] >NullPointer.cpp
    [1] >Avoid null pointer dereferencing (MISRAC2023-DIR_4_1-b-2)
```

Figure 3-7: Parasoft C/C++test and DTP dashboardParasoft

Coding Standards

Regarding coding standards, DO-178C does not prescribe a specific set of coding standards that must be followed. Instead, it provides guidelines and objectives for establishing and adhering to coding standards appropriate for the development of safety-critical airborne software.

The relevant sections in DO-178C that pertain to coding standards are primarily found in Chapter 6 Software Verification Process and Chapter 11 Software Lifecycle Data. Here's what DO-178C typically requires regarding coding standards.

» **Coding Standard Definition, Section 11.8.** Define coding standards for the project that should cover rules and guidelines related to programming practices, naming conventions, code layout, control structures, data structures, and other aspects of software coding.

» **Code Review, Section 6.3.4 d.** The emphasis is on the importance of conducting code reviews to ensure compliance with the coding standards. Code reviews involve thorough inspection of code and related artifacts. The process can be semi-automated with static analysis tools.

» **Traceability to Coding Standards, Section 6.3.4 e.** There should be traceability between the software requirements and the coding standards. The code should be written in accordance with the established coding standards and this relationship should be documented.

DO-178C recognizes that different projects may have different coding standards (for example, MISRA C/C++, CERT C/C++, CWE, OWASP, DISA ASD STIG, and so on) depending on factors such as the complexity of the software, the criticality of the system, and the development environment. Therefore, the specific coding standards and rules are determined by the development team while still satisfying the guidelines outlined above.

A vital part of the certification evidence required for DO-178C compliance is the documentation collected during these reviews and the verification process. It's important that the coding standard support the inspection and the documentation processes required.

MISRA C:2023

MISRA C is a set of coding guidelines for the C programming language, versions C89/C90, C99, C11, and C18. The focus of the standard is increasing safety of software by pre-emptively preventing programmers from making coding mistakes that can lead to runtime failures (and possible safety concerns) by avoiding known problem constructs in the C language.

MISRA C can help satisfy the requirements of DO-178C, which is the software standard used for the certification of airborne systems. Here's how MISRA C can fulfill the requirements.

1. Checks all the boxes of the coding standard requirements listed in the previous section.
2. Provides a well-defined and widely recognized coding standard that can be adopted by the development team to create consistent and reliable code.
3. Involves regular code reviews to ensure compliance with the standard.

The adoption of MISRA C helps minimize the potential for coding errors and ambiguities, leading to improved safety, security, and reliability of the software. The coding standard's focus on robustness and code correctness aligns well with the objectives of DO-178C to ensure the development of high-integrity software for airborne systems.

It's important to note that MISRA C is not a guarantee of certification compliance by itself. It's one of the tools and processes that contribute to the overall software development and verification activities required for DO-178C certification. Additionally, each project may have specific requirements and constraints, so the MISRA C standard may need to be tailored or supplemented with project-specific coding rules and practices.

Over the years, many developers of embedded systems were—and still are—complaining that MISRA C was too stringent of a standard and that the cost of writing fully compliant code was difficult to justify. Realistically, given that MISRA C is applied in safety-critical software, the value of applying the standard to a project depends on factors such as:

- Risk of a system malfunction because of a software failure
- Cost of a system failure to the business
- Development tools and target platform
- Level of developer’s expertise

Programmers must find a practical middle ground that satisfies the spirit of the standard and still claim MISRA compliance without wasting effort on non value added activities.

Proof of MISRA Compliance

A key problem that developers of safety-critical software encounter is how to demonstrate and prove compliance at the end of the project. There's a tendency to add more information into the reports than is required. It can become a contentious issue resulting in wasted time and effort if the evaluation criteria are based on subjective opinions from the various stakeholders.

A recommended approach to improving the evaluation of compliance readiness is to use existing templates for both the final compliance and tool qualification report. If the information is not required by the standard, avoid adding it. Combining extra information is not only a waste of time, but also introduces a risk of delaying an audit process. Having the documentation auto generated as Parasoft does, is the ultimate solution.

The MISRA Compliance: 2020 document is also helping organizations to use a common language articulating the compliance requirements by defining the following artifacts:

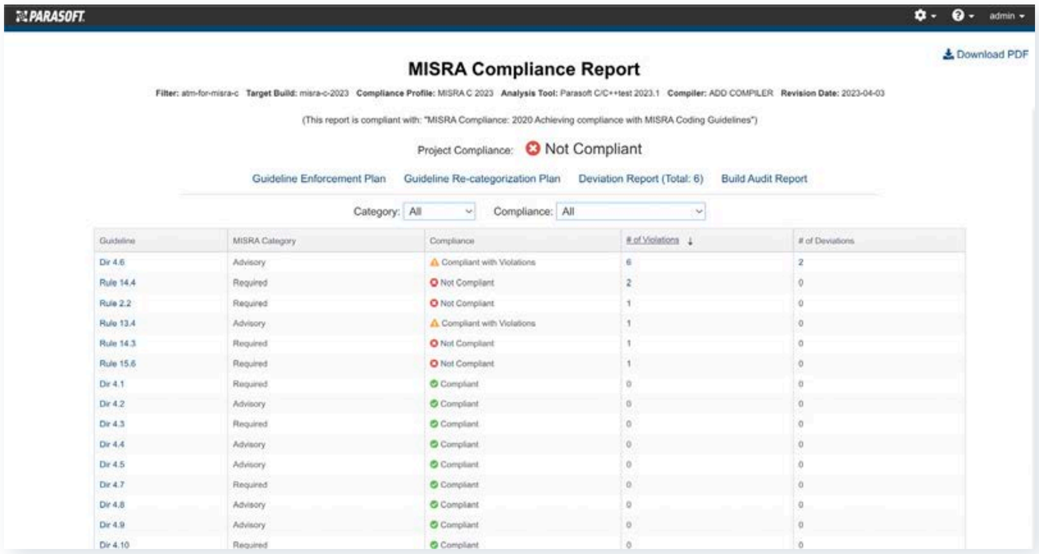
» Guidelines Compliance Summary

» Guideline Enforcement Plan

» Deviations Report

» Guideline Re-categorization Plan

The following Parasoft's screenshots show auto-generated reports with links to other records and/or expansion of information on the page.



MISRA Compliance Report

Filter: atm-for-misra-c **Target Build:** misra-c-2023 **Compliance Profile:** MISRA C 2023 **Analysis Tool:** Parasoft C/C++test 2023.1 **Compiler:** ADD COMPILER **Revision Date:** 2023-04-03

(This report is compliant with: "MISRA Compliance: 2020 Achieving compliance with MISRA Coding Guidelines")

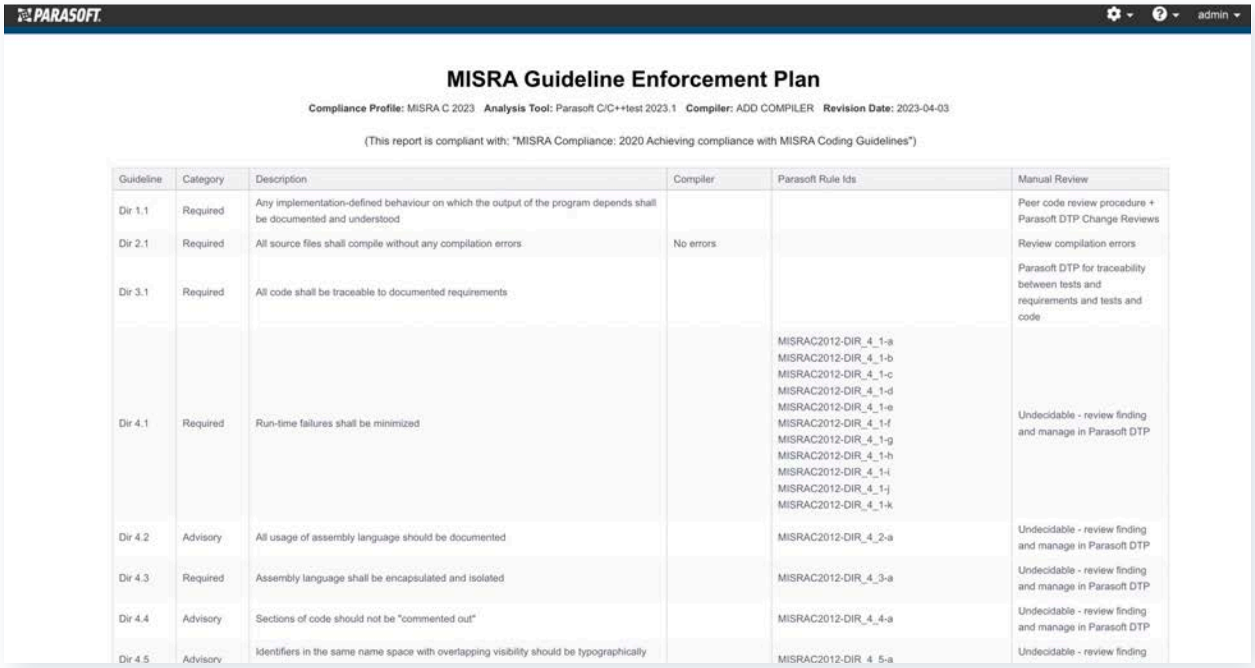
Project Compliance: Not Compliant

Guideline Enforcement Plan Guideline Re-categorization Plan Deviation Report (Total: 6) Build Audit Report

Category: All Compliance: All

Guideline	MISRA Category	Compliance	# of Violations	# of Deviations
Dir 4.6	Advisory	⚠ Compliant with Violations	6	2
Rule 14.4	Required	✖ Not Compliant	2	0
Rule 2.2	Required	✖ Not Compliant	1	0
Rule 13.4	Advisory	⚠ Compliant with Violations	1	0
Rule 14.3	Required	✖ Not Compliant	1	0
Rule 15.6	Required	✖ Not Compliant	1	0
Dir 4.1	Required	✓ Compliant	0	0
Dir 4.2	Advisory	✓ Compliant	0	0
Dir 4.3	Required	✓ Compliant	0	0
Dir 4.4	Advisory	✓ Compliant	0	0
Dir 4.5	Advisory	✓ Compliant	0	0
Dir 4.7	Required	✓ Compliant	0	0
Dir 4.8	Advisory	✓ Compliant	0	0
Dir 4.9	Advisory	✓ Compliant	0	0
Dir 4.10	Required	✓ Compliant	0	0

Figure 3-8: The Guidelines Compliance Summary is the primary record of overall project compliance.



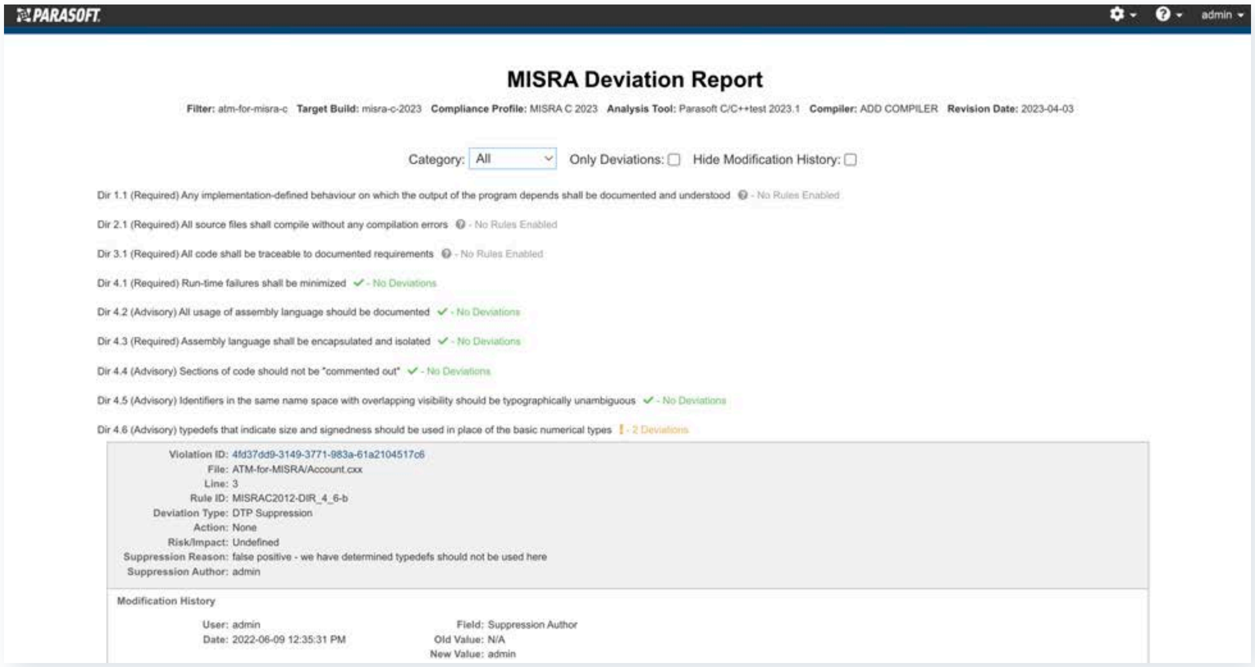
MISRA Guideline Enforcement Plan

Compliance Profile: MISRA C 2023 **Analysis Tool:** Parasoft C/C++test 2023.1 **Compiler:** ADD COMPILER **Revision Date:** 2023-04-03

(This report is compliant with: "MISRA Compliance: 2020 Achieving compliance with MISRA Coding Guidelines")

Guideline	Category	Description	Compiler	Parasoft Rule Ids	Manual Review
Dir 1.1	Required	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood			Peer code review procedure + Parasoft DTP Change Reviews
Dir 2.1	Required	All source files shall compile without any compilation errors	No errors		Review compilation errors
Dir 3.1	Required	All code shall be traceable to documented requirements			Parasoft DTP for traceability between tests and requirements and tests and code
Dir 4.1	Required	Run-time failures shall be minimized		MISRAC2012-DIR_4_1-a MISRAC2012-DIR_4_1-b MISRAC2012-DIR_4_1-c MISRAC2012-DIR_4_1-d MISRAC2012-DIR_4_1-e MISRAC2012-DIR_4_1-f MISRAC2012-DIR_4_1-g MISRAC2012-DIR_4_1-h MISRAC2012-DIR_4_1-i MISRAC2012-DIR_4_1-j MISRAC2012-DIR_4_1-k	Undecidable - review finding and manage in Parasoft DTP
Dir 4.2	Advisory	All usage of assembly language should be documented		MISRAC2012-DIR_4_2-a	Undecidable - review finding and manage in Parasoft DTP
Dir 4.3	Required	Assembly language shall be encapsulated and isolated		MISRAC2012-DIR_4_3-a	Undecidable - review finding and manage in Parasoft DTP
Dir 4.4	Advisory	Sections of code should not be "commented out"		MISRAC2012-DIR_4_4-a	Undecidable - review finding and manage in Parasoft DTP
Dir 4.5	Advisory	Identifiers in the same name space with overlapping visibility should be typographically		MISRAC2012-DIR_4_5-a	Undecidable - review finding

Figure 3-9: The Guideline Enforcement Plan demonstrates how each MISRA guideline is verified.



MISRA Deviation Report

Filter: atm-for-misra-c **Target Build:** misra-c-2023 **Compliance Profile:** MISRA C 2023 **Analysis Tool:** Parasoft C/C++test 2023.1 **Compiler:** ADD COMPILER **Revision Date:** 2023-04-03

Category: All Only Deviations: ☐ Hide Modification History: ☐

Dir 1.1 (Required) Any implementation-defined behaviour on which the output of the program depends shall be documented and understood - No Rules Enabled

Dir 2.1 (Required) All source files shall compile without any compilation errors - No Rules Enabled

Dir 3.1 (Required) All code shall be traceable to documented requirements - No Rules Enabled

Dir 4.1 (Required) Run-time failures shall be minimized ✓ - No Deviations

Dir 4.2 (Advisory) All usage of assembly language should be documented ✓ - No Deviations

Dir 4.3 (Required) Assembly language shall be encapsulated and isolated ✓ - No Deviations

Dir 4.4 (Advisory) Sections of code should not be "commented out" ✓ - No Deviations

Dir 4.5 (Advisory) Identifiers in the same name space with overlapping visibility should be typographically unambiguous ✓ - No Deviations

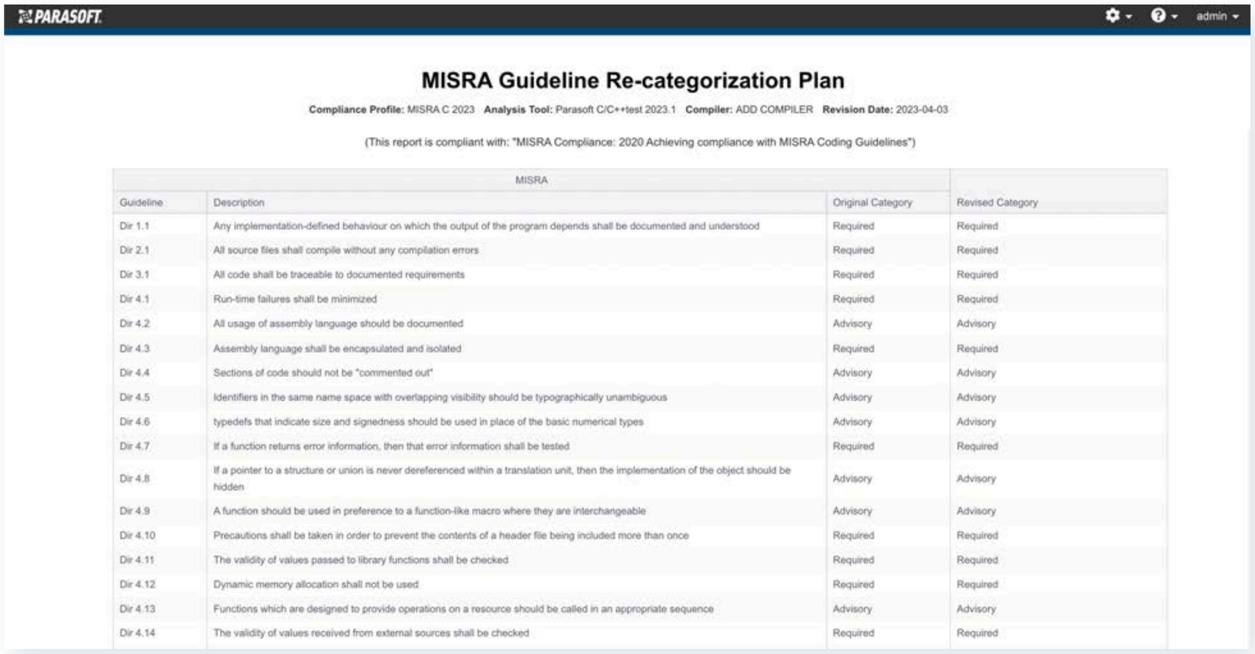
Dir 4.6 (Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types ! - 2 Deviations

Violation ID:	4fd37dd9-3149-3771-983a-61a2104517c6
File:	ATM-for-MISRA/Account.cxx
Line:	3
Rule ID:	MISRAC2012-DIR_4_6-b
Deviation Type:	DTP Suppression
Action:	None
Risk/Impact:	Undefined
Suppression Reason:	false positive - we have determined typedefs should not be used here
Suppression Author:	admin

Modification History

User:	admin	Field:	Suppression Author
Date:	2022-06-09 12:35:31 PM	Old Value:	N/A
		New Value:	admin

Figure 3-10: The Deviations Report documents all of the approved deviation permits.



PARASOFT admin

MISRA Guideline Re-categorization Plan

Compliance Profile: MISRA C 2023 **Analysis Tool:** Parasoft C/C++test 2023.1 **Compiler:** ADD COMPILER **Revision Date:** 2023-04-03

(This report is compliant with: "MISRA Compliance: 2020 Achieving compliance with MISRA Coding Guidelines")

MISRA			
Guideline	Description	Original Category	Revised Category
Dir 1.1	Any implementation-defined behaviour on which the output of the program depends shall be documented and understood	Required	Required
Dir 2.1	All source files shall compile without any compilation errors	Required	Required
Dir 3.1	All code shall be traceable to documented requirements	Required	Required
Dir 4.1	Run-time failures shall be minimized	Required	Required
Dir 4.2	All usage of assembly language should be documented	Advisory	Advisory
Dir 4.3	Assembly language shall be encapsulated and isolated	Required	Required
Dir 4.4	Sections of code should not be "commented out"	Advisory	Advisory
Dir 4.5	Identifiers in the same name space with overlapping visibility should be typographically unambiguous	Advisory	Advisory
Dir 4.6	typedefs that indicate size and signedness should be used in place of the basic numerical types	Advisory	Advisory
Dir 4.7	If a function returns error information, then that error information shall be tested	Required	Required
Dir 4.8	If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden	Advisory	Advisory
Dir 4.9	A function should be used in preference to a function-like macro where they are interchangeable	Advisory	Advisory
Dir 4.10	Precautions shall be taken in order to prevent the contents of a header file being included more than once	Required	Required
Dir 4.11	The validity of values passed to library functions shall be checked	Required	Required
Dir 4.12	Dynamic memory allocation shall not be used	Required	Required
Dir 4.13	Functions which are designed to provide operations on a resource should be called in an appropriate sequence	Advisory	Advisory
Dir 4.14	The validity of values received from external sources shall be checked	Required	Required

Figure 3-11: The Guideline Re-categorization Plan communicates how the guidelines are to be applied as part of the stakeholder/supplier relationship.

SEI/CERT

The Software Engineering Institute (SEI) Computer Emergency Response Team (CERT) has a set of guidelines to help developers create safer, more secure, and more reliable software. Started in 2006 at a meeting of the C Standard Committee, the first CERT C standard was published in 2008 and is constantly developing and evolving.

There's a book version published in 2016, but it doesn't include the latest updates. This standard doesn't have specific frozen releases like CWE Top 25 and OWASP Top 10. The standard arose from a large community of over 3,000 people with a focus on engineering and prevention. So the CERT secure coding standards focus on prevention of the root causes of security vulnerabilities rather than treating or managing the symptoms by searching for vulnerabilities.

The CERT coding guidelines are available for C, C++, Java, Perl, and Android. They fall into two main categories.

1. Rules
2. Recommendations

Rules are guidelines that are detectable by static analysis tools and require strict enforcement, while recommendations are guidelines that have a lower impact and are sometimes difficult to analyze automatically.

CERT includes a risk assessment system that combines likelihood of occurrence, severity, and relative difficulty of mitigation. This helps developers prioritize which guideline violations are the most important to investigate. The inclusion of mitigation effort to the guideline priority is an important addition to the CERT secure coding standards, which many other standards lack.

The cost factor allows for the creation of the CERT bullseye diagram in which the center bullseye is the highest severity guidelines that are more difficult to fix. The benefit of this prioritization is focusing on the most critical violations that provide the biggest bang for the buck in security improvement while helping the development team filter out less important warnings.

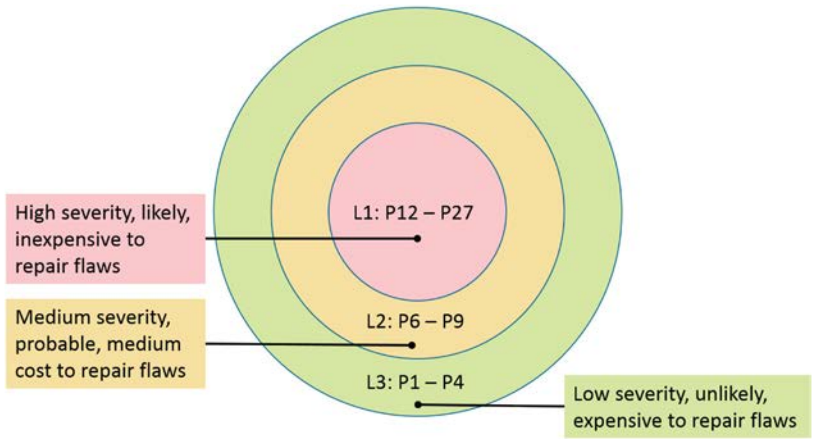

High severity, likely, inexpensive to repair flaws
Medium severity, probable, medium cost to repair flaws
L1: P12 – P27
L2: P6 – P9
L3: P1 – P4
Low severity, unlikely, expensive to repair flaws

Figure 3-12: SEI CERT vulnerability priority and cost diagram

SEI CERT C/C++ Conformance

According to the SEI CERT C documentation, conformance "requires that the code not contain any violations of the rules specified in this standard. If an exceptional condition is claimed, the exception must correspond to a predefined exceptional condition, and the application of this exception must be documented in the source code."

Although conformance is less specific than standards such as MISRA, the principles remain similar. Rules should be followed, and deviations should only occur rarely and be well documented. Recommendations should be used when possible and those that aren't needed should be documented.

Violations that persist in the source code need to be documented. However, no deviation is acceptable for performance or usability and the onus is on the developer to demonstrate that the deviation will not lead to a vulnerability.

Parasoft C/C++test provides comprehensive CERT compliance dashboard and reports. Individual compliance reports are available on demand based on the latest build of the software or any previous build.

These reports can be sorted and navigated to investigate violations in more detail. A conformance test plan is available to correlate the CERT guideline with the corresponding Parasoft static analysis checker, which is an important tool if conformance documentation is needed for audit purposes. In addition, all the interesting reports, as specified by the team, are in a single PDF available for download by auditors.

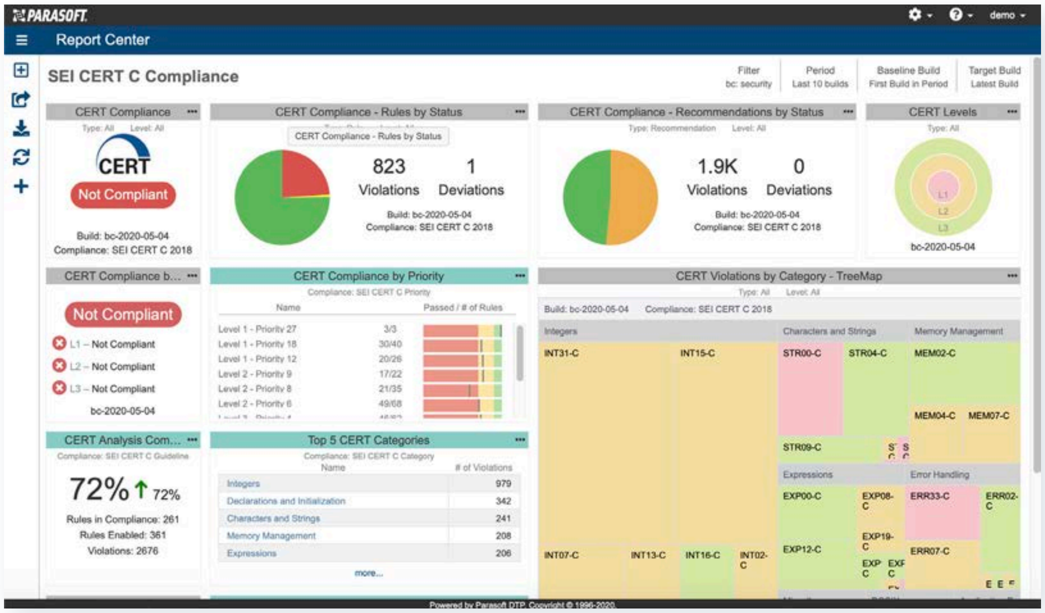
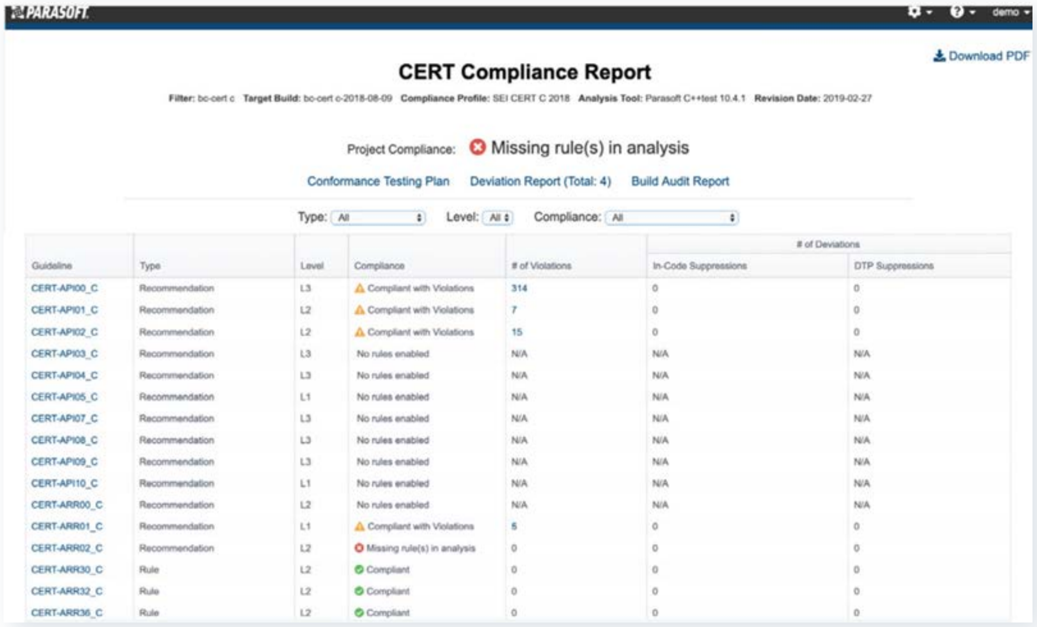


Figure 3-13: Parasoft DTP SEI CERT C Compliance dashboard



Download PDF

CERT Compliance Report

Filter: bc-cert c **Target Build:** bc-cert c-2018-08-09 **Compliance Profile:** SEI CERT C 2018 **Analysis Tool:** Parasoft C++test 10.4.1 **Revision Date:** 2019-02-27

Project Compliance: Missing rule(s) in analysis

Conformance Testing Plan   Deviation Report (Total: 4)   Build Audit Report

Type: All   Level: All   Compliance: All

					# of Deviations	
Guideline	Type	Level	Compliance	# of Violations	In-Code Suppressions	DTP Suppressions
CERT-API00_C	Recommendation	L3	⚠ Compliant with Violations	314	0	0
CERT-API01_C	Recommendation	L2	⚠ Compliant with Violations	7	0	0
CERT-API02_C	Recommendation	L2	⚠ Compliant with Violations	15	0	0
CERT-API03_C	Recommendation	L3	No rules enabled	N/A	N/A	N/A
CERT-API04_C	Recommendation	L3	No rules enabled	N/A	N/A	N/A
CERT-API05_C	Recommendation	L1	No rules enabled	N/A	N/A	N/A
CERT-API07_C	Recommendation	L3	No rules enabled	N/A	N/A	N/A
CERT-API08_C	Recommendation	L3	No rules enabled	N/A	N/A	N/A
CERT-API09_C	Recommendation	L3	No rules enabled	N/A	N/A	N/A
CERT-API10_C	Recommendation	L1	No rules enabled	N/A	N/A	N/A
CERT-ARR00_C	Recommendation	L2	No rules enabled	N/A	N/A	N/A
CERT-ARR01_C	Recommendation	L1	⚠ Compliant with Violations	5	0	0
CERT-ARR02_C	Recommendation	L2	✖ Missing rule(s) in analysis	0	0	0
CERT-ARR30_C	Rule	L2	✓ Compliant	0	0	0
CERT-ARR32_C	Rule	L2	✓ Compliant	0	0	0
CERT-ARR36_C	Rule	L2	✓ Compliant	0	0	0

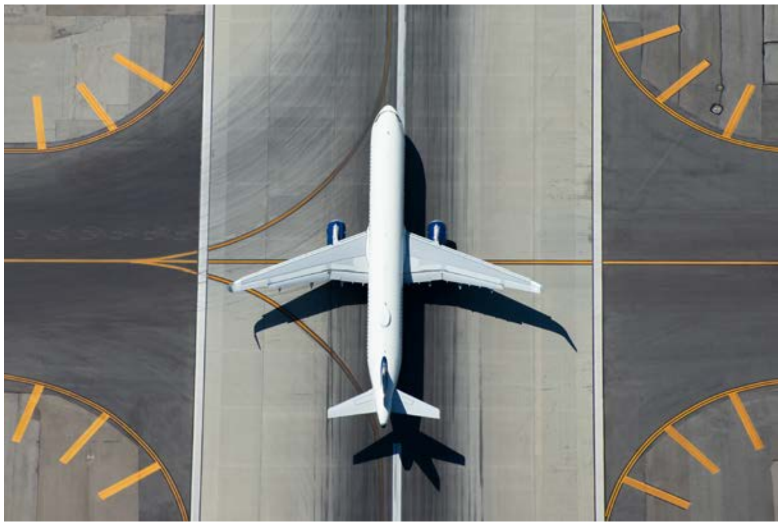
Figure 3-14: Parasoft's CERT Guidelines Compliance Report summary

Support for CERT C/C++ in Parasoft C/C++test

Parasoft provides comprehensive support for CERT C and CERT C++ secure coding standards with complete coverage of all the CERT C/C++ guidelines including rules and recommendations that are detectable by static analysis. Checker names, dashboards, and reports use the CERT naming convention to make conformance and auditing easier. A CERT conformance dashboard, which includes the CERT risk score, helps developers focus on the most critical violations.

CWE

CWE (Common Weakness Enumeration) is a list of discovered software weaknesses based on the analysis of reported vulnerabilities (CVEs). The collection of CVEs and CWEs is a U.S. government-funded initiative developed by the software community and managed by the MITRE organization. In its entirety, the CWE list contains over 900 different software and hardware quality and security issues.



These 900+ items are organized in more usable lists such as the well-known CWE Top 25. The Top 25 lists the most common and dangerous security weaknesses, which are all exploits that have a high chance of occurring and the impact of exploiting the weakness is large. The software weaknesses documented by a CWE are the software implicated in a set of discovered vulnerabilities (documented as CVEs) when analysis was performed to discover the root cause. CVEs are specific observed vulnerabilities in software products that have an exact definition of how to exploit them.

The current version of CWE Top 25 is from 2023. An updated Top 25 is currently in process with improved linking to CVEs and the NVD. Ranking considers realworld information so that it truly represents the Top 25 application security issues today. As soon as it is released, Parasoft will have updated support for the latest version.

Rank	ID	Name	Rank Change vs. 2022
1	CWE-787	Out-of-bounds Write	0
2	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	0
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	0
4	CWE-416	Use After Free	3
5	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	1
6	CWE-20	Improper Input Validation	-2
7	CWE-125	Out-of-bounds Read	-2
8	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	0
9	CWE-352	Cross-Site Request Forgery (CSRF)	0
10	CWE-434	Unrestricted Upload of File with Dangerous Type	0
11	CWE-862	Missing Authorization	5
12	CWE-476	NULL Pointer Dereference	-1
13	CWE-287	Improper Authentication	1
14	CWE-190	Integer Overflow or Wraparound	-1
15	CWE-502	Deserialization of Untrusted Data	-3
16	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	1
17	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	2
18	CWE-798	Use of Hard-coded Credentials	-3
19	CWE-918	Server-Side Request Forgery (SSRF)	2
20	CWE-306	Missing Authentication for Critical Function	-2
21	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	1
22	CWE-269	Improper Privilege Management	7
23	CWE-94	Improper Control of Generation of Code ('Code Injection')	2
24	CWE-863	Incorrect Authorization	4
25	CWE-276	Incorrect Default Permissions	-5

Figure 3-15: The 2023 CWE Top 25

For software teams that have a good handle on the Top 25, there's another grouping of the next most common and impactful vulnerabilities called the CWE CUSP. Another way to think of these are the top 25 honorable mentions.

The CWE uses a risk scoring method to rank the Top 25 and on the CUSP. This score takes into consideration the technical impact of a software weakness (how dangerous an exploit of the weakness is in the real world) as measured by the CWSS (Common Weakness Scoring System). Examples of technical impacts from vulnerabilities may include:

- Denial of service (DoS)
- Distributed denial of service (DDoS)
- Read or write access to protected information
- Unauthorized access and more

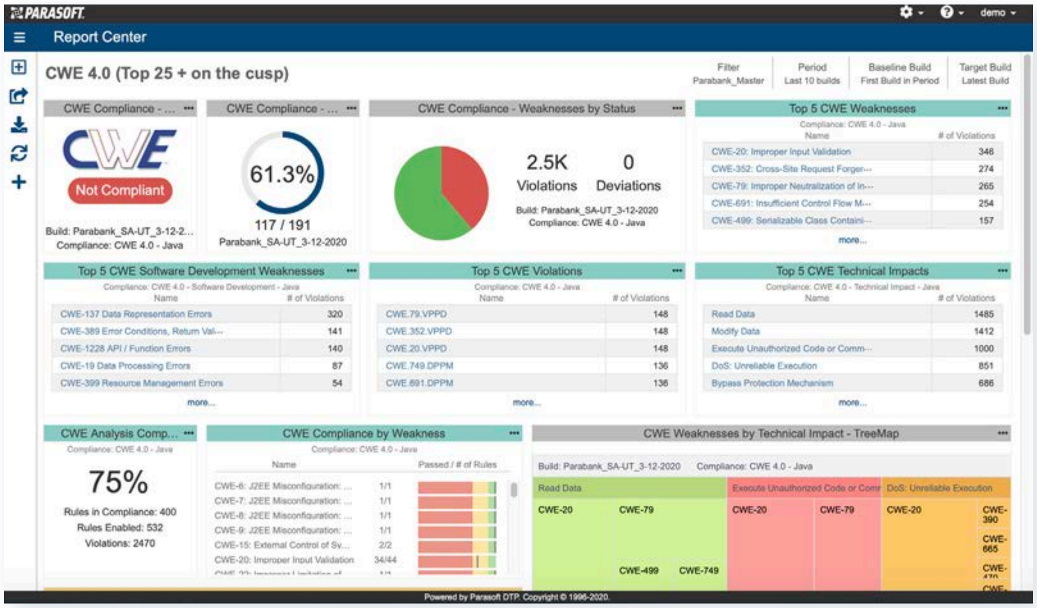
The details of these methods aren't too important, but the sorted list is useful in understanding which vulnerabilities to be concerned about the most. As an example, it's possible that your application is purely internal and DoS issues aren't critical for you. Being able to prioritize on the most important weaknesses for your own application can help overcome overwhelm with static analysis violations.

CWE Top 25 and On the Cusp Compliance

Introducing the coding standard compliance process into the team development workflow isn't an easy task. As such, it's important to select a tool that will help in achieving compliance without imposing too much overhead and without the requirement for additional manual procedures. The following points are important decision-making factors when selecting the solution for static analysis.

The CWE Top 25 and its lesser known sibling, On the Cusp, are not coding standards per se but a list of weaknesses to avoid to improve security. To be CWE compliant, a project should be able to prove that it has made reasonable efforts to detect and avoid these common weaknesses.

Parasoft's advanced static analysis tools for C, C++, Java, and .NET are officially compatible with CWE, providing automated detection of both Top 25 and On the Cusp weaknesses and many more. CWE-centric dashboards give users quick access to standard violations and current project status. A built-in CWE Top 25 configuration is available for C, C++, .NET, and Java with full coverage of all the 25 common weaknesses.


PARASOFT
Report Center
demo
Filter Parabank_Master
Period Last 10 builds
Baseline Build First Build in Period
Target Build Latest Build
CWE Compliance - ...
CWE
Not Compliant
61.3%
CWE Compliance - Weaknesses by Status
2.5K Violations
0 Deviations
CWE Weaknesses by Technical Impact - TreeMap
Read Data
Execute Unauthorized Code or Comr
DoS: Unreliable Execution
CWE-20
CWE-79
CWE-499
CWE-749
CWE-390
CWE-665
more...
Powered by Parasoft DTP. Copyright © 1996-2020.

CWE 4.0 (Top 25 + on the cusp)

Build: Parabank_SA-UT_3-12-2...
Compliance: CWE 4.0 - Java

117 / 191
Parabank_SA-UT_3-12-2020

Build: Parabank_SA-UT_3-12-2020
Compliance: CWE 4.0 - Java

Top 5 CWE Weaknesses
Compliance: CWE 4.0 - Java

Name	# of Violations
CWE-20: Improper Input Validation	346
CWE-352: Cross-Site Request Forger---	274
CWE-79: Improper Neutralization of In---	265
CWE-691: Insufficient Control Flow M---	254
CWE-499: Serializable Class Containi---	157

Top 5 CWE Software Development Weaknesses
Compliance: CWE 4.0 - Software Development - Java

Name	# of Violations
CWE-137 Data Representation Errors	320
CWE-389 Error Conditions, Return Val---	141
CWE-1228 API / Function Errors	140
CWE-19 Data Processing Errors	87
CWE-399 Resource Management Errors	54

Top 5 CWE Violations
Compliance: CWE 4.0 - Java

Name	# of Violations
CWE.79.VPPD	148
CWE.352.VPPD	148
CWE.20.VPPD	148
CWE.749.DPPM	136
CWE.891.DPPM	136

Top 5 CWE Technical Impacts
Compliance: CWE 4.0 - Technical Impact - Java

Name	# of Violations
Read Data	1485
Modify Data	1412
Execute Unauthorized Code or Comm---	1000
DoS: Unreliable Execution	851
Bypass Protection Mechanism	686

CWE Analysis Comp...
Compliance: CWE 4.0 - Java
75%
Rules in Compliance: 400
Rules Enabled: 532
Violations: 2470

CWE Compliance by Weakness
Compliance: CWE 4.0 - Java

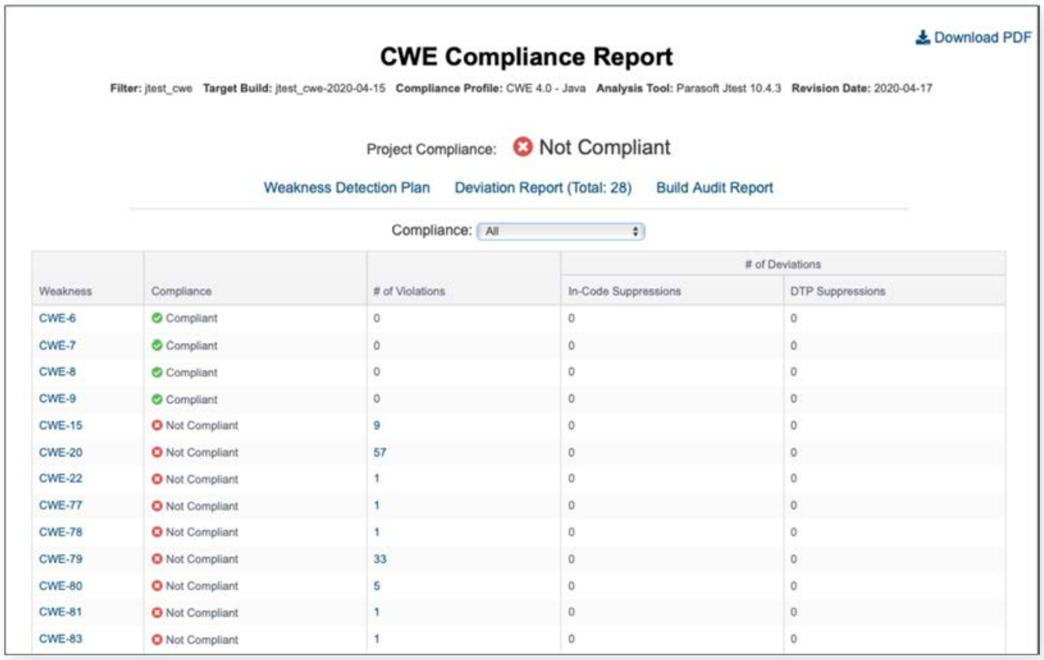
Name	Passed / # of Rules
CWE-6: J2EE Misconfiguration: ...	1/1
CWE-7: J2EE Misconfiguration: ...	1/1
CWE-8: J2EE Misconfiguration: ...	1/1
CWE-9: J2EE Misconfiguration: ...	1/1
CWE-15: External Control of Sy...	2/2
CWE-20: Improper Input Validation	34/44

Build: Parabank_SA-UT_3-12-2020 Compliance: CWE 4.0 - Java

Figure 3-16: Parasoft DTP CWE Compliance dashboard

The Parasoft tools include information from the Common Weakness Risk Analysis Framework (CWRAF), such as technical impact, so you can benefit from the same type of prioritization based on risk and technical impact and weaknesses found in your own code.

Parasoft also supports detailed compliance reporting to streamline audit processes. The web dashboards provide the link to compliance reports for a complete picture of where a project stands. In addition, the CWE Weakness Detection Plan maps the CWE entry against the checkers that are used to detect the weakness. This helps illustrate how compliance was achieved to an auditor, and the audit reports are available to download as PDFs for easy reporting.



Download PDF

CWE Compliance Report

Filter: jtest_cwe **Target Build:** jtest_cwe-2020-04-15 **Compliance Profile:** CWE 4.0 - Java **Analysis Tool:** Parasoft Jtest 10.4.3 **Revision Date:** 2020-04-17

Project Compliance: Not Compliant

Weakness Detection Plan Deviation Report (Total: 28) Build Audit Report

Compliance: All

Weakness	Compliance	# of Violations	# of Deviations	
			In-Code Suppressions	DTP Suppressions
CWE-6	✓ Compliant	0	0	0
CWE-7	✓ Compliant	0	0	0
CWE-8	✓ Compliant	0	0	0
CWE-9	✓ Compliant	0	0	0
CWE-15	✗ Not Compliant	9	0	0
CWE-20	✗ Not Compliant	57	0	0
CWE-22	✗ Not Compliant	1	0	0
CWE-77	✗ Not Compliant	1	0	0
CWE-78	✗ Not Compliant	1	0	0
CWE-79	✗ Not Compliant	33	0	0
CWE-80	✗ Not Compliant	5	0	0
CWE-81	✗ Not Compliant	1	0	0
CWE-83	✗ Not Compliant	1	0	0

Figure 3-17: Parasoft's CWE Guidelines Compliance Report summary

Unit Testing

Software verification is inherently part of safety-critical software development. Testing, by way of execution, is a key way to demonstrate the implementation of requirements and delivery of quality software. Unit testing is the verification of low-level requirements. It ensures that each software unit does what it's required to do within its expected quality of service requirements—safety, security, and reliability.

Safety and security requirements instruct that software units don't behave in unforeseen ways where the system is not susceptible to hijacking, data manipulation, theft, or corruption.

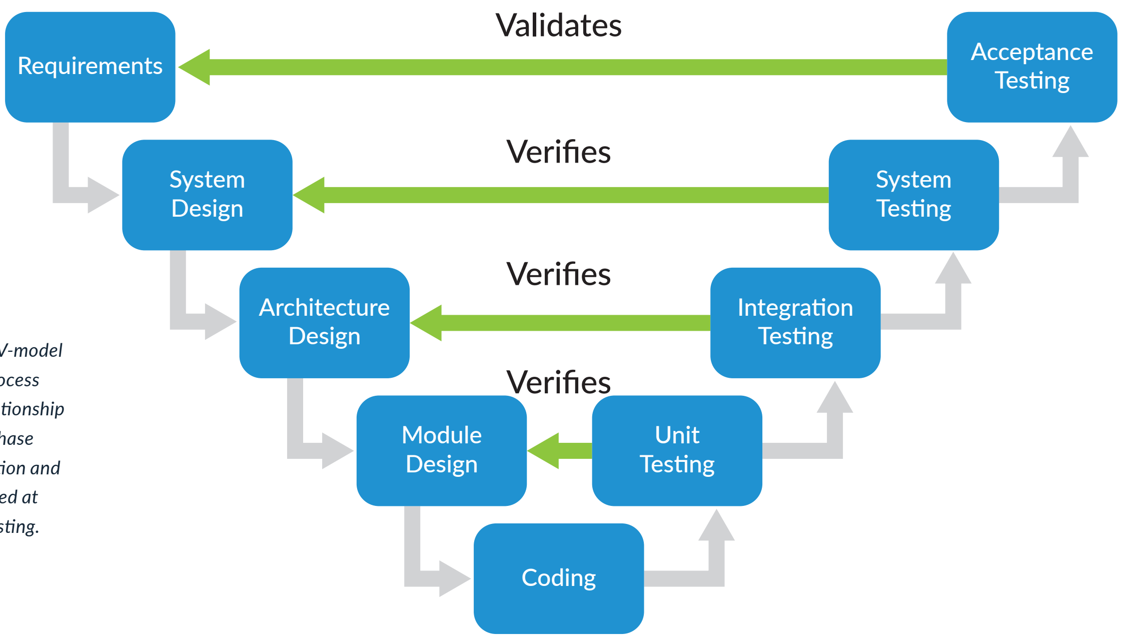


Figure 4-1: The V-model development process showing the relationship between each phase and the verification and validation inferred at each stage of testing.

In terms of the classic V-model process of development, unit test execution is a verification practice to ensure the module is designed correctly. DO-178C does not specifically mandate unit testing by name, but rather uses the terms high- and low-level requirements-based testing.

Low-level testing is commonly understood to be unit testing. In particular, the requirements for this type of requirements-based testing include the following.

- **Software Testing, Section 6.4.** Outlines the software validation process, which includes various testing activities such as software requirements-based testing, low-level requirements testing, and high-level requirements-based testing. Unit testing is typically considered a part of low-level requirements testing, Section 6.4.3 c, where individual software units like functions, procedures, or methods are tested in isolation from the rest of the system.

  DO-178C lists the following as typical errors that unit testing reveals.

  - Failure of an algorithm to satisfy a software requirement
  - Incorrect loop operations
  - Incorrect logic decisions
  - Failure to process correctly legitimate combinations of input conditions
  - Incorrect responses to missing or corrupted input data
  - Incorrect handling of exceptions, such as arithmetic faults or violations of array limits
  - Incorrect computation sequence
  - Inadequate algorithm precision, accuracy, or performance
- **Software Verification and Case and Procedures, Section 11.13.** Details the requirements for verification cases and procedures, which include the test cases used for various testing activities, including unit testing.
- **Software Verification Results, Section 11.14.** Covers the documentation and recording of verification results, which include the results of unit testing activities.

DO-178C does not prescribe specific testing methodologies or tools but does emphasize the need for thorough testing to ensure the safety, security, and reliability of airborne software. Tests must be performed at all levels of the system along with is traceability between requirements, design, source code, and tests. In addition, test plans, test cases and results must be documented for certification.

Unit Test Methods

Requirement-Based Tests

These tests directly test functionality and quality of service as specified in each requirement. Test automation tools need to support bidirectional traceability of requirements to their tests and the requirements testing coverage reports to show compliance.

High-level requirements are derived from top-level system requirements. They decompose a system requirement into various high-level functional and nonfunctional requirements. This phase of the requirements decomposition helps in the architectural design of the system under development.

High-level requirements clarify and help define expected behavior as well as safety tolerances, security expectations, reliability, performance, portability, availability, scalability, and more. Each high-level requirement links up to the system requirement that it satisfies. In addition, high-level test cases are created and linked to each high-level requirement for the purpose of its verification and validation. This software requirements analysis process continues as each high-level requirement is further decomposed into low-level requirements.

Low-level requirements are software requirements derived from high-level requirements. They further decompose and refine the specification of the software's behavior and quality of service.

These requirements drill down to another level of abstraction. They map to individual software units and are written in a way that facilitates software detail design and implementation. Traceability is established from each low-level requirement up to its high-level requirement and down to the low-level tests or unit test cases that verify and validate it.

Unit testing becomes about isolating the function, method, or procedure. It’s done by stubbing and mocking out dependencies and forcing specific paths of execution. Stubs take the place of the code in the unit that's dependent on code outside of the unit. They also provide the developer or tester with the ability to manipulate the response or result so that the unit can be exercised in various ways and for various purposes, for example, to ensure that the unit performs reliably, is safe, and is also free from security vulnerabilities.

Interface Tests

Interface tests ensure programming interfaces behave and perform as specified. Test tools need to create function stubs and data sources to emulate behavior of external components for automatic unit test execution.

Fault Injection Tests

Fault injection tests use unexpected inputs and introduce failures in the execution of code to examine failure handling or lack thereof. Test automation tools must support injection of fault conditions using function stubs and automatic unit test generation using a diverse set of preconditions, such as min, mid, max, and heuristic value testing.

Resource Usage Evaluation

These tests evaluate the amount of memory, file space, CPU execution, or other target hardware resources used by the application.

Test Case Drivers

Analysis of Requirements

Clearly, every requirement drives, at minimum, a single unit test case. Although test automation tools don't generate tests directly from requirements, they must support two-way traceability from requirements to code and requirements to tests, and maintain requirements, tests, and code coverage information.

Generation & Analysis of Equivalence Classes

Test cases must ensure that units behave in the same manner for a range of inputs, not just cherry-picked inputs for each unit. Test automation tools must support test case generation using data sources to efficiently use a wide range of input values. Parasoft C/C++test uses factory functions to prepare sets of input parameter values for automated unit test generation.

Analysis of Boundary Values

Automatically generated test cases, like heuristic values and boundary values, employ data sources to use a wide range of input values in tests.

Error Guessing

The error guessing method uses the function stubs mechanism to inject fault conditions into tested code flow analysis results and can be used to write additional tests.

Automated Test Execution & Test Case Generation

Test automation provides large benefits to safety-critical embedded device software. Moving away from test suites that require a lot of manual intervention means that testing can be done quicker, easier, and more often.

Offloading this manual testing effort frees up time for better test coverage and other safety and quality objectives. An important requirement for automated test suite execution is being able to run these tests on both host and target environments.

Target-Based Testing

Automating testing of embedded software is more challenging due to the complexity of initiating and observing tests on embedded targets, not to mention the limited access to target hardware that software teams have.

DO-178C requires testing software in a representative environment that reflects the actual deployment conditions. This includes testing on the target hardware or using a software environment that closely resembles the final target environment. This approach is required to ensure that the software operates correctly and reliably in the actual aircraft or airborne system.

Software test automation is essential to make embedded testing workable on a continuous basis from host development system to target system. Testing embedded software is particularly time consuming. Automating the regression test suite provides considerable time and cost savings. In addition, C/C++test CT and C/C++test perform code coverage data collection from the target system, which is essential for validation and standards compliance.

Traceability between test cases, test results, source code, and requirements must be recorded and maintained. For those reasons, data collection is critical in test execution.

Parasoft C/C++test is offered with its test harness optimized to take minimal additional overhead for the binary footprint and provides it in the form of source code, where it can be customized if platform-specific modifications are required.

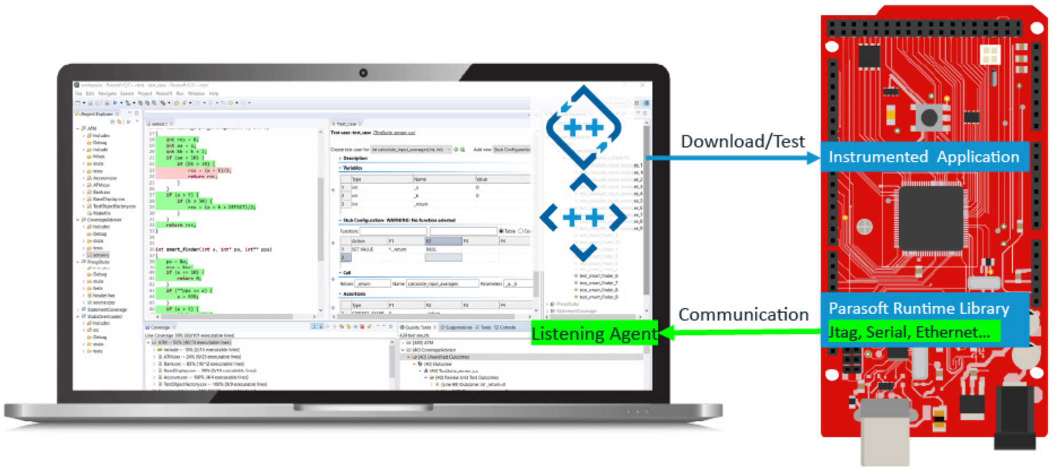


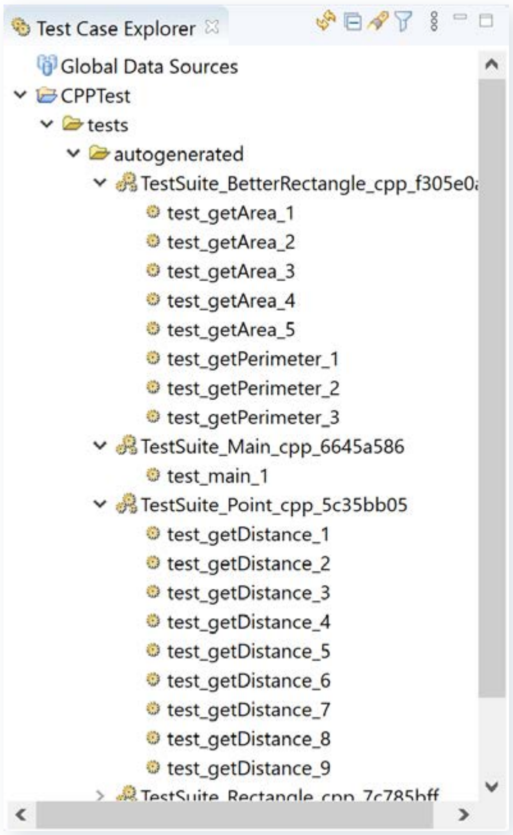
Figure 4-2: A high-level view of deploying, executing, and observing tests from host to target in Parasoft C/C++ testing solutions.

One huge benefit that the Parasoft C/C++test solution offers is its dedicated integrations with embedded IDEs and debuggers that make the process of executing test cases smooth and automated. Supported IDE environments include:

- VS Code
- Eclipse
- Green Hills Multi
- Wind River Workbench
- IAR EW
- ARM MDK
- ARM DS-5
- TI CCS
- Visual Studio
- Many more

Automated Test Case Generation

Unit test automation tools universally support some sort of test framework, which provides the harness infrastructure to execute units in isolation while satisfying dependencies via stubs. Parasoft C/C++test is no exception. Part of its unit test capability is the automated generation of test harnesses and the executable components needed for host and target-based testing.


Parasoft C/C++ automated test case generation, in this case, one test suite per function

Test Case Explorer

```
  Global Data Sources
v CPPTest
  v tests
    v autogenerated
      v TestSuite_BetterRectangle_cpp_f305e0
          test_getArea_1
          test_getArea_2
          test_getArea_3
          test_getArea_4
          test_getArea_5
          test_getPerimeter_1
          test_getPerimeter_2
          test_getPerimeter_3
      v TestSuite_Main_cpp_6645a586
          test_main_1
      v TestSuite_Point_cpp_5c35bb05
          test_getDistance_1
          test_getDistance_2
          test_getDistance_3
          test_getDistance_4
          test_getDistance_5
          test_getDistance_6
          test_getDistance_7
          test_getDistance_8
          test_getDistance_9
      > TestSuite_Rectangle_cpp_7c785bff
```

Test data generation and management is by far the biggest challenge in unit testing. Test cases are particularly important in safety-critical software development because they must ensure functional requirements and test for unpredictable behavior, security, and safety requirements. All while satisfying test coverage criteria.

Parasoft C/C++test automatically generates test cases like the popular CppUnit format. By default, C/C++test generates one test suite per source/header file. It can also be configured to generate one test suite per function or one test suite per source file.

Safe stub definitions are automatically generated to replace "dangerous" functions, which include system I/O routines such as rmdir(), remove(), rename(), and so on. In addition, stubs can be automatically generated for missing function and variable definitions. User defined stubs can be added as needed.

Regression Testing

As part of most software development processes, regression testing is done after changes are made to software. These tests determine if the new changes had an impact on the existing operation of the software.

DO-178C doesn't explicitly mention regression testing, but it is a good engineering practice and is widely employed in the aerospace industry to verify the stability and correctness of the software throughout its development lifecycle. Requirements around the software and hardware integration process imply the need to maintain up-to-date verification and validation after any changes.

» **Requirements-based Hardware/Software Integration Testing, Section 6.4.3.** Integration testing is a level of testing in DO-178C that verifies the interactions between different software units. When changes are made to software components or units, regression testing is necessary to verify that the modifications have not adversely affected the integrated system.

» **Integration Process, Section 5.4.** Focuses on the integration of software components and emphasizes that the integration process should be planned and controlled. Integrating new or modified software units requires regression testing to ensure that the system's overall behavior remains correct and that no unintended side effects have been introduced.

» **Software Verification Results, Section 11.14.** Covers the documentation and recording of verification results, including the results of testing activities. If regression testing is performed, the results should be documented to demonstrate that the changes did not negatively impact the system.

Regression tests are necessary, but they only indicate that recent code changes have not caused tests to fail. There's no assurance that these changes will work. In addition, the nature of the changes that motivate the need to do regression testing can go beyond the current application and include changes in hardware, operating system, and operating environment.

Software Regression Testing in Airborne Systems

In safety-critical software development, validation is critical in proving correct functionality, safety, and security. Tests are needed for two primary reasons.

1. Confirm any changes to the application to ensure functionality.
2. Verify that there aren't any unforeseen impacts on the rest of the system.

If a test case previously passed but now fails, a potential regression has been identified. The failure could be caused by new functionality, in which the test case may need to be updated so that it takes into consideration changes in input and output values.

Regression testing of embedded systems also includes the execution of the following types of test cases:

- Unit
- Integration
- System
- Performance
- Stress and more

In fact, all previously created test cases need to be executed to ensure that no regressions exist and that a new dependable software version release is constructed. This is critical because each new software system or subsystem release is built upon it. If you don't have a solid foundation the whole thing can collapse.

Parasoft C/C++test supports the creation of regression testing baselines as an organized collection of tests and automatically verifies all outcomes. These tests are run automatically on a regular basis to verify whether code modifications change or break the functionality captured in the regression tests. If any changes are introduced, these test cases will fail to alert the team to the problem. During subsequent tests, C/C++test will report tasks if it detects changes to the behavior captured in the initial test.

How to Decide What to Regression Test

The key challenge with regression testing is determining what parts of an application to test. It is common to default to executing all regression tests when there's doubt on what impacts recent code changes have had—the all or nothing approach.

For large software projects, this becomes a huge undertaking and drags down the productivity of the team. This inability to focus testing hinders much of the benefits of iterative and continuous processes, potentially exacerbated in embedded software where test targets are a limited resource.

A couple of tasks are required here.

» Identify which tests need to be re-executed.

» Focus the testing efforts (unit testing, automated functional testing, and manual testing) on validating the features and related code that are impacted by the most recent changes.

Developers and testers can get a clear understanding of the changes in the codebase between builds using the Process Intelligence Engine (PIE) within Parasoft DTP (Development Testing Platform) combined with Parasoft's proprietary coverage analysis engines:

» C/C++test for C and C++

» dotTEST for C#

» Jtest for Java

With this combination, teams can improve efficiency and achieve the promise of Agile.

This form of smart test execution is called test impact analysis. It's sometimes referred to as change-based testing.

Understand the Impact of Code Changes on Testing With Test Impact Analysis

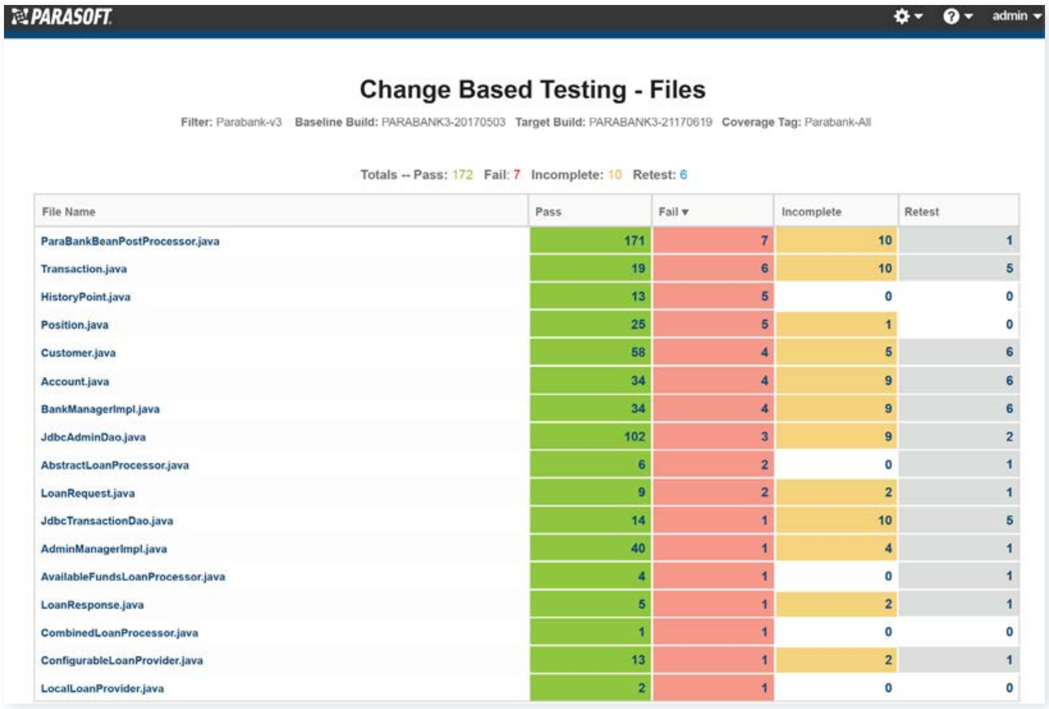
Test impact analysis uses data collected during test runs and changes in code between builds to determine which files have changed and which specific tests touched those files. Parasoft's analysis engine can:

» Analyze the delta between two builds.

» Identify the subset of regression tests that need to be executed.

» Understand the dependencies on the units modified to determine what ripple effect the changes have made on other units.

Parasoft Jtest and dotTEST provide insight into the impact of software changes. Each solution recommends where to add tests and where further regression testing is needed.



Change Based Testing - Files

Filter: Parabank-v3 **Baseline Build:** PARABANK3-20170503 **Target Build:** PARABANK3-21170619 **Coverage Tag:** Parabank-All

Totals -- Pass: 172 Fail: 7 Incomplete: 10 Retest: 6

File Name	Pass	Fail ▾	Incomplete	Retest
ParaBankBeanPostProcessor.java	171	7	10	1
Transaction.java	19	6	10	5
HistoryPoint.java	13	5	0	0
Position.java	25	5	1	0
Customer.java	58	4	5	6
Account.java	34	4	9	6
BankManagerImpl.java	34	4	9	6
JdbcAdminDao.java	102	3	9	2
AbstractLoanProcessor.java	6	2	0	1
LoanRequest.java	9	2	2	1
JdbcTransactionDao.java	14	1	10	5
AdminManagerImpl.java	40	1	4	1
AvailableFundsLoanProcessor.java	4	1	0	1
LoanResponse.java	5	1	2	1
CombinedLoanProcessor.java	1	1	0	0
ConfigurableLoanProvider.java	13	1	2	1
LocalLoanProvider.java	2	1	0	0

Figure 5-1: An example change-based testing report from Parasoft DTP showing areas of the code that are and are not tested.

Software Integration Testing

Integration testing follows unit testing with the goal of validating the architectural design. It ensures that higher level functional capabilities in software components, including subsystems and not units, behave and perform as expected. Testing software integrations can be done bottom up and top down with a combination of approaches in many software organizations.

Integration testing is a critical aspect of the software verification process in DO-178C. The explicit requirements for integration testing can be found primarily in Section 5.4 Integration Process and Section 6.4 Software Testing.

Section 6.4.3 Requirements-Based Testing Methods in DO-178C requires hardware and software requirements-based testing, which includes integration testing. Section 6.4.3 b is more specific and outlines requirements-based integration testing as a method that concentrates on the “inter-relationships between the software requirements" and on the "implementation of requirements by the software architecture.”

DO-178C lists the following typical errors revealed by integration testing.

- Incorrect interrupt handling.
- Failure to satisfy execution time requirements.
- Incorrect software response to hardware transients or hardware failures, for example, start-up sequencing, transient input loads, and input power transients.
- Data bus and other resource contention problems, for example, memory mapping.
- Inability of built-in test to detect failures.
- Errors in hardware/software interfaces.
- Incorrect behavior of control loops.
- Incorrect control of memory management hardware or other hardware devices under software control.
- Stack overflow.
- Incorrect operation of mechanism(s) used to confirm the correctness and compatibility of field-loadable software.
- Violations of software partitioning.
- Incorrect initialization of variables and constants.

- Parameter passing errors.
- Data corruption, especially global data.
- Inadequate end-to-end numerical resolution.
- Incorrect sequencing of events and operations.

Bottom-Up Integration

This approach begins by taking a unit test case and removing stubs and/or mocks to incorporate additional software units to construct higher-level functionality that can be tested. Functionality maps to or equates to a high-level requirement. Integration test cases are used to verify and validate high-level requirements.

Top-Down Integration

In this testing, the highest-level software components or modules are tested first. Progressively, testing of lower-level modules follows or functional capabilities map to high-level requirements. This approach assumes significant subsystems are complete enough to be tested as a whole.

The V-model is good for illustrating the relationship between the stages of development and stages of validation. At each testing stage, more complete portions of the software are validated against the phase that defines it.

For some, the V-model might imply a Waterfall development method. However, this is not the case. DO-178C and previous versions of the standard do not specify a development methodology. The V-model shows a required set of development phases. Organizations determine how to address those phases. Teams can adopt a Waterfall, Agile, Spiral, or any development methodology, and be compliant to the standard.

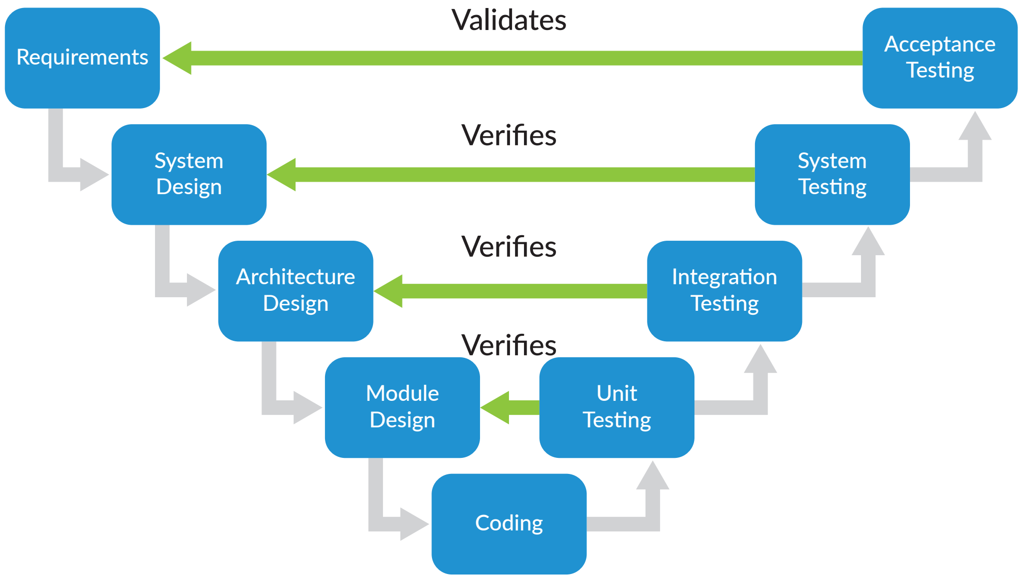


Figure 6-1: The V-model development process showing the relationship between each phase and the verification and validation inferred at each stage of testing.

While the act of executing tests and gathering their results is considered software validation, it's supported by a parallel verification process that involves the following activities to make sure teams are building the process and the product correctly.

- Reviews
- Walkthroughs
- Code analysis
- Traceability
- Test
- Code coverage and more

The key role of verification is to ensure that the building of delivered artifacts from the previous stage to specification is compliant with company and industry guidelines.

Integration & System Testing as Part of a Continuous Testing Process

Performing some level of test automation is foundational for continuous testing. Many organizations start by simply automating manual integration and system testing (top down) or unit testing (bottom up).

To enable continuous testing, organizations need to focus on creating a scalable test automation practice that builds on a foundation of unit tests that are isolated and faster to execute. Once unit testing is fully automated, the next step is integration testing and eventually system testing.

Continuous testing leverages automation and data derived from testing to provide a realtime, objective assessment of the risks associated with a system under development. Applied uniformly, it allows both business and technical managers to make better tradeoff decisions between release scope, time, and quality.

Continuous testing is a powerful testing methodology that ensures continuous code quality through the SDLC. It enforces compliance in static code analysis and is always identifying safety and security defects during each developer's commit action by also integrating unit, integration, and system testing in the loop.

The diagram below illustrates how different phases of testing are part of a continuous process that relies on a feedback loop of test results and analysis.

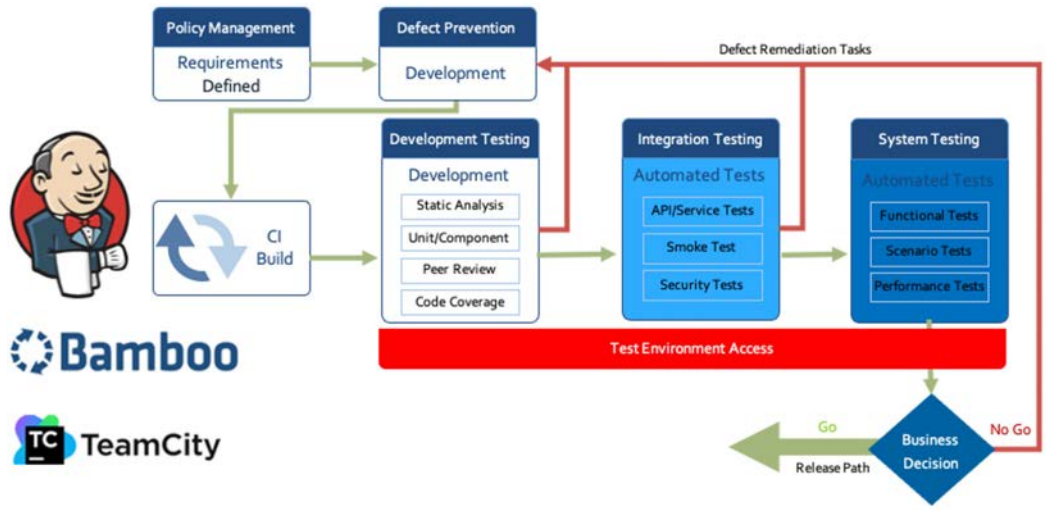


Figure 6-2: A continuous testing cycle

Analysis & Reporting in Support of Integration & System Testing

Parasoft test automation tools support the validation (actual execution testing activities) in terms of test automation and continuous testing. These tools also support the verification of these activities, which means supporting the process and standard requirements. Key aspects of safety-critical software development are requirements traceability and code coverage.

DO-178C considers traceability a key activity and artifact of the development process. Sections 5.4 Software Development Process and 6.4 Software Testing require bidirectional traceability between high-level and low-level requirements and the implementation, verification, and validation of assets, which include:

- Source code
- Requirement documents
- Test results
- Development plans and more

Requirements analysis requires “All software requirements should be identified in such a way as to make it possible to demonstrate traceability between the requirement and software system testing.” Providing a requirements traceability matrix helps satisfy this requirement.

Two-Way Traceability

Requirements in safety-critical software are the key driver for product design and development. These requirements include functional safety, application requirements, and nonfunctional requirements that fully define the product. This reliance on documented requirements is a mixed blessing because poor requirements are one of the critical causes of safety incidents in software. In other words, the implementation wasn't at fault, but poor or missing requirements were.

Automating Bidirectional Traceability

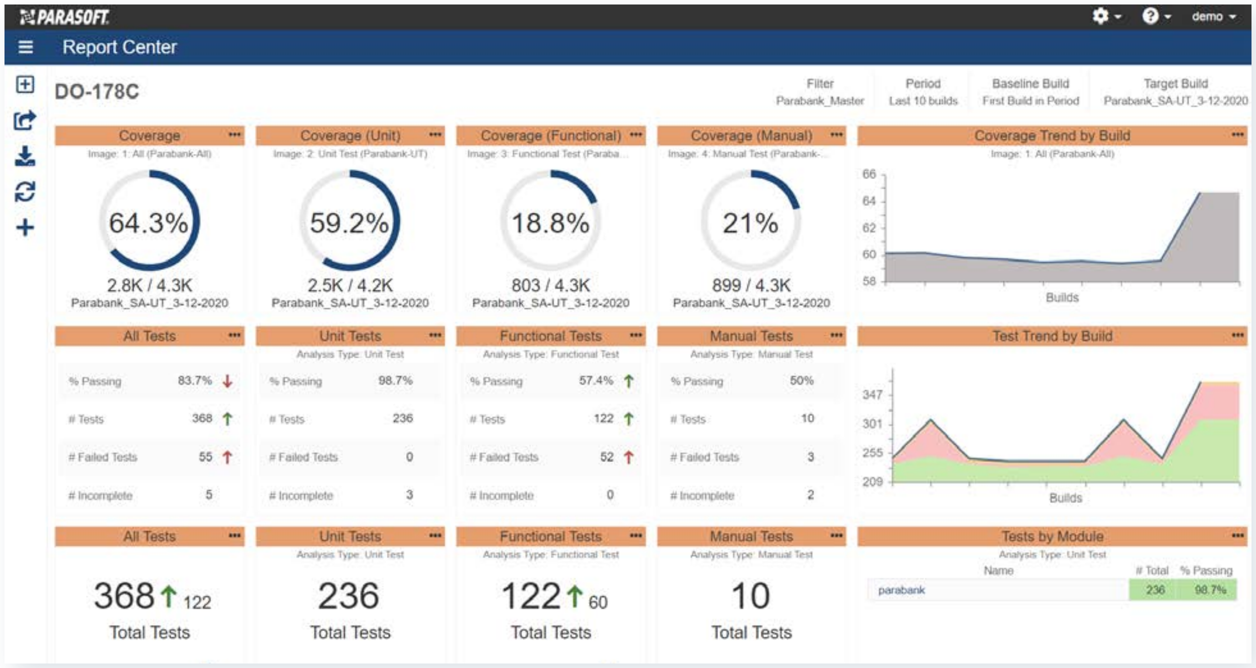
Maintaining traceability records on any sort of scale requires automation. Application life cycle management tools include requirements management capabilities that are mature and tend to be the hub for traceability.

Integrated software testing tools like Parasoft complete the verification and validation of requirements by providing an automated bidirectional traceability to the executable test case. This includes the pass or fail result and traces down to the source code that implements the requirement.

Parasoft integrates with market leading requirements management tools or ALM systems including:

- IBM DOORS Next
- PTC Codebeamer
- Siemens Polarion
- Atlassian Jira
- Jama Connect and more

As shown in the image below, each of Parasoft's test automation solutions (C/C++test, C/C++test CT, Jtest, dotTEST, SOAtest, and Selenic) used within the development life cycle supports the association of tests with work items defined in these systems, such as requirements, defects, and test cases or test runs. The central reporting and analytics dashboard, Parasoft DTP, manages traceability.



Filter	Period	Baseline Build	Target Build
Parabank_Master	Last 10 builds	First Build in Period	Parabank_SA-UT_3-12-2020

Coverage	Coverage (Unit)	Coverage (Functional)	Coverage (Manual)
Image: 1: All (Parabank-All)	Image: 2: Unit Test (Parabank-UT)	Image: 3: Functional Test (Paraba...	Image: 4: Manual Test (Parabank-...
64.3%	59.2%	18.8%	21%
2.8K / 4.3K	2.5K / 4.2K	803 / 4.3K	899 / 4.3K
Parabank_SA-UT_3-12-2020	Parabank_SA-UT_3-12-2020	Parabank_SA-UT_3-12-2020	Parabank_SA-UT_3-12-2020

Coverage Trend by Build
Image: 1: All (Parabank-All)
66 64 62 60 58

	All Tests	Unit Tests	Functional Tests	Manual Tests
		Analysis Type: Unit Test	Analysis Type: Functional Test	Analysis Type: Manual Test
% Passing	83.7% ↓	98.7%	57.4% ↑	50%
# Tests	368 ↑	236	122 ↑	10
# Failed Tests	55 ↑	0	52 ↑	3
# Incomplete	5	3	0	2

Test Trend by Build
347 301 255 209

All Tests	Unit Tests	Functional Tests	Manual Tests
	Analysis Type: Unit Test	Analysis Type: Functional Test	Analysis Type: Manual Test
368 ↑ 122	236	122 ↑ 60	10
Total Tests	Total Tests	Total Tests	Total Tests

Tests by Module
Analysis Type: Unit Test

Name	# Total	% Passing
parabank	236	98.7%

Figure 6-3: An example of a DO-178C reporting dashboard that captures the project's testing status and progress towards completion.

Figure 6-4: Codebeamer traceability matrix, which lists system requirements from high level to low level along with test cases and test results.

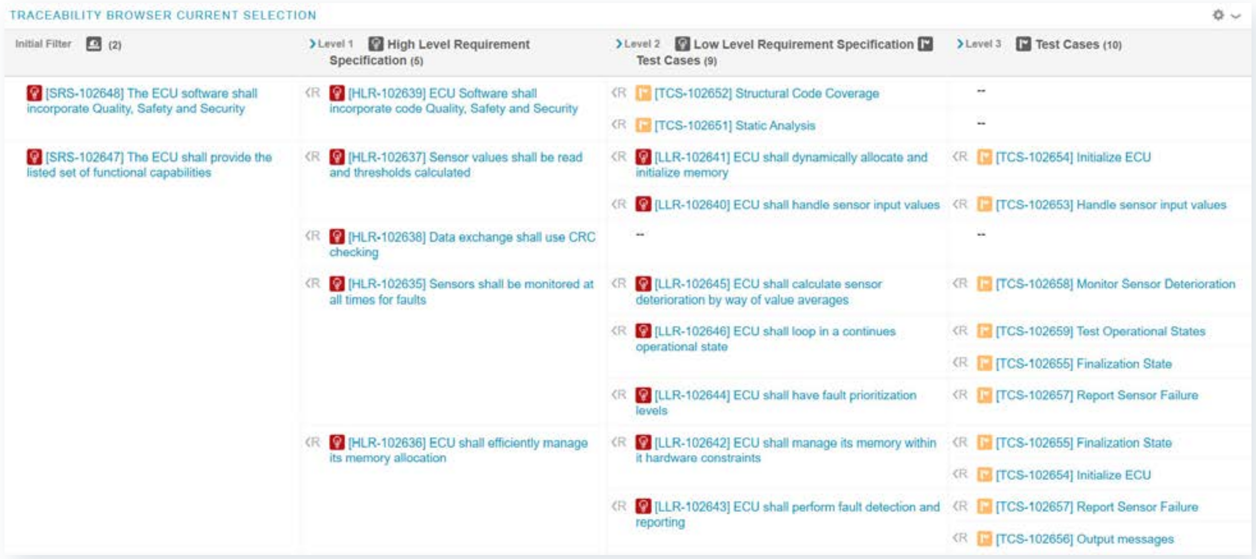
Parasoft DTP correlates the unique identifiers from the management system with:

» Static analysis findings

» Code coverage

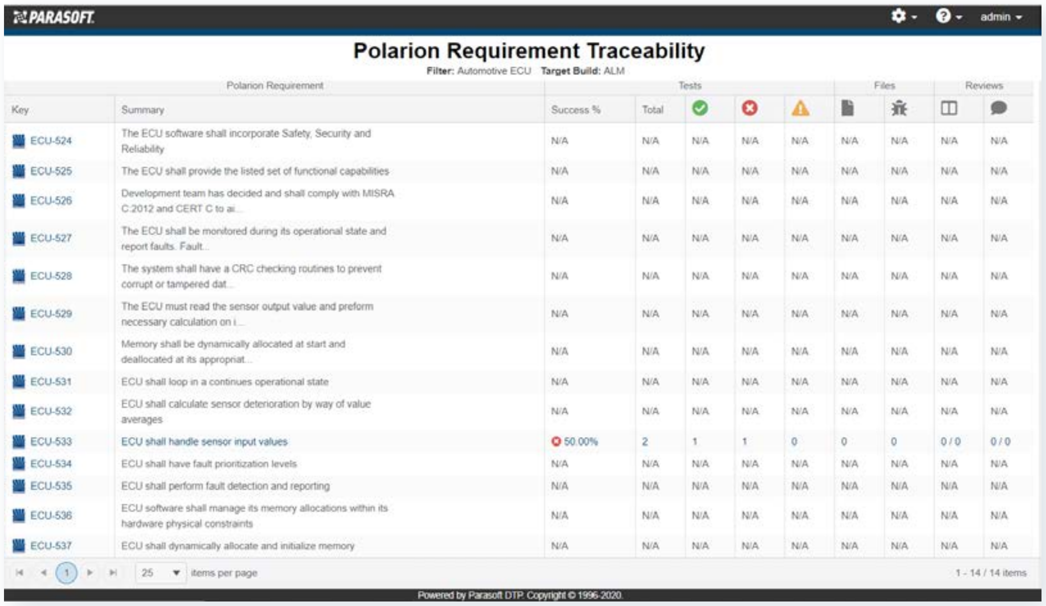
» Results from unit, integration, and functional tests

Results are displayed within Parasoft DTP's traceability reports and sent back to the requirements management system. They provide full bidirectional traceability and reporting as part of the system's traceability matrix.


TRACEABILITY BROWSER CURRENT SELECTION

Initial Filter (2)	Level 1 High Level Requirement Specification (5)	Level 2 Low Level Requirement Specification Test Cases (9)	Level 3 Test Cases (10)
[SRS-102648] The ECU software shall incorporate Quality, Safety and Security	[HLR-102639] ECU Software shall incorporate code Quality, Safety and Security	[TCS-102652] Structural Code Coverage	--
		[TCS-102651] Static Analysis	--
[SRS-102647] The ECU shall provide the listed set of functional capabilities	[HLR-102637] Sensor values shall be read and thresholds calculated	[LLR-102641] ECU shall dynamically allocate and initialize memory	[TCS-102654] Initialize ECU
		[LLR-102640] ECU shall handle sensor input values	[TCS-102653] Handle sensor input values
	[HLR-102638] Data exchange shall use CRC checking	--	--
	[HLR-102635] Sensors shall be monitored at all times for faults	[LLR-102645] ECU shall calculate sensor deterioration by way of value averages	[TCS-102658] Monitor Sensor Deterioration
		[LLR-102646] ECU shall loop in a continues operational state	[TCS-102659] Test Operational States
			[TCS-102655] Finalization State
		[LLR-102644] ECU shall have fault prioritization levels	[TCS-102657] Report Sensor Failure
	[HLR-102636] ECU shall efficiently manage its memory allocation	[LLR-102642] ECU shall manage its memory within it hardware constraints	[TCS-102655] Finalization State
			[TCS-102654] Initialize ECU
		[LLR-102643] ECU shall perform fault detection and reporting	[TCS-102657] Report Sensor Failure
			[TCS-102656] Output messages

The traceability reporting in Parasoft DTP is highly customizable. The following image shows a requirements traceability matrix template for requirements authored in Polarion and traces to the test cases, static analysis findings, the source code files, and the manual code reviews.



Polarion Requirement Traceability

Filter: Automotive ECU **Target Build:** ALM

Polarion Requirement		Tests					Files		Reviews	
Key	Summary	Success %	Total	✓	✗	⚠	📄	🐞	◫	💬
ECU-524	The ECU software shall incorporate Safety, Security and Reliability	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-525	The ECU shall provide the listed set of functional capabilities	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-526	Development team has decided and shall comply with MISRA C:2012 and CERT C to ai...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-527	The ECU shall be monitored during its operational state and report faults. Fault...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-528	The system shall have a CRC checking routines to prevent corrupt or tampered dat...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-529	The ECU must read the sensor output value and preform necessary calculation on i...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-530	Memory shall be dynamically allocated at start and deallocated at its appropriat...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-531	ECU shall loop in a continues operational state	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-532	ECU shall calculate sensor deterioration by way of value averages	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-533	ECU shall handle sensor input values	✗ 50.00%	2	1	1	0	0	0	0 / 0	0 / 0
ECU-534	ECU shall have fault prioritization levels	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-535	ECU shall perform fault detection and reporting	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-536	ECU software shall manage its memory allocations within its hardware physical constraints	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-537	ECU shall dynamically allocate and initialize memory	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

1 | 25 items per page | 1 - 14 / 14 items

Powered by Parasoft DTP. Copyright © 1996-2020.

Figure 6-5: Requirements traceability matrix template from Parasoft DTP integrated with Siemens Polarion.

The bidirectional correlation between test results and work items provides the basis of requirements traceability. Parasoft DTP adds test and code coverage analysis to evaluate test completeness. Maintaining this bidirectional correlation between requirements, tests, and the artifacts that implement them is an essential component of traceability.

Code Coverage

Code coverage expresses the degree to which the application's source code is exercised by all testing practices, including unit, integration, and system testing—both automated and manual.

Collecting coverage data throughout the life cycle enables more accurate quality and coverage metrics, while exposing untested or under tested parts of the application.

As with traceability, code coverage is a key metric in airborne systems development. DO-178C has specific requirements in Section 6.4.4 Test Coverage Analysis. These requirements extend beyond code coverage and include the test coverage of all high-level and low-level requirements, along with the test coverage of the entire software structure.

Section 6.4.4.2 Structural Code Analysis requires the test coverage of source code beyond what may already be covered with requirements-based testing. This ensures that all code is executed by tests before certification. This code coverage analysis may reveal issues such as missing tests and dead or deactivated code. Section 6.4.4.3 Structural Coverage Analysis Resolution requires the remediation of these discrepancies discovered during coverage analysis.

Application coverage can also help organizations focus testing efforts when time constraints limit their ability to run the full suite of manual regression tests. Capturing coverage data on the running system on its target hardware during integration and system testing completes code coverage from unit testing.

Benefits of Aggregate Code Coverage

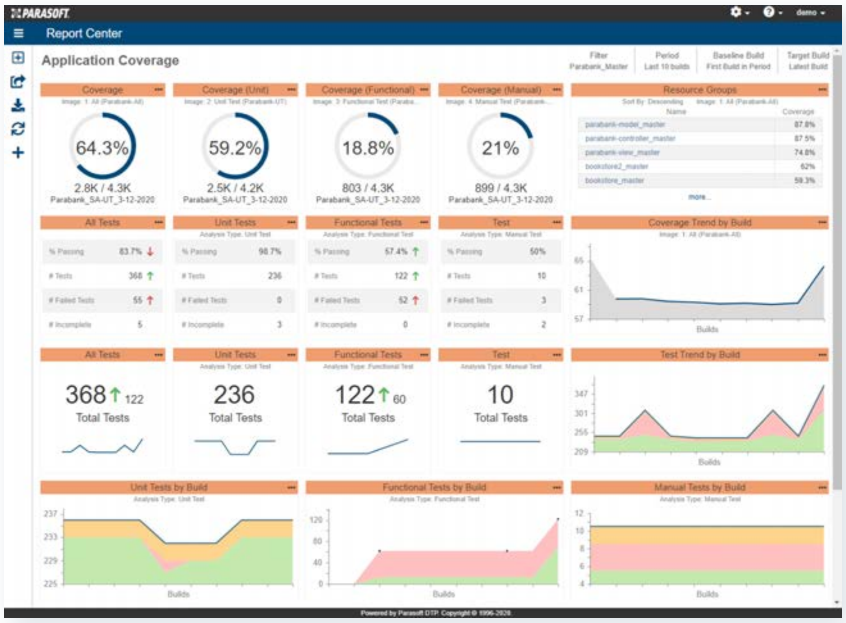
Captured coverage data is leveraged as part of the continuous integration (CI) process as well as the tester’s workflow. Parasoft DTP performs advanced analytics on code coverage from all tests, source code changes, static analysis results, and test results. The results help identify untested and undertested code and other high risk areas in the software.

Analyzing code, executing tests, tracking coverage, and reporting the data in a dashboard or chart is a useful first step toward assessing risk, but teams must still dedicate significant time and resources to reading the tea leaves and hope that they’ve interpreted the data correctly.

Understanding the potential risks in the application requires advanced analytics processes that merge and correlate the data. This provides greater visibility into the true code coverage and helps identify testing gaps and overlapping tests. For example, what's the true coverage for the application under test when your tools report different coverage values for unit tests, automated functional tests, and manual tests?

The percentages cannot simply be added together because the tests overlap. This is a critical step for understanding the level of risk associated with the application under development.

Figure 6-6: Aggregated code coverage from various testing methods



Name	Coverage
parabank-model_master	87.8%
parabank-controller_master	87.5%
parabank-view_master	74.8%
bookstore2_master	62%
bookstore_master	58.3%

	All Tests	Unit Tests	Functional Tests	Test
% Passing	83.7%	98.7%	57.4%	50%
# Tests	368	236	122	10
# Failed Tests	55	0	52	3
# Incomplete	5	3	0	2

Understanding the Impact of Code Changes on Testing With Test Impact Analysis

Test impact analysis uses data collected during test runs and changes in code between builds to determine which files have changed and which specific tests touched those files. Parasoft's analysis engine can analyze the delta between two builds and identify the subset of regression tests that need to be executed. It also understands the dependencies on the units modified to determine the ripple effect the changes have made on other units.

Parasoft Jtest and dotTEST provide insight into the impact of software changes and recommend where to add tests and where further regression testing is needed.

Accelerating Integration & System Testing With Test Automation Tools

Parasoft's software test automation tools accelerate verification by automating the many tedious aspects of record keeping, documentation, reporting, analysis, and reporting.

» **Two-way traceability for all artifacts** ensures requirements have code and tests to prove they are being fulfilled. Metrics, test results, and static analysis results are traced to components and vice versa.

» **Code and test coverage** verifies all requirements are implemented and makes sure the implementation is tested as required.

» **Target and host-based test execution** supports different validation techniques as required.

» **Smart test execution** manages change with a focus on tests for only code that changed and any impacted dependents.

» **Reporting and analytics** provides insight to make important decisions and keeps track of progress. Decision making needs to be based on data collected from the automated processes.

» **Automated documentation generation** from analytics and test results support process and standards compliance.

» **Standards compliance automation** reduces the overhead and complexity by automating the most repetitive and tedious processes. The tools can keep track of the project history and relating results against requirements, software components, tests, and recorded deviations.

Software System Testing

System testing tests the system as a whole. Once all the components are integrated, the entire system is tested rigorously to verify that it meets the specified functional, safety, security, and other nonfunctional requirements.

DO-178C specifies both software and hardware/software integration testing. In terms of the software development aspect of airborne systems, this aligns with the concept of "system testing" for the purposes here. There are many more aspects of system and flight testing of airborne systems that aren't covered here.

Section 6.4.3 a Requirements-Based Hardware/Software Integration Testing focuses on the operation of the software on the target hardware environment. The aim is to validate high-level requirements. It's also important to point out that nonfunctional requirements must be tested, and Section 6.4.2.1 requires normal range tests to demonstrate normal operation of the software alongside Section 6.4.2.2, which requires robustness test cases. These are tests that use abnormal data ranges that fall outside expected values for inputs to demonstrate the system can handle them without failure.

This type of testing in safety-critical software is performed by a specialized testing team. System testing falls within the scope of black box testing. As such, it shouldn't require any knowledge of the inner design of the code or logic.

An important distinction with system level testing is that the system is tested in an environment that is close to the production environment where the application will be deployed. At this stage, specific safety functions are verified, and system wide security testing is run.

Service Level Testing of Airborne Systems

Airborne systems may have connectivity into larger systems that, as an example, collect and analyze status and flight data. Any sort of communication bus or network must be tested for data integrity, security, and confidentiality. System testing needs to include these environments for complete validation.

Instead of viewing system quality in terms of meeting individual component requirements, the scope is broadened to consider the quality of the services provided. Testing at the service level ensures nonfunctional requirements are met. For example, performance and reliability are difficult to assess at the device level or during software unit testing. Service based testing can simulate the operational environment of a device to provide realistic loads.

Security is a growing concern in airborne systems. Cyberattacks are possible in modern systems and likely originate from the network itself by attacking the exposed APIs. Service based testing can create simulated environments for robust security testing, either through fuzzing (random and erroneous data inputs) or denial-of-service attacks.

Virtual Test Environment & Service Level Testing

A real test lab requires the closest physical manifestation of the environment in which a system is planned to work. Even in the most sophisticated lab, it's difficult to scale to a realistic environment. A virtual lab fixes this problem.

Virtual labs evolve past the need for hard-to-find (or nonexistent) hardware dependencies. They use sophisticated service virtualization with other key test automation tools.

Service Virtualization

Service virtualization simulates all of the dependencies needed by the device under test in order to perform full system testing. This includes all connections and protocols used by the device with realistic responses to communication. For example, service virtualization can simulate an enterprise server backend with which a system under test communicates. Similarly, virtualization can control and simulate a dependent system, like patient information, in a realistic manner.

Service & API Testing

This testing drives the system under test in a manner that ensures the services and APIs it provides perform flawlessly. These tests can be manipulated via the automation platform to run performance and security tests as needed.

Runtime Monitoring

This detects errors in realtime on the system under test and captures important trace information.

Test Lab Management & Analytics

Once virtualized, an entire lab setup can be replicated as needed, providing overarching control of the virtual labs. Test runs can be automated and repeated. Analytics provide the necessary summary of activities and outcomes.

Parasoft SOAtest & Virtualize for Service Level Testing of Airborne Software

Developers can build integrations earlier, stabilize dependencies, and gain full control of their test data with Parasoft Virtualize. Teams can move forward quickly without waiting for access to dependent services that are either incomplete or unavailable. Companies can enable partners to test against their applications with a dedicated sandbox environment.

Parasoft SOAtest delivers fully integrated API and web service testing tools that automate end-to-end functional API testing. Teams can streamline automated testing with advanced functional test creation capabilities for applications with multiple interfaces and protocols.

SOAtest and Virtualize are well suited for network-based, system-level testing of various types, including the following:

» Comprehensive protocol stack that supports HTTP, MQTT, RabbitMQ, JMS, XML, JSON, REST, SOAP, and more.

» Security and performance testing during integration and system testing with integration into the existing CI/CD process.

» End-to-end testing that combines API, web, mobile, and database interactions into virtual test environments.

Structural Code Coverage

Collecting and analyzing code coverage metrics is an important aspect of safety-critical software development. Code coverage measures the completion of test cases and executed tests. It provides evidence that verification is complete, at least as specified by the software design. The objectives for test coverage analysis include achieving the following test coverage targets:

- High-level requirements
- Low-level requirements
- Software structure to the appropriate coverage criteria
- Software structure, both data coupling and control coupling

DO-178C Section 6.4.4.1 covers requirements test coverage analysis, which determines how well functional testing has verified the implementation of the requirements. It is expected that code coverage analysis is collected during this testing and the remaining gaps in code coverage are closed with further testing.

Section 6.4.4.2 requires analysis to determine what remains of code coverage, including interfaces between components. Section 6.4.4.3 outlines the requirements to resolve any of the gaps in coverage, including the identification of extraneous, dead, and deactivated code.

How this translates to types and amounts of coverage is somewhat open to interpretation. However, in airborne software development, the onus is on the manufacturer to plan for code coverage, adhere to the plan, document, and complete it.

Types of Code Coverage

Following are the different types of code coverage.

- **Statement coverage** requires that each program statement be executed at least once. Branch and MC/DC coverage encompass statement coverage.
- **Branch coverage** ensures that each decision branch (if-then-else constructs) is executed.

» **Modified condition/decision coverage (MC/DC)** requires the most complete code coverage to ensure test cases execute each decision branch and all the possible combinations of inputs that affect the outcome of decision logic. For complex logic, the number of test cases can explode, so the modified condition restrictions are used to limit test cases to those that result in standalone logical expressions changing.

» **Executable/object code** is required if the software level criteria is at A. This is due to the fact that a compiler or linker generates additional assembly code that is not directly traceable to source code statements. Therefore, object level coverage must be performed.

Advanced unit test automation tools, such as Parasoft C/C++test, provide all these code coverage metrics and more. C/C++test CT also automates this data collection on host and target testing and accumulates test coverage history over time. This code coverage history can span unit, integration, and system testing to ensure coverage is complete and traceable at all levels of testing.

Coverage From System Testing

Obtaining code coverage through system testing is an excellent method to determine if enough testing has been performed. The approach is to run all your system tests, and then examine what parts of the code have not been exercised.

The unexecuted code implies that there may be need for new test cases to exercise the untouched code where a defect may be lurking and helps answer the question: Have I done enough testing?

When teams perform system testing, the average resulting metric is 60% coverage. Much of the 40% unexecuted code is due to defensive code in your application. Defensive code only executes upon the system triggering a fault or entering a problematic state that may be difficult to produce. Conditions like memory leakage or other types of faults caused by hardware failure may take weeks, months, or years to encounter.

There's also defensive code mandated by your coding guidelines where system test cases can never get you to execute. For these reasons, system testing cannot take you to 100% structural code coverage. You'll need to employ other testing methods like manual and/or unit testing to reach 100%.

Coverage From Unit Testing

As mentioned, unit testing can be used as a complementary approach to system testing to obtain 100% coverage. Obtaining code coverage through unit testing is one of the more popular methods used, but it doesn't expose whether you have done enough testing of the system because the focus is at the unit level (function/procedure).

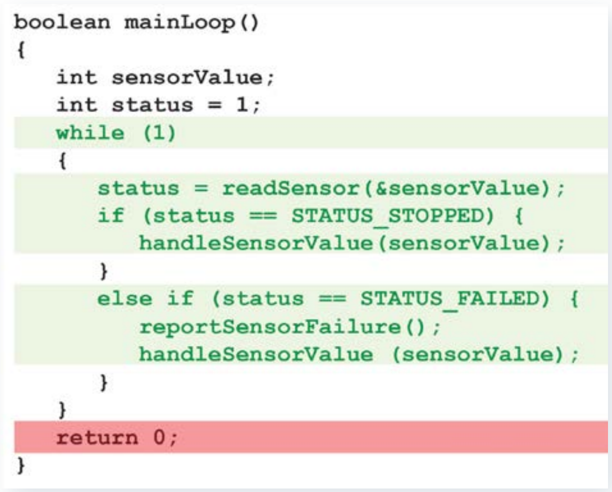
The goal here is to create a set of unit test cases that exercise the entire unit at the required coverage need (statement, branch, and MC/DC) in order to reach 100% coverage for that single unit. This is repeated for every unit until the entire code base is covered. However, to get the most out of unit testing, do not solely focus on obtaining code coverage. That can generally be accomplished through sunny day scenario test cases.

Truly exercise the unit through sunny and rainy-day scenarios to ensure robustness, safety, security, and low-level requirements traceability. Let code coverage be a biproduct of your test cases and fill in coverage where needed.

To help expedite code coverage through unit testing, configurable and automated test case generation capabilities exist in Parasoft C/C++test. Test cases can be automatically generated to test for use of null pointers, min-mid-max ranges, boundary values, and much more. This automation can get you far. In minutes, you'll obtain a substantial amount of code coverage.

Additionally, C/C++test CT extends development workflows with code coverage by integrating with proprietary unit testing frameworks and IDEs. Tightly integrate code coverage line, statement, simple condition, decision, branch, function, call, and MC/DC with proprietary unit testing frameworks like GoogleTest and CppUnit and IDEs like VS Code.

However, as in system testing, obtaining 100% code coverage is elusive due to the use of defensive code or formal language semantics. At the granular level of a unit, defensive code may come in the form of a *default* statement in a *switch*. If every possible case in a *switch* is captured, this leaves the *default* statement unreachable. In the example below, the *return 0;* will never get executed because the *while (1)* is infinite.



```
boolean mainLoop()
{
   int sensorValue;
   int status = 1;
   while (1)
   {
      status = readSensor(&sensorValue);
      if (status == STATUS_STOPPED) {
         handleSensorValue(sensorValue);
      }
      else if (status == STATUS_FAILED) {
         reportSensorFailure();
         handleSensorValue (sensorValue);
      }
   }
   return 0;
}
```

Figure 8-1: Unreachable return 0; Statement

How does one obtain 100% coverage for these special cases?

Answer: Deploying manual methods.

Follow these steps.

1. Label or notate the statement as covered by using a debugger.
2. Modify the call stack and execute the *return 0;* statement.
3. Visually witness the execution and, at minimum, document the file name, line of code, and code statement that is now considered covered.

This coverage performed through manual/visual inspection and reports can be used to supplement the coverage captured through unit testing. The addition of both coverage reports can be used to prove 100% structural code coverage.

The goal of obtaining code coverage is an added means to help ensure code safety, security, and reliability.

Code Instrumentation

Code coverage is more often than not identified by having the code instrumented. Instrumented refers to having the user code adorned with additional code to ascertain during execution if that statement, branch, or MC/CD has been executed.

Based on the target or system under test, the coverage data can be stored in the file system, written to memory, or sent out through various communication channels, such as the serial port, TCP/IP port, USB, and even JTAG.

Partial Instrumentation

Be aware that code instrumentation causes code bloat. The increase in code size may impact the ability to load the code onto memory-constrained target hardware for testing.

The workaround is to instrument part of the code by following these steps:

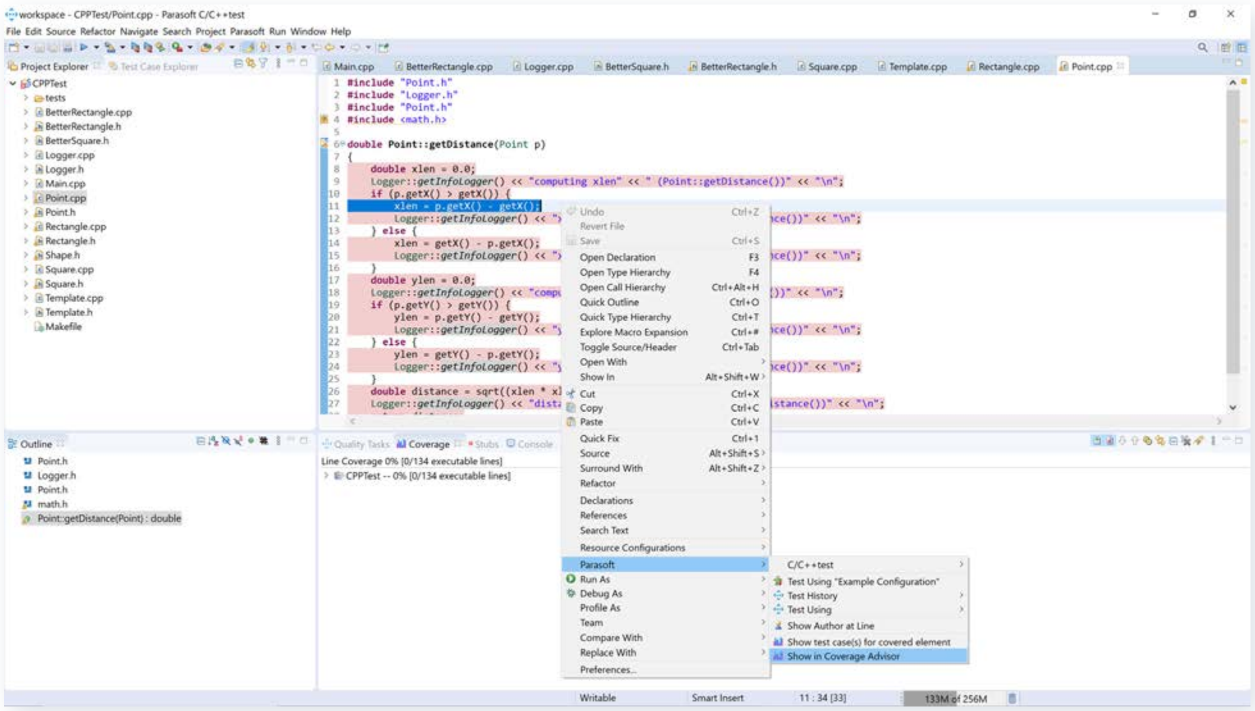
1. Run your tests and capture the coverage.
2. Instrument the other part of the code.
3. Run your tests again.
4. Capture the coverage.
5. Merge the coverage from the previous test execution.

Coverage Advisor

Parasoft C/C++test resolves coverage gaps in test suites. Parasoft discovered how to use advanced static code analysis (data and control flow analysis) to find values for the input parameters required to execute specific lines of uncovered code.

In complex code, there are always those elusive code statements for which it is exceedingly difficult to obtain coverage. It's likely there are multiple input values with various permutations and possible paths that make it mind twisting and time consuming to decipher. But only one combination can get you the coverage you need. Parasoft makes it easy to obtain coverage of those difficult to reach lines of code.

When you select the line of code you want to cover, the Coverage Advisor will tell you what input values, global variables, and external calls you need to stimulate the code and obtain coverage.



```
#include "Point.h"
#include "Logger.h"
#include "Point.h"
#include <math.h>

double Point::getDistance(Point p)
{
    double xlen = 0.0;
    Logger::getInfoLogger() << "computing xlen" << " (Point::getDistance())" << "\n";
    if (p.getX() > getX()) {
        xlen = p.getX() - getX();
        Logger::getInfoLogger() << "x                                   ce())" << "\n";
    } else {
        xlen = getX() - p.getX();
        Logger::getInfoLogger() << "x                                   ce())" << "\n";
    }
    double ylen = 0.0;
    Logger::getInfoLogger() << "compu                                 ))" << "\n";
    if (p.getY() > getY()) {
        ylen = p.getY() - getY();
        Logger::getInfoLogger() << "y                                   ce())" << "\n";
    } else {
        ylen = getY() - p.getY();
        Logger::getInfoLogger() << "y                                   ce())" << "\n";
    }
    double distance = sqrt((xlen * xl
    Logger::getInfoLogger() << "dista                               istance())" << "\n";
```

```
Line Coverage 0% [0/134 executable lines]
> CPPTest -- 0% [0/134 executable lines]
```

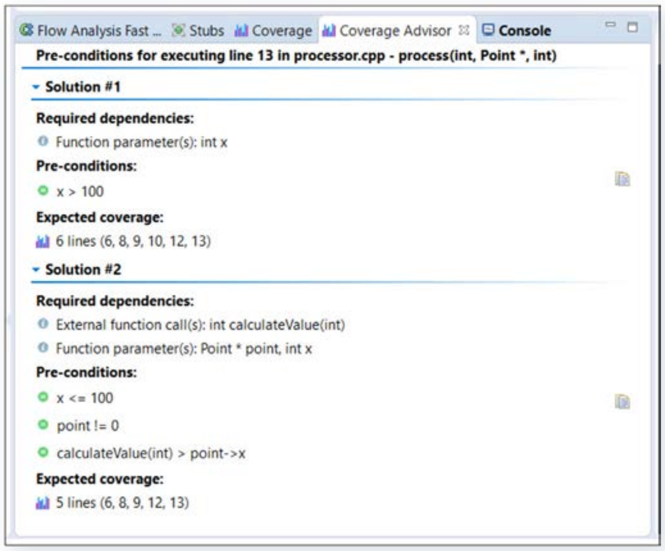
Figure 8-2: Invoking Coverage Advisor by right-clicking on the line of code.

The figure below shows an analysis report providing the user with a solution. The Preconditions field expresses:

» The range and input values for `mainSensorSignal` and `coSensorSignal`

» The expected outputs from the external calls

Upon creating the unit test case with these set parameter values and stubs for external calls, you get coverage of the line selected, plus the additional lines expressed in the Expected Coverage field.



Pre-conditions for executing line 13 in processor.cpp - process(int, Point *, int)

Solution #1

Required dependencies:
- Function parameter(s): int x

Pre-conditions:
- x > 100

Expected coverage:
- 6 lines (6, 8, 9, 10, 12, 13)

Solution #2

Required dependencies:
- External function call(s): int calculateValue(int)
- Function parameter(s): Point * point, int x

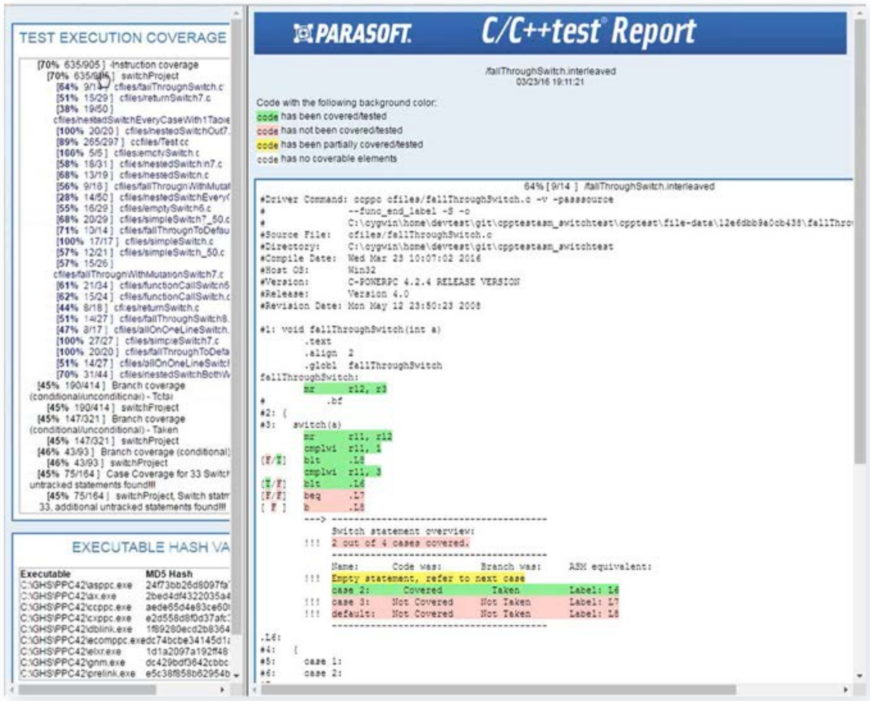
Pre-conditions:
- x <= 100
- point != 0
- calculateValue(int) > point->x

Expected coverage:
- 5 lines (6, 8, 9, 12, 13)

Figure 8-3: Two test case solutions provided by Coverage Advisor.

Object Code Coverage

For the most stringent safety-critical applications, DO-178C Level A, Object Code Coverage is required. Therefore, assembly level coverage must be performed. Imagine the rigor and labor cost of having to perform this task. Fortunately, Parasoft ASMTools provides an automated solution for obtaining object code coverage.



TEST EXECUTION COVERAGE

```
[70%  635/905 ]  Instruction coverage
  [70%  635/905 ]  switchProject
    [64%  9/14 ]  cfiles/fallThroughSwitch.c
    [51%  15/29 ]  cfiles/returnSwitch7.c
    [38%  19/50 ]
   cfiles/nestedSwitchEveryCaseWith1Table
    [100%  20/20 ]  cfiles/nestedSwitchOut7.
    [89%  265/297 ]  ccfiles/Test.cc
    [100%  5/5 ]  cfiles/emptySwitch.c
    [58%  18/31 ]  cfiles/nestedSwitchIn7.c
    [68%  13/19 ]  cfiles/nestedSwitch.c
    [56%  9/16 ]  cfiles/fallThroughWithMutat
    [28%  14/50 ]  cfiles/nestedSwitchEvery(
    [55%  16/29 ]  cfiles/emptySwitch6.c
    [68%  20/29 ]  cfiles/simpleSwitch7_50.c
    [71%  10/14 ]  cfiles/fallThroughToDefau
    [100%  17/17 ]  cfiles/simpleSwitch.c
    [57%  12/21 ]  cfiles/simpleSwitch_50.c
    [57%  15/26 ]
   cfiles/fallThroughWithMutationSwitch7.c
    [61%  21/34 ]  cfiles/functionCallSwitch6
    [62%  15/24 ]  cfiles/functionCallSwitch.c
    [44%  8/18 ]  cfiles/returnSwitch.c
    [51%  14/27 ]  cfiles/fallThroughSwitch8.
    [47%  8/17 ]  cfiles/allOnOneLineSwitch.
    [100%  27/27 ]  cfiles/simpleSwitch7.c
    [100%  20/20 ]  cfiles/fallThroughToDefa
    [51%  14/27 ]  cfiles/allOnOneLineSwitcl
    [70%  31/44 ]  cfiles/nestedSwitchBothW
  [45%  190/414 ]  Branch coverage
(conditional/unconditional) - Total
    [45%  190/414 ]  switchProject
  [45%  147/321 ]  Branch coverage
(conditional/unconditional) - Taken
    [45%  147/321 ]  switchProject
  [46%  43/93 ]  Branch coverage (conditional)
    [46%  43/93 ]  switchProject
  [45%  75/164 ]  Case Coverage for 33 Switch
untracked statements found!!!
    [45%  75/164 ]  switchProject, Switch statm
   33, additional untracked statements found!!!
```

EXECUTABLE HASH VA

Executable	MD5 Hash
C:\GHS\PPC42\asppc.exe	24f73bb26d8097fa
C:\GHS\PPC42\ax.exe	2bed4df4322035a4
C:\GHS\PPC42\ccppc.exe	aede65d4e83ce60
C:\GHS\PPC42\cxppc.exe	e2d558d8f0d37afc
C:\GHS\PPC42\dblink.exe	1f89280ecd2b8364
C:\GHS\PPC42\ecomppc.exe	dc74bcbe34145d1
C:\GHS\PPC42\elxr.exe	1d1a2097a192ff48
C:\GHS\PPC42\gnm.exe	dc429bdf3642cbbc
C:\GHS\PPC42\prelink.exe	e5c38f858b62954b

/fallThroughSwitch.interleaved
03/23/16 19:11:21

Code with the following background color:
🟩 code has been covered/tested
🟥 code has not been covered/tested
🟨 code has been partially covered/tested
code has no coverable elements

64% [9/14] /fallThroughSwitch.interleaved

```
#Driver Command: ccppc cfiles/fallThroughSwitch.c -v -passsource
#                --func_end_label -S -o
#                C:\cygwin\home\devtest\git\cpptestasm_switchtest\cpptest\file-data\12e6dbb9a0cb438\fallThro
#Source File:    cfiles/fallThroughSwitch.c
#Directory:      C:\cygwin\home\devtest\git\cpptestasm_switchtest
#Compile Date:   Wed Mar 23 10:07:02 2016
#Host OS:        Win32
#Version:        C-POWERPC 4.2.4 RELEASE VERSION
#Release:        Version 4.0
#Revision Date:  Mon May 12 23:50:23 2008

#1: void fallThroughSwitch(int a)
        .text
        .align  2
        .globl  fallThroughSwitch
fallThroughSwitch:
        mr      r12, r3
#           .bf
#2: {
#3:    switch(a)
        mr      r11, r12
        cmplwi  r11, 1
[F/T]   blt     .L8
        cmplwi  r11, 3
[T/F]   blt     .L6
[F/F]   beq     .L7
[ F ]   b       .L8
        ---> ------------------------------------
             Switch statement overview:
        !!!  2 out of 4 cases covered.
             ------------------------------------
             Name:      Code was:      Branch was:     ASM equivalent:
        !!!  Empty statement, refer to next case
             case 2:      Covered        Taken          Label: L6
        !!!  case 3:    Not Covered    Not Taken        Label: L7
        !!!  default:   Not Covered    Not Taken        Label: L8
             ------------------------------------
.L6:
#4:    {
#5:      case 1:
#6:      case 2:
```

Figure 8-4: Parasoft ASMTool for Assembly/Object Code Coverage

Requirements & the Traceability Matrix

In airborne systems, requirements management is a mandatory part of the software development process and the traceability of those requirements to implementation. Subsequently, teams must ensure proof of correct implementation.

Requirements traceability is defined as "the ability to describe and follow the life of a requirement in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases)."

The objectives of traceability are to ensure the following:

» Functional, performance, and safety-related requirements of the system that are allocated to software were developed into the high-level requirements.

» High-level requirements and derived requirements were developed into the low-level requirements.

» Low-level requirements were developed into source code.

» Traceability between requirements and test cases, test procedures, and test results.

In the simplest sense, requirements traceability keeps track of each requirement's decomposition into software and the tests used to verify and validate each requirement. It also tracks exactly what you're building when writing software. This means making sure the software does what it's supposed to and that you're only building what's needed.

If there are architectural elements or source code that can't be traced to a requirement, then it's a risk and shouldn't be there. The benefits also go beyond providing proof of the implementation. Tracking each requirement's analysis and decomposition is commonly used for visibility into development progress.

Requirements analysis requires that "All software requirements should be identified in such a way as to make it possible to demonstrate traceability between the requirement and software system testing."

It's important to realize that many requirements in safety-critical software are derived from safety analysis and risk management. The system must perform its intended functions, of course, but it must also mitigate risks to greatly reduce the possibility of injury. Moreover, in order to document and prove that these safety functions are implemented and tested fully and correctly, traceability is critical.

Tracing requirements isn't simply linking a paragraph from a document to a section of code or a test. Traceability must be maintained throughout the phases of development as requirements manifest into design, architecture, and implementation. Consider the typical V-model of software.

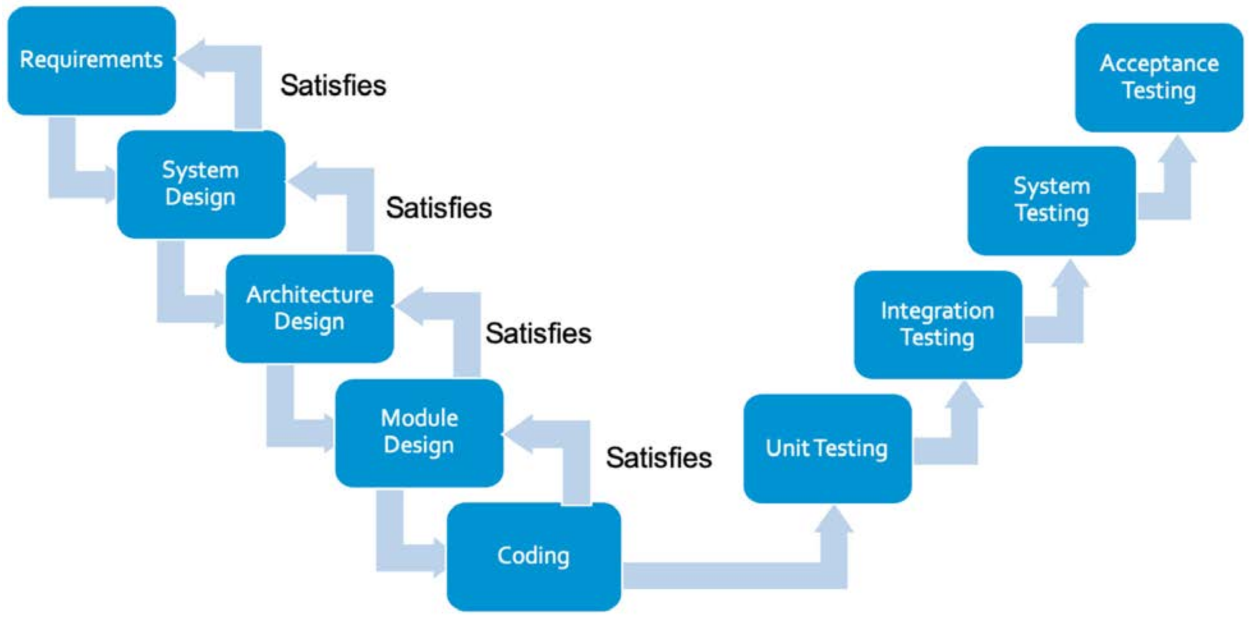


Figure 9-1: The classic V-model diagram shows how traceability goes forward and backward through each phase of development.

Each phase drives the subsequent phase. In turn, the work items in these phases must satisfy the requirements from the previous phase. System design is driven from requirements. System design satisfies the requirements and so on.

Requirements traceability management (RTM) proves that each phase is satisfying the requirements of each subsequent phase. However, this is only half of the picture. None of this traceability demonstrates that requirements are being met. That requires testing.

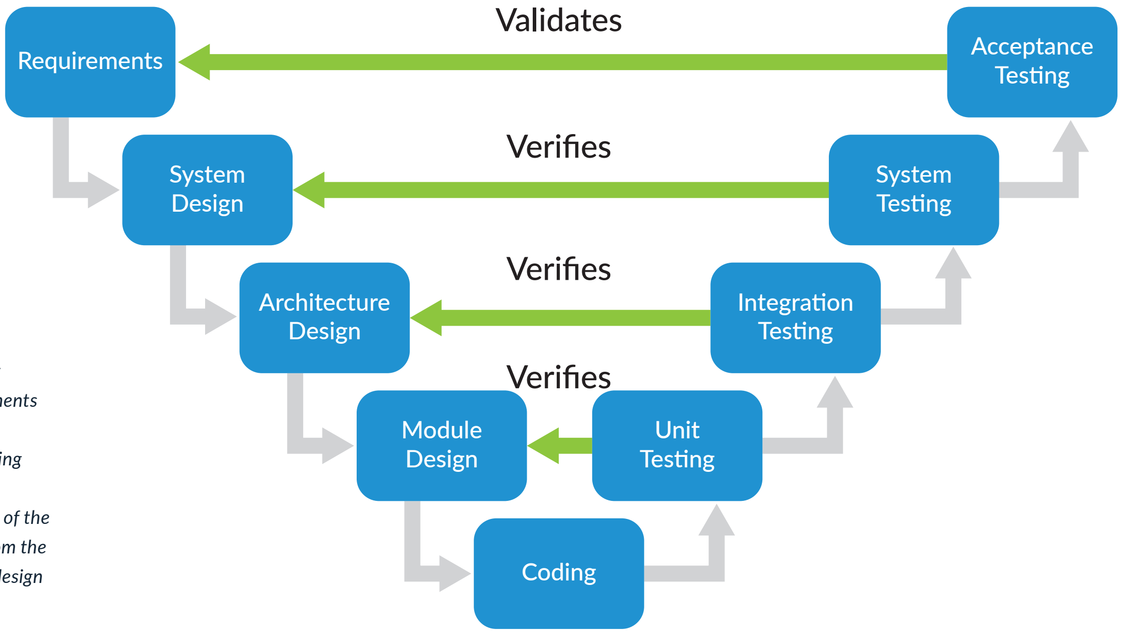


Figure 9-2: The other important part of requirements traceability is verification testing to prove the implementation of the specification from the corresponding design phase.

In the V-model, each testing phase verifies and validates (V&V) the corresponding design/implementation phase. In the example, you see:

» Acceptance testing validates requirements.

» System testing verifies the system design.

» Integration testing verifies architecture design.

» Unit testing verifies module design and so on.

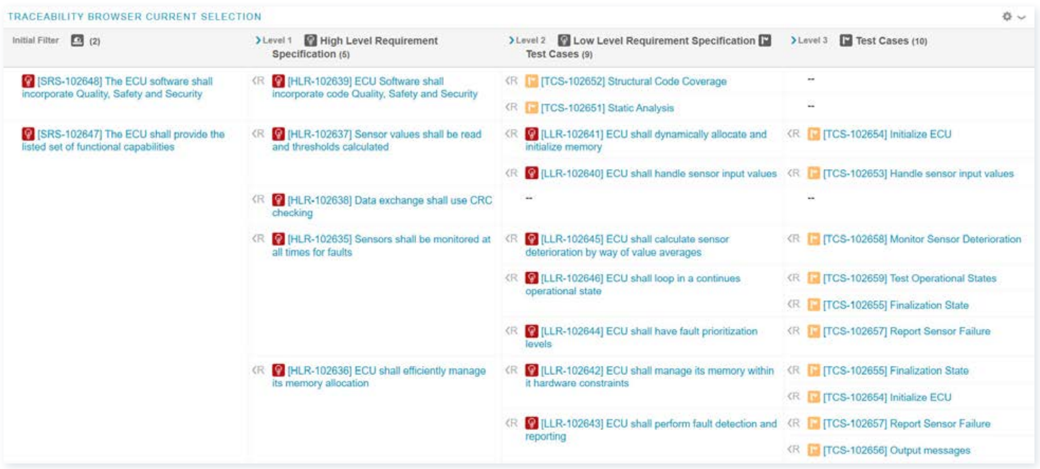
Software development on any realistic scale will have many requirements, complex design and architecture, and possibly thousands of units and unit tests. Automation of RTM in testing is necessary, especially for safety-critical software that requires documentation of traceability for certifications and audits.

Requirements Traceability Matrix

A requirement traceability matrix is an artifact or document that illustrates the linking of requirements with corresponding work items, like a unit test, module source code, architecture design element, other requirements, and so on.

The matrix is often displayed as a table, which shows how each requirement is "checked off" by a corresponding part of the product. Creation and maintenance of these matrices are often automated with requirements management tools with the ability to display them visually in many forms and even hard copy, if required.

Below is a requirements traceability matrix example from Intland Codebeamer. It shows system level requirements decomposed to high-level and low-level requirements, and the test cases that verify each.



TRACEABILITY BROWSER CURRENT SELECTION

Initial Filter (2)	Level 1 High Level Requirement Specification (5)	Level 2 Low Level Requirement Specification Test Cases (9)	Level 3 Test Cases (10)
[SRS-102648] The ECU software shall incorporate Quality, Safety and Security	[HLR-102639] ECU Software shall incorporate code Quality, Safety and Security	[TCS-102652] Structural Code Coverage	--
		[TCS-102651] Static Analysis	--
[SRS-102647] The ECU shall provide the listed set of functional capabilities	[HLR-102637] Sensor values shall be read and thresholds calculated	[LLR-102641] ECU shall dynamically allocate and initialize memory	[TCS-102654] Initialize ECU
		[LLR-102640] ECU shall handle sensor input values	[TCS-102653] Handle sensor input values
	[HLR-102638] Data exchange shall use CRC checking	--	--
	[HLR-102635] Sensors shall be monitored at all times for faults	[LLR-102645] ECU shall calculate sensor deterioration by way of value averages	[TCS-102658] Monitor Sensor Deterioration
		[LLR-102646] ECU shall loop in a continues operational state	[TCS-102659] Test Operational States
			[TCS-102655] Finalization State
		[LLR-102644] ECU shall have fault prioritization levels	[TCS-102657] Report Sensor Failure
	[HLR-102636] ECU shall efficiently manage its memory allocation	[LLR-102642] ECU shall manage its memory within it hardware constraints	[TCS-102655] Finalization State
			[TCS-102654] Initialize ECU
		[LLR-102643] ECU shall perform fault detection and reporting	[TCS-102657] Report Sensor Failure
			[TCS-102656] Output messages

Figure 9-3: Requirements traceability matrix example in Intland Codebeamer

Automating Bidirectional Traceability

Maintaining traceability records on any sort of scale requires automation. Application life cycle management tools include requirements management capabilities that are mature and tend to be the hub for traceability. Integrated software testing tools like Parasoft complete the verification and validation of requirements by providing an automated bidirectional traceability to the executable test case, which includes the pass or fail result and traces down to the source code that implements the requirement.

Parasoft integrates with market-leading requirements management and Agile planning systems including:

- IBM DOORS Next
- PTC Codebeamer
- Siemens Polarion
- Jama Connect
- Atlassian Jira
- CollabNet VersionOne
- TeamForge
- Azure DevOps Requirements

As shown in the image below, each of Parasoft's test automation tools, C/C++test, C/C++test CT, Jtest, dotTEST, SOAtest, and Selenic, support the association of tests with work items defined in these systems, such as:

- Requirements
- Stories
- Defects
- Test case definitions

Traceability is managed through the central reporting and analytics dashboard, Parasoft DTP.

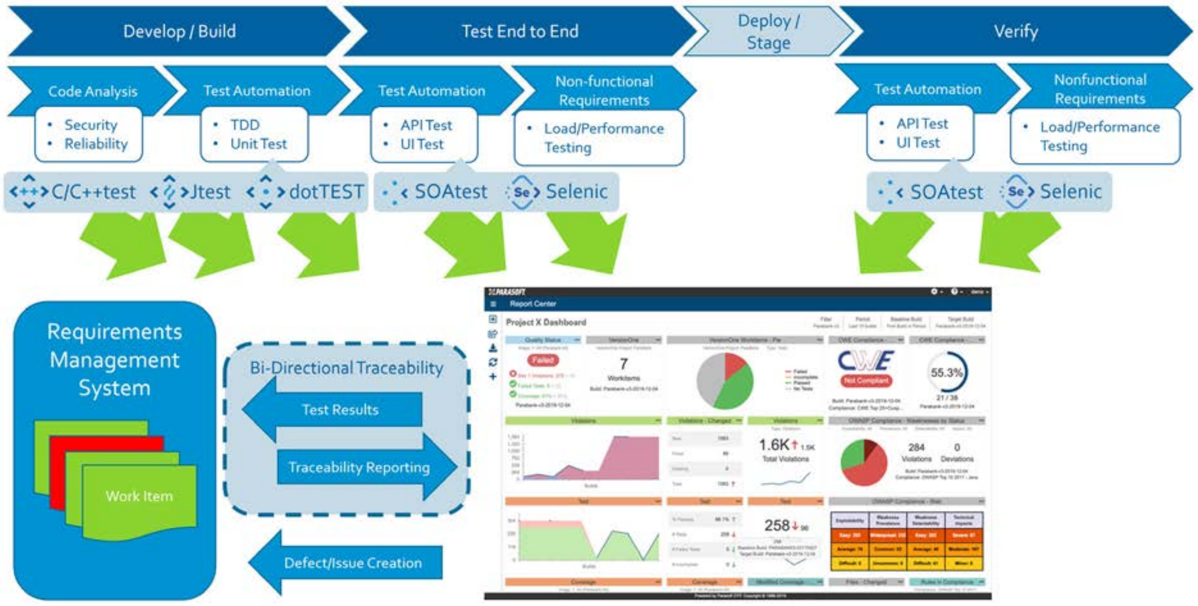


Figure 9-4: Parasoft provides bidirectional traceability from work items to test cases and test results, displaying traceability reports with Parasoft DTP and reporting results back to the requirements management system.

Parasoft DTP correlates the unique identifiers from the management system with the following:

» Static analysis findings

» Code coverage

» Test results from unit, integration, and functional tests.

Results are displayed within Parasoft DTP's traceability reports and sent back to the requirements management system. They provide full bidirectional traceability and reporting as part of the system's traceability matrix.

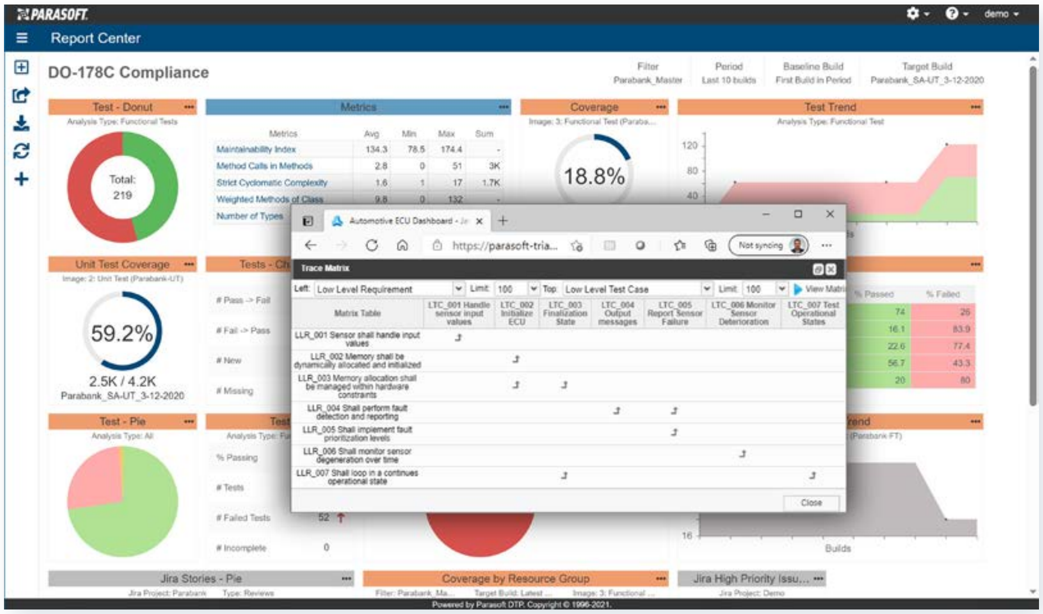
The traceability reporting in Parasoft DTP is highly customizable. The following image shows a requirements traceability matrix template with requirements authored in Polarion that trace to the following:

» Test cases

» Static analysis findings

» Source code files

» Manual code reviews



Metrics	Avg	Min	Max	Sum
Maintainability Index	134.3	78.5	174.4	-
Method Calls in Methods	2.8	0	51	3K
Strict Cyclomatic Complexity	1.6	1	17	1.7K
Weighted Methods of Class	9.8	0	132	-
Number of Types				

Left: Low Level Requirement Limit: 100 Top: Low Level Test Case Limit: 100 View Matrix

Matrix Table	LTC_001 Handle sensor input values	LTC_002 Initialize ECU	LTC_003 Finalization State	LTC_004 Output messages	LTC_005 Report Sensor Failure	LTC_006 Monitor Sensor Deterioration	LTC_007 Test Operational States
LLR_001 Sensor shall handle input values	✓						
LLR_002 Memory shall be dynamically allocated and initialized		✓					
LLR_003 Memory allocation shall be managed within hardware constraints		✓	✓				
LLR_004 Shall perform fault detection and reporting				✓	✓		
LLR_005 Shall implement fault prioritization levels					✓		
LLR_006 Shall monitor sensor degeneration over time						✓	
LLR_007 Shall loop in a continues operational state			✓				✓

Figure 9-5: Jama Requirements matrix, and integration with Parasoft DTP

The bidirectional correlation between test results and work items provides the basis of requirements traceability. Parasoft DTP adds test and code coverage analysis to evaluate test completeness. Maintaining this bidirectional correlation between requirements, tests, and the artifacts that implement them is an essential component of traceability.

Bidirectional traceability is important so that requirement management tools and other life cycle tools can correlate results and align them with requirements and associated work items.

The complexity of modern software projects requires automation to scale requirements traceability. Parasoft tools are built to integrate with best-of-breed requirement management tools to aid traceability of test automation results and complete the software test verification and validation of requirements.

A Unified, Fully Integrated Testing Solution for C/C++ Software Development

Tool Qualification for Safety-Critical Airborne Systems

Safety-critical software development standards recommend that manufacturers prove that the tools they're using to develop software don't introduce issues and do provide correct, predictable results.

The process of providing such evidence is known as tool qualification. While it's a necessary process, tool qualification is often a tedious and time-consuming activity for which many organizations fail to plan. To make this painless, select tools are certified and have a history of being used in the development of safety-critical applications.

In the case of airborne systems software development, DO-330, Software Tool Qualification Considerations, provides guidance on tool qualification. The purpose is to provide a framework for a tool qualification life cycle that includes planning, verification, quality assurance, and documentation. There are different levels of tool qualification from 1 to 5, with 5 being the least rigorous. The level is based on the possible impact of the tool on system safety.

Here are some of the key steps involved in tool qualification, according to DO-330.

» **Plan for tool qualification.** A comprehensive tool qualification plan (TQP) is required. In this plan, define the scope of the qualification effort, identify the tools to be qualified, outline the qualification activities, and specify the qualification objectives.

» **Tool classification.** Software tools are classified based on their impact on system safety primarily but also the potential impact on the development and verification processes. Tools are classified into one of five Tool Qualification Levels (TQL): TQL 1, TQL 2, TQL 3, TQL 4, TQL 5. TQL 1 represents the highest impact and TQL 5 the lowest.

» **Tool assessment.** Conduct a thorough assessment of each tool's development process, documentation, and characteristics to determine its qualification requirements. This includes reviewing the tool's design, verification, validation, and maintenance procedures. Obviously, this requires cooperation if tools are purchased from third parties.

» **Tool qualification assurance levels (AL).** Assign an Assurance Level that corresponds to DO-278A assurance levels to each tool based on the TQL and the level of confidence in the tool's development process. ALs range from AL 1 (highest assurance) to AL 5 (lowest assurance).

» **Tool verification and validation.** Perform the necessary verification and validation activities for each tool, demonstrating correct operation and accurate results.

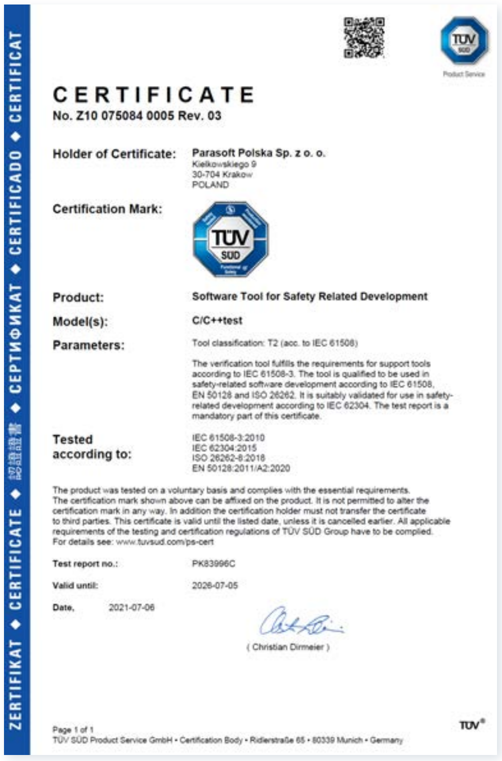
» **Tool life cycle maintenance.** Establish a process for the ongoing maintenance and monitoring of each tool. This includes periodic reviews, updates, and requalification as needed when changes occur to the tool or its environment.

» **Qualification records.** Maintain records of all tool qualification activities, including the assessment, verification, validation, and results. These records are essential for audit purposes and to demonstrate compliance with DO-330.

» **Final qualification report.** Prepare a final qualification report for each tool, summarizing the entire qualification process, the results of assessments and verification and validation activities, and the compliance status with DO-330 requirements.

The end deliverable is proof in the form of documentation. The qualification process outlined in DO-330 is complex and time consuming. Parasoft's Qualification Kits for C/C++test includes a convenient tool wizard that brings automation into the picture and reduces the time and effort required for tool qualification.

Figure 10-1: Parasoft CIC++test and C/C++test CT TÜV SÜD certificate



ZERTIFIKAT ◆ CERTIFICATE ◆ 認證證書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

Product Service

CERTIFICATE
No. Z10 075084 0005 Rev. 03

Holder of Certificate:	**Parasoft Polska Sp. z o. o.** Kielkowskiego 9 30-704 Krakow POLAND
Certification Mark:	
Product:	**Software Tool for Safety Related Development**
Model(s):	**C/C++test**
Parameters:	Tool classification: T2 (acc. to IEC 61508) The verification tool fulfills the requirements for support tools according to IEC 61508-3. The tool is qualified to be used in safety-related software development according to IEC 61508, EN 50128 and ISO 26262. It is suitably validated for use in safety-related development according to IEC 62304. The test report is a mandatory part of this certificate.
Tested according to:	IEC 61508-3:2010 IEC 62304:2015 ISO 26262-8:2018 EN 50128:2011/A2:2020

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. This certificate is valid until the listed date, unless it is cancelled earlier. All applicable requirements of the testing and certification regulations of TÜV SÜD Group have to be complied. For details see: www.tuvsud.com/ps-cert

Test report no.:	PK83996C
Valid until:	2026-07-05
Date,	2021-07-06

[illegible]
(Christian Dirmeier)

Page 1 of 1
TÜV SÜD Product Service GmbH • Certification Body • Ridlerstraße 65 • 80339 Munich • Germany

Precertified Tools

Tool qualification needs to start with tool selection to ensure that you're using a development tool that's certified by an organization like TÜV SÜD. This will significantly reduce the effort when it comes to tool qualification.

Parasoft C/C++test, C/C++test CT, and DTP are certified by TÜV SÜD for functional safety according to IEC, ISO, and other functional safety industry standards for both host based and embedded target applications. Though the certificate is not enough for RTCA DO-178C/DO-330, it demonstrates a historical commitment by Parasoft in providing quality products.

To satisfy DO-330 tool qualification requirements, C/C++ software development paves the way for a streamlined qualification of static analysis, unit testing, and coverage requirements for the safety-critical standards by offering a tool qualification kit that automates the tool qualification process for any development host and/or target ecosystem.

Automating Tool Qualification Testing

Traditionally, tool qualification has meant significant amounts of manual labor, testing, and documenting to satisfy a certification audit. But this documentation-heavy process requires manual interpretation and completion. As a result, it's time consuming and adds to an organization's already heavy testing schedule and budget.

Parasoft leverages its own software test automation tool qualification with Qualification Kits, which include a documented workflow to dramatically reduce the amount of effort required.

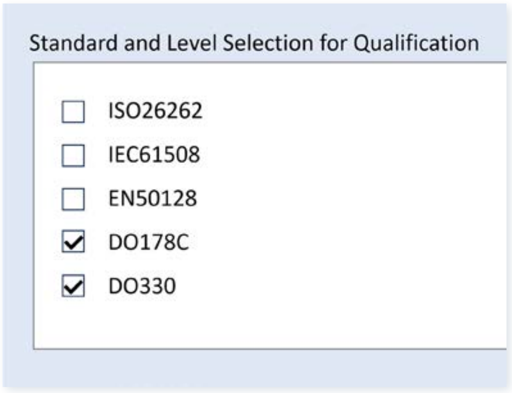
Benefits of Using the Qualification Kits

- Automatically reduce the scope of qualification to only the parts of the tool in use.
- Automate tests required for qualification as much as possible.
- Manage any manual tests as eloquently as possible and integrate results alongside automated tests.
- Automatically generate audit-ready documentation that reports on exactly what's being qualified—not more, not less.

Qualify Only the Tools Used

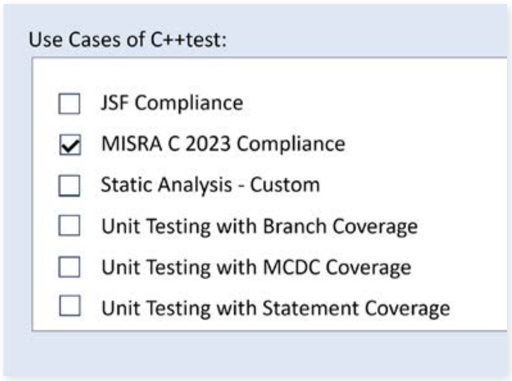
There should be no need to do any extra work for qualifying capabilities not used during development. Reducing the scope of testing, reporting, and documentation is a key way to reduce the qualification workload.

For example, as part of the DO-178C/DO-330 tool qualification kit and process, users can select Parasoft C/C++test for static analysis of C/C++ code to check its compliance to the MISRA C:2023 standard. The tool then selects only the parts of the qualification suite needed for this function.



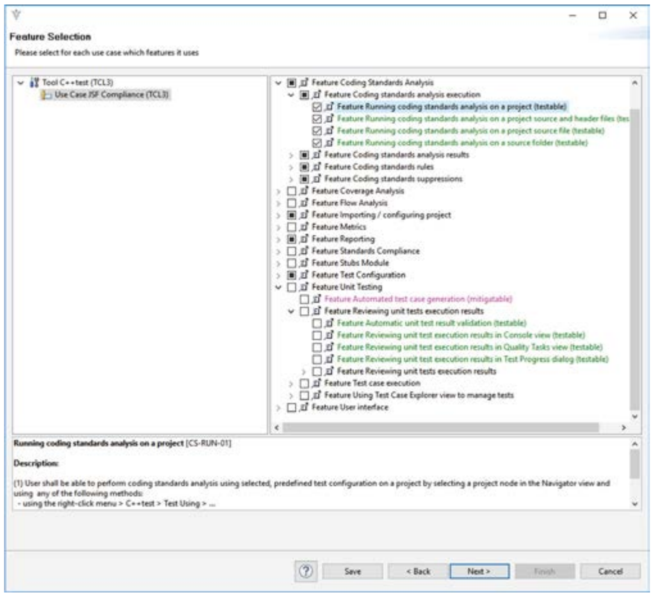
Standard and Level Selection for Qualification

- ☐ ISO26262
- ☐ IEC61508
- ☐ EN50128
- ☑ DO178C
- ☑ DO330



Use Cases of C++test:

- ☐ JSF Compliance
- ☑ MISRA C 2023 Compliance
- ☐ Static Analysis - Custom
- ☐ Unit Testing with Branch Coverage
- ☐ Unit Testing with MCDC Coverage
- ☐ Unit Testing with Statement Coverage



Feature Selection

Please select for each use case which features it uses

```
Tool C++test (TCL3)
    Use Case JSF Compliance (TCL3)
```

```
■ Feature Coding Standards Analysis
    ■ Feature Coding standards analysis execution
        ☑ Feature Running coding standards analysis on a project (testable)
        ☑ Feature Running coding standards analysis on a project source and header files (tes
        ☑ Feature Running coding standards analysis on a project source file (testable)
        ☑ Feature Running coding standards analysis on a source folder (testable)
    ■ Feature Coding standards analysis results
    ■ Feature Coding standards rules
    ■ Feature Coding standards suppressions
☐ Feature Coverage Analysis
☐ Feature Flow Analysis
■ Feature Importing / configuring project
☐ Feature Metrics
■ Feature Reporting
☐ Feature Standards Compliance
☐ Feature Stubs Module
■ Feature Test Configuration
☐ Feature Unit Testing
    ☐ Feature Automated test case generation (mitigatable)
    ☐ Feature Reviewing unit tests execution results
        ☐ Feature Automatic unit test result validation (testable)
        ☐ Feature Reviewing unit test execution results in Console view (testable)
        ☐ Feature Reviewing unit test execution results in Quality Tasks view (testable)
        ☐ Feature Reviewing unit test execution results in Test Progress dialog (testable)
        ☐ Feature Reviewing unit tests execution results
    ☐ Feature Test case execution
    ☐ Feature Using Test Case Explorer view to manage tests
☐ Feature User interface
```

Running coding standards analysis on a project [CS-RUN-01]

Description:

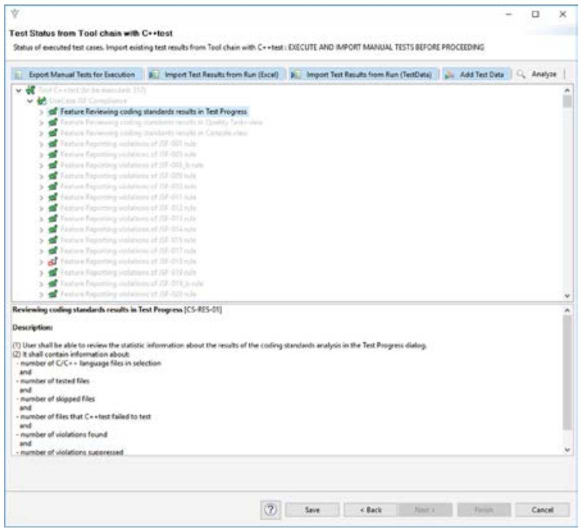
(1) User shall be able to perform coding standards analysis using selected, predefined test configuration on a project by selecting a project node in the Navigator view and using any of the following methods:
- using the right-click menu > C++test > Test Using > ...

Figure 10-2: Parasoft Qualification Kits allow users to select the options required for their project. Upon selection, only tests and documentation are used and provided from this point forward.

Leverage Test Automation & Analytics

A unique advantage to qualifying test automation tools is that the tools can be used to automate their own testing. Automating this as much as possible is key to making it as painless as possible. Even manual tests, which are inevitable for any development tool, are handled as efficiently as possible. Step by step instructions are provided and results are entered and stored as part of the qualification record.

Parasoft C/C++test collects and stores all test results from each build. Tests run as they do for any type of project. These results are brought into the test status wizard in the Parasoft Qualification Kits to provide a comprehensive overview of the results like those shown below.


Test Status from Tool chain with C++test
Export Manual Tests for Execution
Import Test Results from Run (Excel)
Import Test Results from Run (TestData)
Add Test Data
Analyze
Save
< Back
Next >
Finish
Cancel

Status of executed test cases. Import existing test results from Tool chain with C++test : EXECUTE AND IMPORT MANUAL TESTS BEFORE PROCEEDING

```
Test C++test (to be executed: 317)
  UseCase JSF Compliance
    Feature Reviewing coding standards results in Test Progress
    Feature Reviewing coding standards results in Quality Tasks view
    Feature Reviewing coding standards results in Console view
    Feature Reporting violations of JSF-001 rule
    Feature Reporting violations of JSF-003 rule
    Feature Reporting violations of JSF-008_b rule
    Feature Reporting violations of JSF-009 rule
    Feature Reporting violations of JSF-010 rule
    Feature Reporting violations of JSF-011 rule
    Feature Reporting violations of JSF-012 rule
    Feature Reporting violations of JSF-013 rule
    Feature Reporting violations of JSF-014 rule
    Feature Reporting violations of JSF-015 rule
    Feature Reporting violations of JSF-017 rule
    Feature Reporting violations of JSF-018 rule
    Feature Reporting violations of JSF-019 rule
    Feature Reporting violations of JSF-019_b rule
    Feature Reporting violations of JSF-020 rule
```

Reviewing coding standards results in Test Progress [CS-RES-01]

Description:

(1) User shall be able to review the statistic information about the results of the coding standards analysis in the Test Progress dialog.
(2) It shall contain information about:
- number of C/C++ language files in selection
  and
- number of tested files
  and
- number of skipped files
  and
- number of files that C++test failed to test
  and
- number of violations found
  and
- number of violations suppressed

Figure 10-3: Leveraging centralized data collection and automating the qualification process greatly reduces manual tracking of the compliance progress.

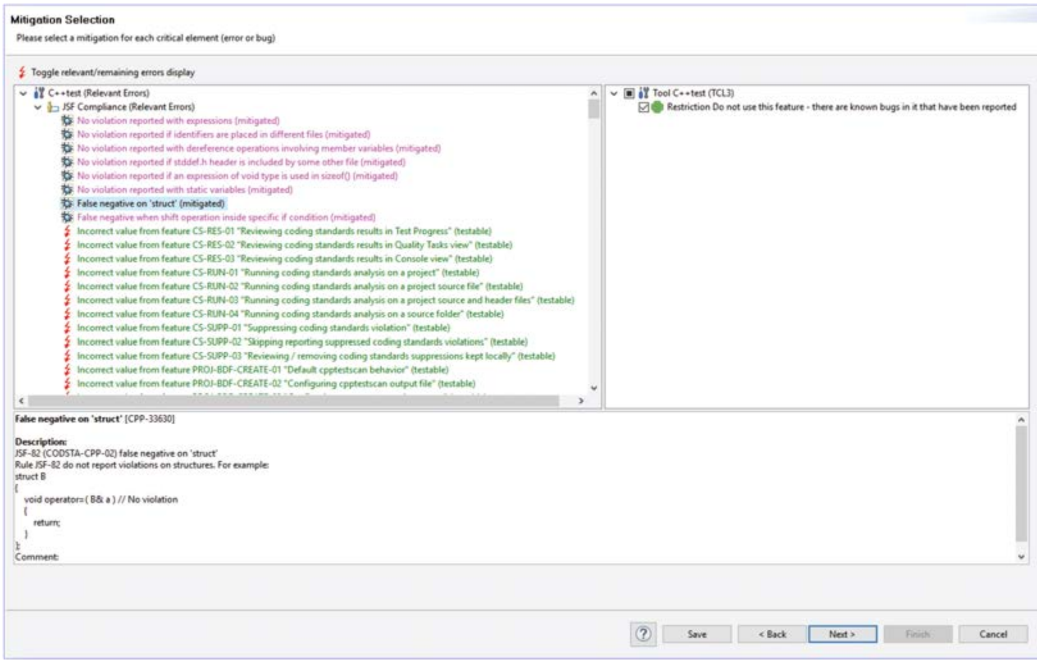
Managing Known Defects

Every development tool has known bugs and any vendor selling products for safety-critical development must have these documented. There's more to dealing with known defects than just documenting them.

Tool qualification requires proof that these defects are not affecting the results used for verification and validation. For each known defect, the manufacturer must provide a mitigation for each one and document it to the satisfaction of the certifying auditor.

It's incumbent on the tool vendor to automate the handling of known defects as much as possible. After all, the vendor is expecting customers to deal with third-party software bugs as part of their workload!

The Parasoft C/C++test Qualification Kits include a wizard to automate the recording of mitigation for known defects as shown in the example below.



Mitigation Selection
Please select a mitigation for each critical element (error or bug)

Toggle relevant/remaining errors display

- C++test (Relevant Errors)
  - JSF Compliance (Relevant Errors)
    - No violation reported with expressions (mitigated)
    - No violation reported if identifiers are placed in different files (mitigated)
    - No violation reported with dereference operations involving member variables (mitigated)
    - No violation reported if stddef.h header is included by some other file (mitigated)
    - No violation reported if an expression of void type is used in sizeof() (mitigated)
    - No violation reported with static variables (mitigated)
    - False negative on 'struct' (mitigated)
    - False negative when shift operation inside specific if condition (mitigated)
    - Incorrect value from feature CS-RES-01 "Reviewing coding standards results in Test Progress" (testable)
    - Incorrect value from feature CS-RES-02 "Reviewing coding standards results in Quality Tasks view" (testable)
    - Incorrect value from feature CS-RES-03 "Reviewing coding standards results in Console view" (testable)
    - Incorrect value from feature CS-RUN-01 "Running coding standards analysis on a project" (testable)
    - Incorrect value from feature CS-RUN-02 "Running coding standards analysis on a project source file" (testable)
    - Incorrect value from feature CS-RUN-03 "Running coding standards analysis on a project source and header files" (testable)
    - Incorrect value from feature CS-RUN-04 "Running coding standards analysis on a source folder" (testable)
    - Incorrect value from feature CS-SUPP-01 "Suppressing coding standards violation" (testable)
    - Incorrect value from feature CS-SUPP-02 "Skipping reporting suppressed coding standards violations" (testable)
    - Incorrect value from feature CS-SUPP-03 "Reviewing / removing coding standards suppressions kept locally" (testable)
    - Incorrect value from feature PROJ-BDF-CREATE-01 "Default cpptestscan behavior" (testable)
    - Incorrect value from feature PROJ-BDF-CREATE-02 "Configuring cpptestscan output file" (testable)

- Tool C++test (TCL3)
  - [✓] Restriction Do not use this feature - there are known bugs in it that have been reported

```
False negative on 'struct' [CPP-33630]

Description:
JSF-82 (CODSTA-CPP-02) false negative on 'struct'
Rule JSF-82 do not report violations on structures. For example:
struct B
{
   void operator=( B& a ) // No violation
   {
      return;
   }
};
Comment:
```

Figure 10-4: Known defects are managed directly in Parasoft C/C++test.

Automation of Tool Qualification Documentation

The end result of tool qualification is documentation and lots of it. Every test executed with results, every known defect with mitigation, manual test results, and exceptions are all recorded and reported. Qualification kits from other vendors can be just documentation alone and, without automation, documenting compliance is tedious.

Instead, using the Qualification Kits for C/C++test, the critical documents are generated automatically as part of the workflow.

» **Tool Classification Report** determines the qualification needed and presents the maximum safety level classification for C/C++test and C/C++test CT based on the use cases selected by the user.

» **Tool Qualification Plan** describes how C/C++test and CC++test CT will be qualified for use in a safety relevant development project.

» **Tool Qualification Report** demonstrates that C/C++test and C/C++test CT have has been qualified according to the tool qualification plan.

» **Tool Safety Manual** describes how C/C++test and C/C++test CT should be used safely, for example, in compliance with safety standards like IEC 62304 in safety-critical projects.

In each of these documents, only the documentation required for the tool featured in use is generated because the scope of the qualification was narrowed down at the beginning of the project. Teams greatly reduce the documentation burden with automation and narrowing the qualification scope.

Reporting & Analytics for Safety-Critical Airborne Systems

Parasoft's extensive reporting capabilities bring the results of Parasoft C/C++test and C/C++test CT into context. Test results can quickly be accessed within the IDE or exported into the web-based reporting system, DTP.

In DTP, reports can be automatically generated as part of CI builds and printed for code audits in safety-critical organizations. Results from across builds can be aggregated to give the team a detailed view without requiring access to the code within their IDE.

In the reporting dashboard, Parasoft's Process Intelligence Engine (PIE) helps managers understand the quality of a project over time. It illustrates the impact of change after each new code change. Integrating with the overall toolchain, PIE provides advanced analytics that pinpoint areas of risk.

Developer's View in the IDE

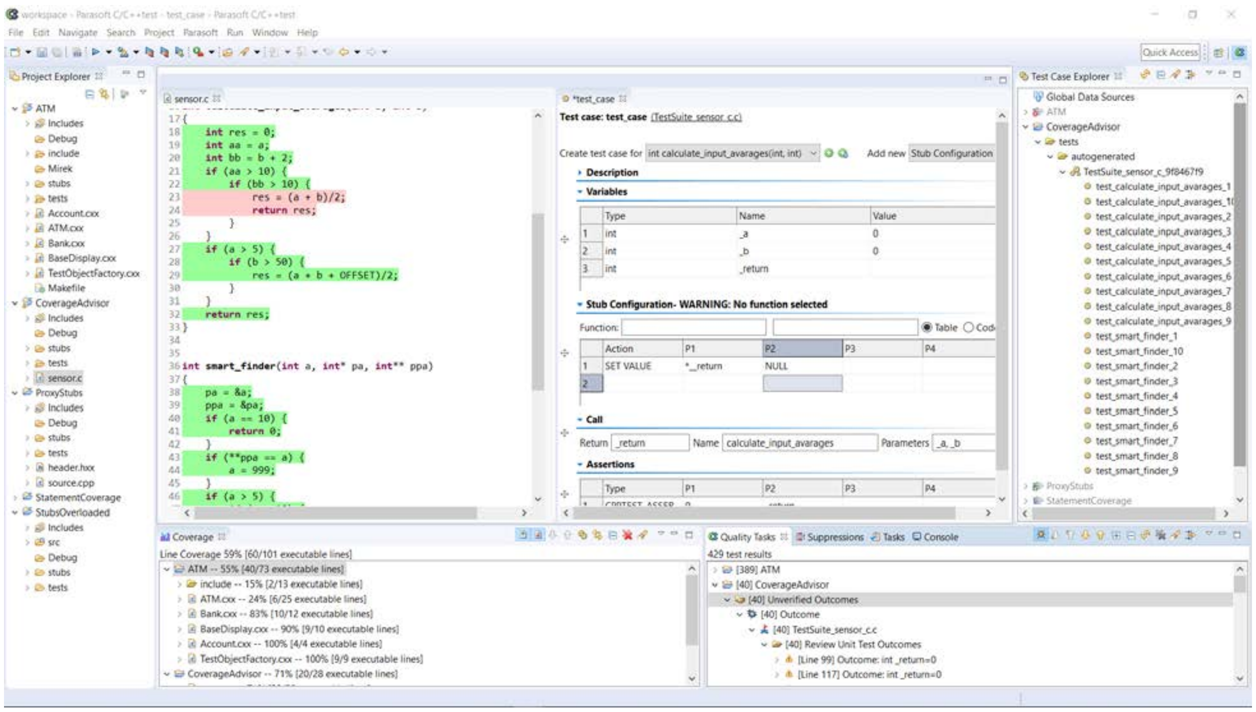
Parasoft C/C++test helps teams efficiently understand results from software testing by reporting and analyzing results in multiple ways. Users can view the following directly in the developer's IDE:

- Static analysis findings including warnings and coding standard violations
- Unit testing details like passed/failed assertions, exceptions with stack traces, info/debug messages
- Runtime analysis failures with allocation stack traces
- Code coverage details such as percentage values and code highlights like coverage test case correlation

The Quality Tasks view in the IDE makes it easy for developers to sort and filter the results, for example, by file, rule, or project. Developers can make annotations directly in the source code editors to correlate issues with the source code. This provides context and more details about reported issues and how to apply a fix.

Code coverage information is presented with visual green and red highlights displayed in the code editor, together with percentage values for project, file, and function in a dedicated Coverage view.

Analysis results for both IDE and command line workflows can also be exported to standard HTML and PDF reports for local reporting. For safety-critical software development, C/C++test provides an additional dedicated report format. It details unit test case configuration and includes the log of results from test execution. Users get a complete report of how the test case was constructed and what happened during runtime.


```
{
    int res = 0;
    int aa = a;
    int bb = b + 2;
    if (aa > 10) {
        if (bb > 10) {
            res = (a + b)/2;
            return res;
        }
    }
    if (a > 5) {
        if (b > 50) {
            res = (a + b + OFFSET)/2;
        }
    }
    return res;
}


int smart_finder(int a, int* pa, int** ppa)
{
    pa = &a;
    ppa = &pa;
    if (a == 10) {
        return 0;
    }
    if (**ppa == a) {
        a = 999;
    }
    if (a > 5) {
```

Test case: test_case (TestSuite_sensor_c.c)

Create test case for: int calculate_input_avarages(int, int)   Add new: Stub Configuration

Variables

	Type	Name	Value
1	int	_a	0
2	int	_b	0
3	int	_return	

Stub Configuration- WARNING: No function selected

	Action	P1	P2	P3	P4
1	SET VALUE	*__return	NULL		
2					

Call: Return _return   Name calculate_input_avarages   Parameters _a, _b

```
Line Coverage 59% [60/101 executable lines]
ATM -- 55% [40/73 executable lines]
  include -- 15% [2/13 executable lines]
  ATM.cxx -- 24% [6/25 executable lines]
  Bank.cxx -- 83% [10/12 executable lines]
  BaseDisplay.cxx -- 90% [9/10 executable lines]
  Account.cxx -- 100% [4/4 executable lines]
  TestObjectFactory.cxx -- 100% [9/9 executable lines]
CoverageAdvisor -- 71% [20/28 executable lines]
```

```
429 test results
[389] ATM
[40] CoverageAdvisor
  [40] Unverified Outcomes
    [40] Outcome
      [40] TestSuite_sensor_c.c
        [40] Review Unit Test Outcomes
          [Line 99] Outcome: int _return=0
          [Line 117] Outcome: int _return=0
```

Figure 11-1: Parasoft C/C++test IDE unified code coverage and unit testing view

Team Web-Based Reporting

For team collaboration, Parasoft C/C++test and C/C++test CT publishes analysis results to DTP, a centralized server. Developers can access test results from automated runs and project managers can quickly assess the quality of the project. Reported results are stored with a build identifier for full traceability between the results and the build. Those results include details about the following:

- Static analysis
- Metric analysis
- Unit testing
- Code coverage
- Source code

When integrating into CI/CD workflows, Parasoft users benefit from a centralized and flexible web-based interface for browsing results. The dynamic web-based reporting dashboard includes:

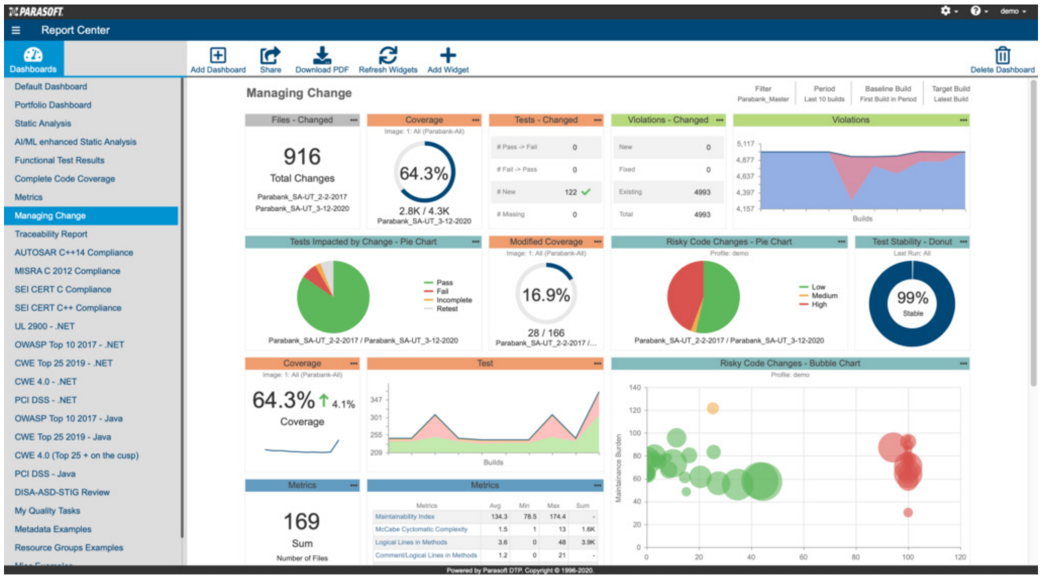
» Customizable reporting widgets

» Source code navigation

» Advanced filtering

» Advanced analytics from the Process Intelligence Engine

Users can access historical data and trends, apply baselining and test impact analysis, and integrate with external systems like those for test requirements traceability.



Metrics	Avg	Min	Max	Sum
Maintainability Index	134.3	78.5	174.4	-
McCabe Cyclomatic Complexity	1.5	1	13	1.6K
Logical Lines in Methods	3.6	0	48	3.9K
Comment/Logical Lines in Methods	1.2	0	21	-

Figure 11-2: Centralized web based dashboard for test impact analysis and more

Test Impact Analysis

Each and every test performed, including manual, system level, and UI-based, is recorded as a pass/fail result, including the coverage impact on the code base. Each additional test is overlaid on this existing information, creating a complete picture of test success and coverage.

As code is changed, the impact is clearly visible on the underlying record, highlighting tests that now fail or code that is now untested. Raising this information in various degrees of detail allows developers and testers to quickly identify what needs to be altered or fixed for the next test run.

Risk-Based Assessment

In addition to change impact analysis, static analysis can be used to highlight areas of the code that appear riskier than others. Risk can take a variety of forms including:

» Highly complex code

» Unusually high number of coding standard violations

» High number of reported static analysis warnings

These are areas of code that may require additional test coverage and even refactoring.

Functional Safety Reporting

Parasoft C/C++test and C/C++test CT provide specific reporting capabilities suited to functional safety development. Here are two report examples.

1. Unit Testing Execution Details Tests to Requirements Traceability
2. Test to Code Coverage Traceability

Code Coverage Metrics

There are various coverage metrics to consider. For safety-critical airborne systems, coverage may be one of the following:

» Statements

» Branch

» Modified condition/decision coverage (MC/DC)

» Object/assembly code for the strictest requirements

Parasoft supports gathering all of these coverage metrics, including terms other industries use like block, call, function, path, decision, and more.

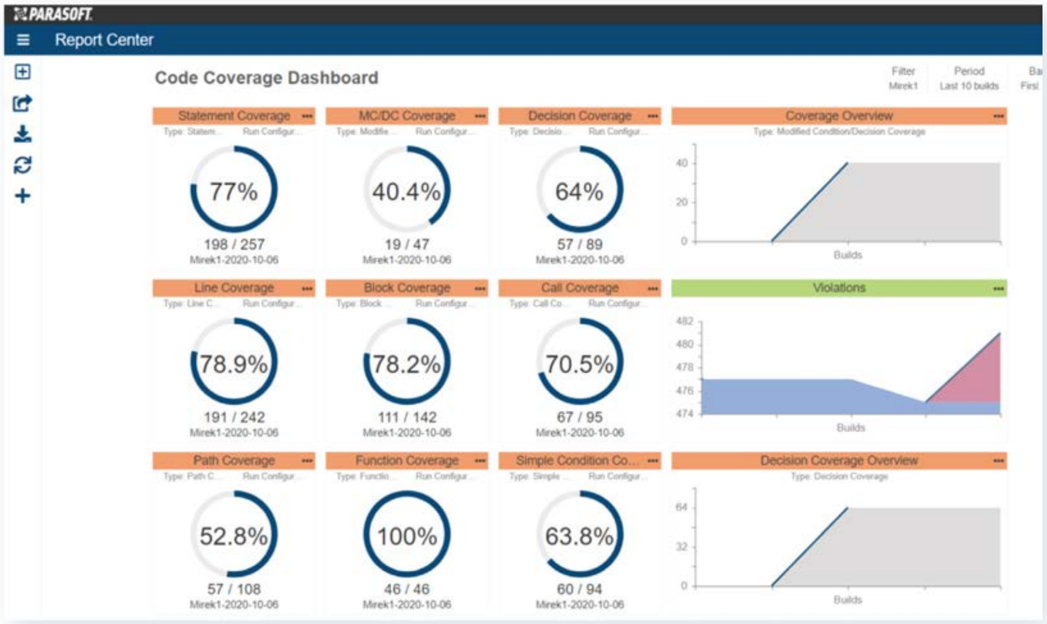


Figure 11-3: Individual code coverage metrics available within the reporting dashboard

Custom Analytics, Reports, & Dashboards

Parasoft DTP is highly customizable and supports a user-configured custom processor for project-specific analysis, custom widgets, and dashboards.

Benefits of Centralized, Aggregated Data Analysis & Reporting

Development teams with one analysis and reporting system for compliance reap the following benefits.

» Efficiency, visibility, and ease of use

» Reduced overhead

» Clear insight into new and legacy code

Manage Compliance With Efficiency, Visibility, & Ease

Instead of just providing static analysis checkers with basic reporting and trends visualization, Parasoft's solution for coding standards compliance provides a complete framework for building a stable and sustainable compliance process.

In addition to standard reporting, Parasoft provides a dedicated compliance reporting module that gives users a dynamic view into the compliance process. Users can see results grouped according to categorizations from the original coding standard, manage the deviations process, and generate compliance documents required for code audits and certification as defined by the MISRA Compliance:2020 specification.

Reduce the Overhead of Testing

With a unified reporting framework, Parasoft C/C++test efficiently provides multiple testing methodologies required by the functional safety standards including static analysis, unit testing, and code coverage.

By presenting cumulative results from the multiple testing techniques, Parasoft provides consistent reporting that reduces the overhead of testing activities. The analytics, reports, and dashboards provide the following benefits.

- Simplify code audits and the certification process.
- Eliminate the need for users to manually process reporting to build documentation for the certification process.
- Focus testing efforts where needed by eliminating extraneous testing and guesswork from test management.
- Reduce the costs of testing while improving test outcomes with better tests, more coverage, and streamlined test execution.
- Minimize the impact of changes by efficiently managing the change itself.

Pinpoint Priority & Risk Between New & Legacy Code

Parasoft's Process Intelligence Engine enables users to look at the changes between two builds to understand, for example, the level of code coverage or static analysis violations on the code that has been modified between development iterations, different releases, or an incremental development step from the baseline set on the legacy code.

Teams can converge on better quality over time by improving test coverage and reducing the potential risky code. The technical debt due to untested code, missed coding guidelines, and potential bugs and security vulnerabilities can be reduced gradually build by build. Using the information provided by Parasoft tools, teams can focus in on the riskiest code for better testing and maintenance.

Take the Next Step

Request a demo to see how your embedded development team can accelerate the delivery of high-quality, compliant software for safety-critical airborne systems.

About Parasoft

Parasoft helps organizations continuously deliver high-quality software with its AI-powered software testing platform and automated test solutions. Supporting the embedded, enterprise, and IoT markets, Parasoft's proven technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award-winning reporting and analytics dashboard provides a centralized view of quality, enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives—security, safety-critical, Agile, DevOps, and continuous testing.

More Resources

Safety-Critical Airborne Systems Software Development

Case Studies

- » Federal Agency Fulfills Rigorous DO-178C Standard With Unified Automated Testing Solution
- » Industry Leader Streamlines Workflow & Delivers Safe, Secure Avionic Systems
- » Aerospace/Defense Company Deploys Parasoft to Support DevSecOps for Major DoD Initiative

Website

- » Software Testing for Military and Defense Systems
- » DO-178C Compliance With Parasoft
- » MISRA Compliance With Parasoft
- » Easily Automate the Tool Qualification Process

Whitepapers

- » Developing DO-178C Compliant Software for Airborne Systems
- » A Practical Guide to Accelerate MISRA C 2023 Compliance With Test Automation
- » How to Streamline Unit Testing for Embedded and Safety-Critical Systems
- » Embedded Cybersecurity Through Secure Coding Standards CWE and CERT

Datasheets

- » Develop Compliant DO-178C Software for Airborne Systems
- » Assembly Coverage Tool
- » Parasoft C/C++test
- » Parasoft C/C++test CT

Blog Posts

- How to Obtain 100% Structural Code Coverage of Safety-Critical Systems
- Regression Testing of Embedded Systems
- Verification vs Validation in Embedded Software
- Robustness Testing: What Is It & How to Deliver Reliable Software Systems With Test Automation
- Reducing the Risk and Cost of Achieving Compliant Software
- MISRA C/C++ Code Checking
- The Two Big Traps of Code Coverage
- Shift-Left Your Safety-Critical Software Testing With Test Automation
- Requirements Management and the Traceability Matrix

Webinars

- Object Code Structural Coverage for DO-178C
- How to Validate DO-326A Airworthiness Security Requirements
- How Industry Leaders Are Delivering Safe & Secure Software
- Cut Compliance Costs and Ensure Lifecycle Traceability With codebeamer ALM & Parasoft
- Make Your C/C++ Applications Safe and Secure With MISRA and CERT
- Automate Essential Testing to Verify & Validate Polarion Requirements
- Requirement Traceability for Safety-Critical Applications
- Mastering Aviation Safety & Cybersecurity: DO-178C & DO-326A