PARASOFT®

ISO 26262 SOFTWARE COMPLIANCE IN THE AUTOMOTIVE INDUSTRY

Table of Contents

3 Overview

3 Automotive Industry Outlook

7 What Is ISO 26262?

14 Requirements for Compliance in Testing

14 Static Analysis

28 Unit Testing

33 Regression Testing

36 Software Integration Testing

44 Software System Testing

47 Structural Code Coverage

50 Requirements and the Traceability Matrix

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

55 Tool Qualification for Safety-Critical Automotive Systems

60 Reporting and Analytics for Automotive Software

66 More Resources

66 Safety-Critical Automotive Software Development Assets for Download

Overview

AUTOMOTIVE INDUSTRY OUTLOOK

The automotive industry continues to rapidly evolve and grow into technical areas where other industries have operated for many years. For example, NASA's Jet Propulsion Laboratory releases code fixes and new functionality that is currently being developed to spacecraft that is millions of miles away, on route to their destination. Similarly, we find the automotive industry now providing software updates on cars that have been sold and are being driven by their consumers all around the world.

SAFETY & SECURITY CHALLENGES

This type of evolution—particularly that of advanced driver-assistance systems (ADAS)—comes with a new set of challenges in safety and security. Standards like ISO 26262 address functional safety of the development of electric and electronic systems (E/E), which include propulsion, dynamic control systems, and driver assistance.

Additionally, platforms like AUTOSAR provide an open standardized software layer architecture that further improves safety. They include guidelines for the use of the C++14 language in development of critical and safety-related systems. However, manufacturers have realized that due to the increased complexity and unknows of modern technologies working together, along with changes in the internal and external environment, safety and security concerns have arisen that these standards don't address.

When addressing ISO 21434, it's important to understand that the recommended security consideration for cybersecurity should be integrated into your existing development processes. ISO 21434 references ISO 26262 in consideration of having these two disciplines take an interdisciplinary exchange of strategies, coordination and even tools used. This means that your organization should have your system engineers work with your security engineers through the requirements analysis phase for safety and security.

In parallel, perform hazard analysis and risk assessment (HARA) for safety, and threat analysis and risk assessment (TARA) for security. Nonetheless, a strong collaborative environment is needed to ensure a safe and secure result.

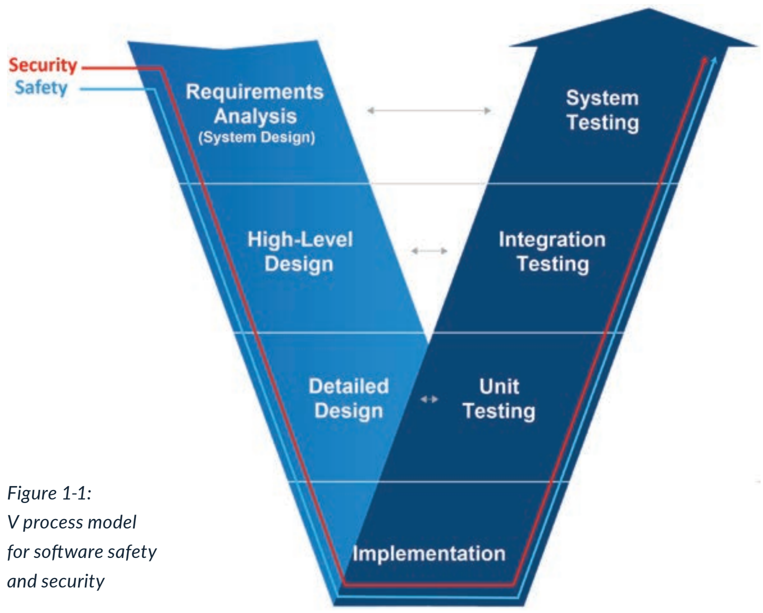


Figure 1-1: V process model for software safety and security

Ensuring security at the software implementation phase starts by applying static code analysis. The MISRA coding standard incorporates security guidelines, but you can also augment and strengthen code security by adopting CERT.

Continuing up the right side of the V, perform unit testing of all your low-level security requirements. In the next phase, create test cases that incorporate additional functionality. These test cases ensure that your high-level requirements are satisfied.

Moving to system testing, create system tests to ensure that the system requirements are verified. Confirm that all the test cases trace back to your requirements. This guarantees that no requirement goes untested. However, to safeguard that each requirement is fully tested, incorporate structural code coverage as recommended by ISO 21434 and ISO 26262. Code coverage ensures that your security test cases fully cover every possible path of execution through its security functionality remediation measures.

To overcome safety and security challenges, teams can turn to solutions like Parasoft C/C++test, which has been certified for use in safety-critical applications per ISO 26262 and is TÜV SÜD certified to satisfy ISO 21434. Both of these standards recommend performing static analysis, dynamic analysis—which includes unit, integration, and system testing—code coverage, and requirements traceability. Offering exactly what ISO 26262 and ISO 21434 recommend for software verification in safety and security, Parasoft also provides the documentation required to prove compliance with both standards.

UNECE WP.29 REGULATORY REQUIREMENTS

The United Nations Economic Commission for Europe (UNECE) released regulatory requirements on June 23, 2020, where they outlined new processes and technologies that automotive manufacturers must incorporate into both their organization and vehicles. These regulations also apply to Tier 1 and Tier 2 suppliers of software and hardware components, including mobile services.

Vehicle manufacturers are required to put into the organizational structure, a risk-based management framework for discovering, analyzing, and protecting against relevant threats, vulnerabilities, and cyberattacks.

The following categories require cybersecurity testing and passing inspections.

- Category M covers standard four wheel cars.
- Category N is for pickup trucks and vans.
- Categories L6 and L7 include electric cars and autonomous capabilities.

A passing grade on both organizational and vehicle WP.29 key requirements means that the manufacturer receives a certificate of compliance. New vehicles without this certificate cannot be sold in the EU after July 2024. Be aware, that the United States does not participate or have its own similar regulations. However, the writing is on the wall.

AUTOMOTIVE SPICE

Automotive Software Process Improvement and Capability Determination (ASPICE) provides a measurement framework for independent assessors to evaluate an organization's capability for software development. Ensuring software safety and cybersecurity does not only lie within the technical engineering aspects of the development of the electronic system, but also requires the organization to incorporate processes and checks.

These processes and checks must include ways to track and monitor progress within all practices of the organization to ensure:

1. Safety and cybersecurity practices have been adopted.
2. Safety and cybersecurity requirements are being satisfied.

This is also one of the two key certification criteria for UNECE WP.29 on organizational cybersecurity capability.

UNSAFE SCENARIOS

It's brought to fruition other outgrowths from ISO 26262, like ISO/PAS 21448 more commonly referred to as SOTIF (safety of the intended functionality). SOTIF helps you analyze and prevent the misuse of the intended functionality where it creates an unsafe scenario. For example, your vehicle inadvertently shuts down while you're driving it, due to an initiated software update.

Security vulnerabilities also pose unsafe scenarios. An attacker could use the car's Wi-Fi connection to remotely exploit an exposed port. They could somehow work their way from the advanced in-vehicle infotainment (IVI) into taking control of, or influencing, safety-critical components like braking or steering due to sharing the same communications infrastructure.

THE ROLE OF STANDARDS

Standards like SAE J3061, superseded by ISO/SAE 21434, specify that an initial Threat Analysis and Risk Assessment (TARA) be completed to assess potential threats related to operation, privacy, and other factors where a road user/driver can be impacted. If the risk for any threat is sufficiently high, then a cybersecurity process is necessary. There are various approaches to flushing out security vulnerabilities and requirements that mitigate the risks. Learn more about TARA and why your development team needs TARA.

Standards like UL 4600 now exists specifically for fully autonomous vehicle operation. This means that there is no human supervision, and the autonomy assumes full responsibility. This standard focuses on building a safety case for the deployment of SAE Level 4/5 vehicles, not on how to test safety of autonomous vehicles on public roads. That would involve a different standard.

These standards and others play a crucial role in safety and security for the automotive industry. OEMs carry the liability costs for delivering unsafe and insecure vehicles to the masses. To mitigate these risks, OEMs need to adopt and adhere to these standards. However, OEMs should mandate the same quality and adherence by their suppliers. A weakness in one component can undermine the safety and security of the entire system.

BUILDING CUSTOM CODING STANDARDS

Working with some of its automotive OEMs, Parasoft has built custom coding standards that incorporate MISRA, AUTOSAR C++14, CERT, CWE, and other custom rules to be used by their suppliers. This ensures that the same level of quality software exists across the entire supply chain.

Parasoft C/C++test is a unified testing solution that includes unit testing and structural code coverage as part of its functionality. This solution for C/C++ software development supports a comprehensive set of hardware targets and development ecosystems that suppliers and OEMs can use with varying development infrastructures. Parasoft C/C++test has been certified by TÜV SÜD for use on safety- and security-critical systems. For ADAS and secure connected cards, C/C++test's seamless integration with Parasoft SOAtest and Parasoft Virtualize combines API testing with runtime application coverage and simulated virtual test beds.

WHAT IS ISO 26262?

ISO 26262 is a functional safety standard that covers the entire automotive product development process. It includes activities such as requirements specification, design, implementation, integration, verification, validation, and configuration.

The standard provides guidance on automotive safety lifecycle activities by specifying the following requirements:

- Functional safety management for automotive applications
- The concept phase for automotive applications
- Product development at the system level for automotive applications software architectural design
- Product development at the hardware level for automotive applications software unit testing
- Product development at the software level for automotive applications
- Production, operation, service, and decommissioning
- Supporting processes: interfaces within distributed developments, safety management requirements, change and configuration management, verification, documentation, use of software tools, qualification of software components, qualification of hardware components, and proven-in-use argument
- Automotive Safety Integrity Level (ASIL) oriented and safety-oriented analyses

ISO 26262 is an adaptation of IEC 61508 for the automotive industry. IEC 61508 is a basic functional industrial safety standard for electrical, electronic, and programmable electronic devices, and applicable to all kinds of industries. Other sectors like Medical IEC 62304, Railway EN 50128 have also been derived from IEC 61508.

Since ISO 26262 has been extracted and expanded from IEC 61508 for the automotive industry, by inheritance it is a functional safety standard that provides guidance for regulating the entire product lifecycle process, at the software and hardware level from conceptual development through to decommissioning. It covers electrical and electronic automotive systems and their development process, including requirements specification, design, implementation, integration, verification, validation, and configuration.

The latest release, ISO 26262:2018 is subdivided into 12 parts. The standard has been evolving since its first edition, released back in 2018.

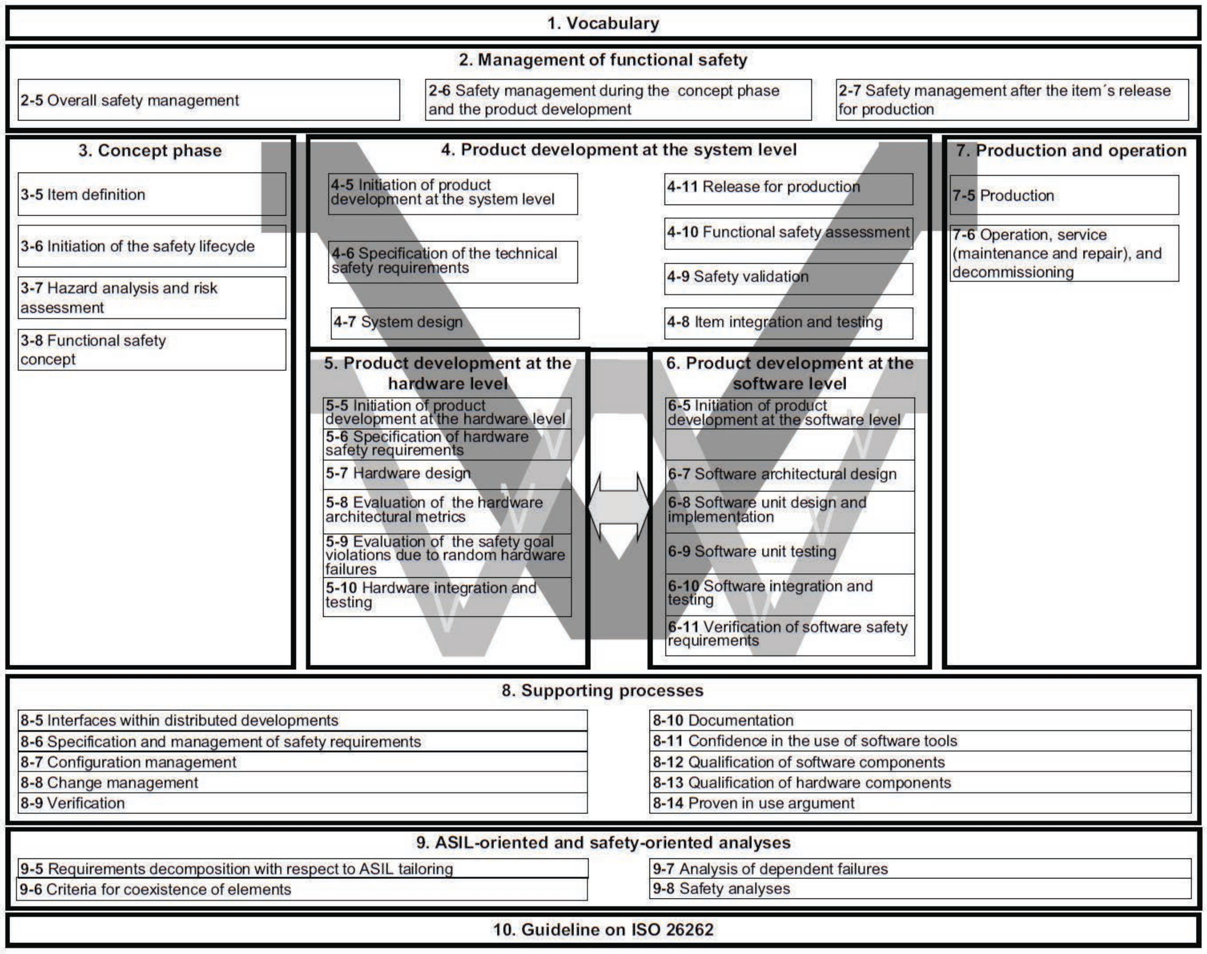

1. Vocabulary
2. Management of functional safety
2-5 Overall safety management
2-6 Safety management during the concept phase and the product development
2-7 Safety management after the item´s release for production
3. Concept phase
3-5 Item definition
3-6 Initiation of the safety lifecycle
3-7 Hazard analysis and risk assessment
3-8 Functional safety concept
4. Product development at the system level
4-5 Initiation of product development at the system level
4-6 Specification of the technical safety requirements
4-7 System design
4-11 Release for production
4-10 Functional safety assessment
4-9 Safety validation
4-8 Item integration and testing
7. Production and operation
7-5 Production
7-6 Operation, service (maintenance and repair), and decommissioning
5. Product development at the hardware level
5-5 Initiation of product development at the hardware level
5-6 Specification of hardware safety requirements
5-7 Hardware design
5-8 Evaluation of the hardware architectural metrics
5-9 Evaluation of the safety goal violations due to random hardware failures
5-10 Hardware integration and testing
6. Product development at the software level
6-5 Initiation of product development at the software level
6-7 Software architectural design
6-8 Software unit design and implementation
6-9 Software unit testing
6-10 Software integration and testing
6-11 Verification of software safety requirements
8. Supporting processes
8-5 Interfaces within distributed developments
8-6 Specification and management of safety requirements
8-7 Configuration management
8-8 Change management
8-9 Verification
8-10 Documentation
8-11 Confidence in the use of software tools
8-12 Qualification of software components
8-13 Qualification of hardware components
8-14 Proven in use argument
9. ASIL-oriented and safety-oriented analyses
9-5 Requirements decomposition with respect to ASIL tailoring
9-6 Criteria for coexistence of elements
9-7 Analysis of dependent failures
9-8 Safety analyses
10. Guideline on ISO 26262

Figure 2-1: Overview of ISO 26262

Part 1 is the vocabulary section for the standard. Terms, definitions, and abbreviations are found here.

Part 2 is the management of functional safety, which defines an internal functional safety process for the team or company. This includes having a safety organization that oversees the planning, coordinating and documentation activities related to functional safety.

Functional safety is of the utmost importance in the development of safety-critical automotive systems because people's lives depend on it. Especially now with the introduction of driver assist and automated driving systems. The management of security could be adapted to part 2. Security is crucial in the world we live in today.

Part 3 is the concept phase that takes in the stakeholder requirements and drives what you are going to build and ultimately deliver. In figure 2-1, notice on the right side of the concept phase box, the beginning of a grey shaded V watermark. The shaded Vs represent the interconnection among parts 3, 4, 5, 6, and 7 of the standard. These part series are based upon the V-model software development lifecycle. You have the different phases of development represented on the left and the verification and validation or testing phases on the right. If you are a systems or software engineer in the embedded industry, the V-model is well known.

Part 4 is the beginning of product development at the system level, which includes parts 5 and 6 but looking at these from a high level of abstraction. The architecture is defined, including functional testcases that verify and validate the architecture. To dive in deeper into the detail design and implementation, part 5 and part 6 are defined.

Part 5 targets development of hardware, which is out of scope for this document.

Part 6 targets software development. You can see a smaller lighter grey V watermark for software development and again the left-hand side of the V encapsulates the requirements decomposition, design, and implementation phases but now a much lower level of granularity. On the right-hand side of the V, sections 6.9, 6.10, and 6.11 represent the testing or verification and validation of the software. This includes unit testing, static analysis, structural code coverage, requirements traceability and more.

It also includes requirements for the software development of automotive applications. This includes obligations for initiation of product development, specification of software safety requirements, software architectural design, software unit design and implementation. On the verification and validation of the software component, you have multiple methods recommended or mandated based on the assigned safety integrity level (ASIL).

Part 7 address the production and operation of the product, once it's out in the field. This means you must consider things like maintenance and decommissioning or sunsetting of your product.

Part 8 specifies the various supporting processes and solutions needed in the development of the system that help ensure functional safety. This includes having a configuration management solution, a change management, a documentation management, and other solutions in place.

Another important aspect of Part 8 is the qualification of the software tools being used. You don't want to use an open source tool or an uncertified tool from a vendor that undermines the safety or security of your product by introducing issues. Use a tool that has been certified by the Technical Inspection Association (TÜV) and has a proven in-use history or argument.

Part 9 is a critical section to understand because it pertains to assigning a risk classification on the system under development. This means that you have to take into consideration the risk to the passengers or pedestrians if the electrical or electronic system in development were to malfunction or fail.

A hazard analysis and risk assessment need to be performed. ISO 26262 is a risk-based safety standard, where the risk of hazardous operational situations is assessed, and safety measures are defined to detect and to avoid or control failures, so mitigating actions can take effect.

Part 10 basically provides an overview of the ISO 26262 standard with additional explanations that enhance the understanding and concepts of the other parts in the standard, so its informative.

Part 11 is the adaptation of functional safety guidelines to semiconductors for automotive. It offers guidance and information to semiconductor manufacturers on how to develop ISO 26262 compliant IP. It helps incorporate functional safety because users of semiconductors may not know how to use the semiconductor safely. This came about because automotive systems have become very complex and semiconductors have enabled most of the recent innovations. That includes vision-based technology, enhanced graphics processing units (GPUs), application processors, sensors, DRAM, and other components that empower advanced driver-assistance systems or ADAS.

Part 12 is the adaptation of the standard for motorcycles, which has been intentionally left out of Figure 2-1 and this ebook.

PERFORMING HAZARD ANALYSIS AND RISK ASSESSMENT

In ISO 26262, a hazard analysis and risk assessment (HARA) needs to be performed on the system under development. Upon completion of the HARA an ASIL is assigned to the software component and there are levels A through D. Level A representing the lowest hazard assignment and Level D representing the highest hazard assignment. Meaning that the failure of a system with ASIL D assignment could be catastrophic.

There is also a quality management (QM) level assignment, which means that there is no safety requirement. ASIL is assigned by taking the severity of the injury times the probability of the failure times the controllability. The following table spells out each level for severity, exposure, and controllability.

SEVERITY	EXPOSURE	CONTROLLABILITY
S0 No Injuries	**E0** Incredibly unlikely	**C0** Controllable in general
S1 Light to moderate injuries	**E1** Very low probability (injury could happen only in rare operating conditions)	**C1** Simply controllable
S2 Severe to life-threatening (survival probable) injuries	**E2** Low probability	**C2** Normally controllable (most drivers could act to prevent injury)
S3 Life-threatening (survival uncertain) to fatal injuries	**E3** Medium probability	**C3** Difficult to control or uncontrollable
	E4 High probability (injury could happen under most operating conditions)	

Figure 2-2: Hazard Analysis and Risk Assessment

Severity = What would be the impact or damage if the failure occurred?
Exposure = The frequency or probability that the failure would occur.
Controllability = The extent to which we can ensure that the event doesn't happen.

There are several tables freely made available that provide help in determining the ASIL value. The table below is an example of one that's much easier to read and shows the ASIL levels in colors based on severity, exposure, and controllability.

CONTROLLABILITY	EXPOSURE	SEVERITY			
		S0	S1	S2	S3
C1	E1	QM	QM	QM	QM
	E2	QM	QM	QM	QM
	E3	QM	QM	QM	A
	E4	QM	QM	A	B
C2	E1	QM	QM	QM	QM
	E2	QM	QM	QM	A
	E3	QM	QM	A	B
	E4	QM	A	B	C
C3	E1	QM	QM	QM	A
	E2	QM	QM	A	B
	E3	QM	A	B	C
	E4	QM	B	C	D

Figure 2-3: Simplified ASIL assesment table

Active and Passive Safety

Roadside vehicles come with lots of safety systems and some are considered active safety and others passive safety.

Active safety is used to refer to technology assisting in the prevention of a crash or accident. You have your traction control, anti-lock braking system, vision ADAS, and others.

Passive safety systems are to keep the passengers safe. For example, in case of a crash, you have airbags, and seatbelts. The electronic windshield wiper and instrument cluster are also passive safety systems.

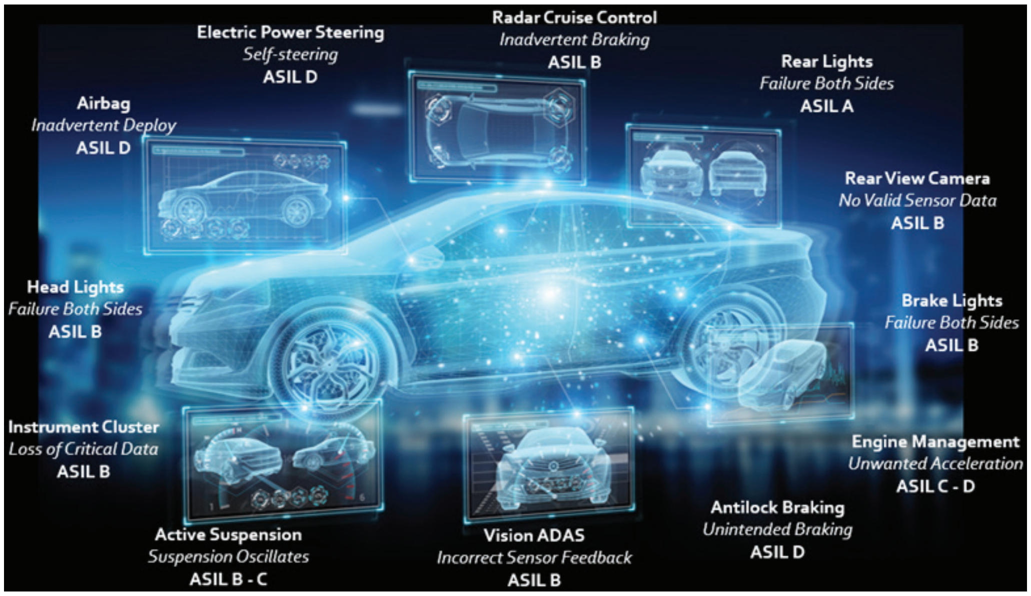


Figure 2-4: Active and Passive Safety

PERFORMING TEST VERIFICATION & VALIDATION OF SOFTWARE UNIT DESIGN AND IMPLEMENTATION

Since the focus of this ebook is software, it's important to cover the test verification and validation methods recommended by the standard. For example, Table 9 captures verification methods 1a through 1h to be applied during unit design and implementation. Method 1f, "Static code analysis" is recommended for ASIL level A and highly recommended for ASIL levels B through D.

The columns on the right in Table 9 shows the A to D ASIL levels. A single "+" symbol represents recommended by the standard and "++" representing highly recommended.

Table 9 - Methods for the verification of software unit design and implementation

METHODS		ASIL			
		A	B	C	D
1a	Walk-through[a]	++	+	o	o
1b	Inspection[a]	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis[b,c]	+	+	++	++
1f	Data flow analysis[b,c]	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis[d]	+	+	+	+

[a] In the case of model-based software develeopment the software unit specification design and implementation can be verified at the model level.
[b] Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code develeopment and to model-based development.
[c] Methods 1e and 1f can be part of methods 1d, 1g or 1h.
[d] Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

Figure 2-5: ISO 26262 Part 6, 8.4.5:2011

++	Highly Recommended
+	Recommended
o	No Recommendation

Other key methods of verification are done through dynamic analysis, for requirements-based testing and fault injection. Table 11 for example has "Analysis of boundary values". This is a method for deriving test case to flush out defects by means of proving inputs into the unit that are not just the min, mid, and max, but the boundaries outside the scope of its range, to see if the unit is robust enough to handle these outlier cases.

Table 11 - Methods for deriving test cases for software unit testing

METHODS		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes[a]	+	++	++	++
1c	Analysis of boundary values[b]	+	++	++	++
1d	Error guessing[c]	+	+	+	+

[a] Equivalence classes can be identified based on the division of inputs and outputs, such that a representative test value can be selected for each class.
[b] This method applies to interfaces, values approching and crossing the boundaries and out of range values.
[c] Error guessing tests can be based on data collected through a "lessons learned" process and expert judfement.

Figure 2-6:
ISO 26262 Part 6,
9.4.4:2011

And Table 12 lists the recommended structural code coverage metrics to ensure test coverage, flush out dead code, and hidden defects.

Table 12 - Structural coverage metrics at the software level

METHODS		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage	+	+	+	++

Figure 2-7:
ISO 26262 Part 6,
9.4.5:2011

Requirements for Compliance in Testing

STATIC ANALYSIS

Many of the quality tasks specified in ISO 26262, including data and control flow analysis and semantic analysis are supported by modern advanced tools like Parasoft C/C++test. In addition, static analysis tools include metrics and support peer code review with capabilities that assist unit testing and runtime error detection.

THE ROLE OF STATIC ANALYSIS IN ISO 26262 SOFTWARE VERIFICATION

Verification methods like static analysis provide teams a practical way to expose, prevent, and correct errors in automotive software systems. The real power from advanced static analysis tools comes from the ability to analyze the code based on industry coding compliance standards like MISRA, CERT and AUTOSAR.

Not only does the analysis report include code rules and directive violations, but also code complexity, quality metrics. This data can be source controlled for historical and auditing purposes. Equally important is the use of a defect tracking and managing system to provide meaningful analytical views and prioritization for the intent of solving the highest risk issues down to the lowest.

Table 9 - Methods for the verification of software unit design and implementation

METHODS		ASIL			
		A	B	C	D
1a	Walk-through[a]	++	+	o	o
1b	Inspection[a]	+	++	++	++
1c	Semi-formal verification	+	+	++	++
1d	Formal verification	o	o	+	+
1e	Control flow analysis[b,c]	+	+	++	++
1f	Data flow analysis[b,c]	+	+	++	++
1g	Static code analysis	+	++	++	++
1h	Semantic code analysis[d]	+	+	+	+

[a] In the case of model-based software develeopment the software unit specification design and implementation can be verified at the model level.
[b] Methods 1e and 1f can be applied at the source code level. These methods are applicable both to manual code develeopment and to model-based development.
[c] Methods 1e and 1f can be part of methods 1d, 1g or 1h.
[d] Method 1h is used for mathematical analysis of source code by use of an abstract representation of possible values for the variables. For this it is not necessary to translate and execute the source code.

Figure 3-1:
ISO 26262 Part 6,
8.4.5:2011

The specific sections of the ISO 26262, part 6: Product development: software level that are addressed by static analysis tools are described below.

Walkthroughs and Inspections

Informal methods used to verify design and implementation. Static analysis tools automate much of the tedious aspects of code inspection such as coding standards compliance while flagging errors and possible software weaknesses.

Control flow Analysis

A static code analysis technique for determining the control flow of a program. Modern advanced static analysis tools, such Parasoft C/C++test, use sophisticated control and data flow analysis to detect complex defects and security vulnerabilities.

Data Flow Analysis

A technique for gathering information about the possible set of values calculated at various points in a computer program. Data flow analysis is a critical aspect of advanced static analysis tools that helps detect complex errors such as tainted data vulnerabilities.

Static Code Analysis

The general term used to describe the analysis of code that is performed without actual code execution. This includes the terms used above.

Semantic Code Analysis

An analysis of the code performed during compile time where semantic information is gathered to perform type checking. The analysis judges whether the syntax structure constructed in the code derive any meaning or not. For example, making sure variables are declared before used.

THE ROLE OF STATIC ANALYSIS TOOLS IN SUPPORT OF ISO 26262 DESIGN PRINCIPLES FOR SOFTWARE UNIT DESIGN AND IMPLEMENTATION

Coding standards embody the best practices learned from years of experience and aim to harden code by avoiding bad practices that result in inadequate quality and security while promoting good practices that create more resilient code. In the case of automotive standards, they are based on best practices plus guidance on preventing the types of software failures that have been observed over the years.

Coding standards usually define a subset of a programming language deemed safer and more secure to use. The aim of this is to prevent unpredictable behavior in the first place, limiting the risky language features that make them possible.

The only practical, objective, and sustainable way to enforce coding standards is with static code analysis tools, which can automatically analyze enormous amounts of source code at a time. These tools integrate into software builds in a CI/CD pipeline and are available directly in a developer's IDE. And they provide reports indicating the conformance of analyzed software to the standard selected.

The following sections cover the important industry standards in the automotive software industry and how automation, tools, and processes can be leveraged to ease compliance.

MISRA C 2023

MISRA C is a set of coding guidelines for the C programming language. 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.

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.

MISRA C Compliance

In the document, "MISRA Compliance:2020," the MISRA Consortium provides the response needed by the community with a well-defined framework of what the statement, “MISRA Compliant,” truly means.

The document helps organizations use a common language articulating the compliance requirements by defining the following artifacts:

The Guideline Enforcement Plan

Demonstrates how each MISRA guideline is verified.

The Guideline Re-Categorization Plan

Communicates the agreed upon severity of individual rules in the guidelines as part of the vendor/client relationship.

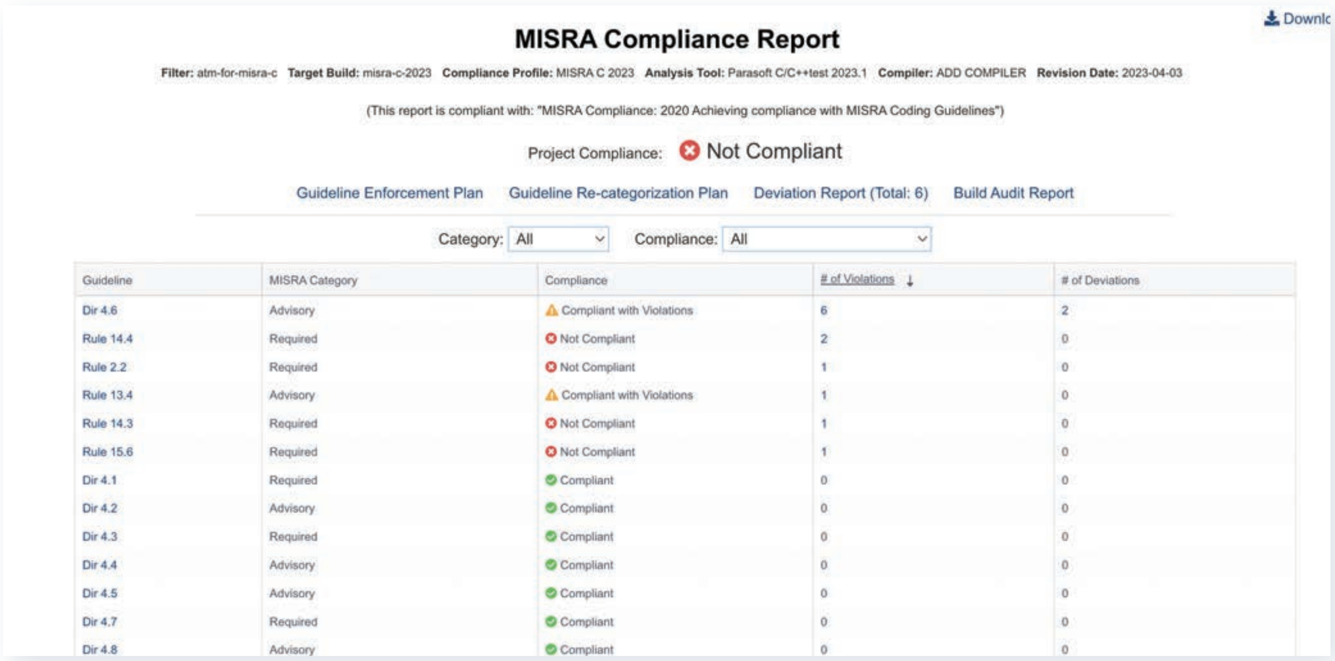
The Deviations Report

Documents the violations of guidelines with appropriate rationale.

The Guidelines Compliance Summary

This is the primary record of overall project compliance.

When first introducing MISRA C into a project, commonly where code already exists, the key document is the guideline re-categorization plan. This document captures all directives, rules, and identifies which categories have been re-categorized. However, it's important to have the same rational categorization for newly developed code as well. For example, the following diagram shows part of a re-categorization plan.



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

Figure 3-2: MISRA Compliance Report

The "MISRA C 2023" compliance document recommends against re-categorizations from a less stringent to a more stringent classification. In addition, it is possible to disapply advisory rules all together after reviewing the types of violations with the team.

The requirement to document deviations is only necessary for all required rules. Any violations in adopted code should be reviewed. Deviations need to clearly state that violations do not compromise safety and security. Regardless of recategorization, if there is a finding that compromises the safety or security of the system, the issue must be fixed. Also, modifications to the existing code may introduce other issues not clearly seen by the developer.

AUTOSAR C++14

AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of OEM manufacturers, Tier 1 automotive suppliers, semiconductor manufacturers, software suppliers, tool suppliers, and others that focus on establishing and standardizing automotive software architecture.

Adaptive AUTOSAR defines a platform for developing automotive control units, which provide sophisticated functionalities like advanced driving assistance systems, media streaming, or software updates via internet. The platform contains the specification of interfaces that define services and APIs for building modern automotive systems.

A key component of AUTOSAR Adaptive Platform is AUTOSAR C++14 coding standard that defines guidelines for the use of the modern C++ language in critical and safety-related systems. This is the only standard on the market that supports modern C++. The standard is well documented and provides traceability to the other existing C++ standards, such as HIC++ 4.0, JSF, SEI CERT C++, C++ Core Guidelines, and to MISRA C++ 2008.

In Jan 2019, MISRA consortium announced the integration of AUTOSAR C++ 14 coding guidelines with MISRA. The MISRA Working Group will continue releasing new C++ industry standards and incorporate the latest version of C++ language, C++ 17, and its successor, C++20, later.

Parasoft C/C++test, a unified testing tool for C/C++ development, and Parasoft DTP, a reporting and analytics dashboard, provide a comprehensive solution to help organizations overcome the challenges associated with automotive software quality and compliance. Parasoft C/C++test and Parasoft DTP support AUTOSAR C++ 14 (19.03) with the most comprehensive coverage of static analysis checkers for the AUTOSAR C++14 guidelines and unique compliance reporting.

AUTOSAR C++14 Compliance

AUTOSAR C++14 does not provide explicit guidance on the process of achieving compliance. However, given that AUTOSAR guidelines are based on MISRA C++ 2008, it's reasonable to refer to the MISRA compliance guidelines, discussed above, for guidance on achieving compliance.

The desired situation is to have a static analysis tool that covers as many guidelines as possible. The rules that cannot be enforced with static analysis will require manual reviews, which are expensive.

As with MISRA C compliance, a deviation handling procedure needs to be established. The deviation procedure formalizes the steps that need to be taken when development needs to deviate from a specific guideline. As MISRA prescribes it, it is expected that "...the procedure will be based around obtaining a sign-off for every deviation or class of deviations."

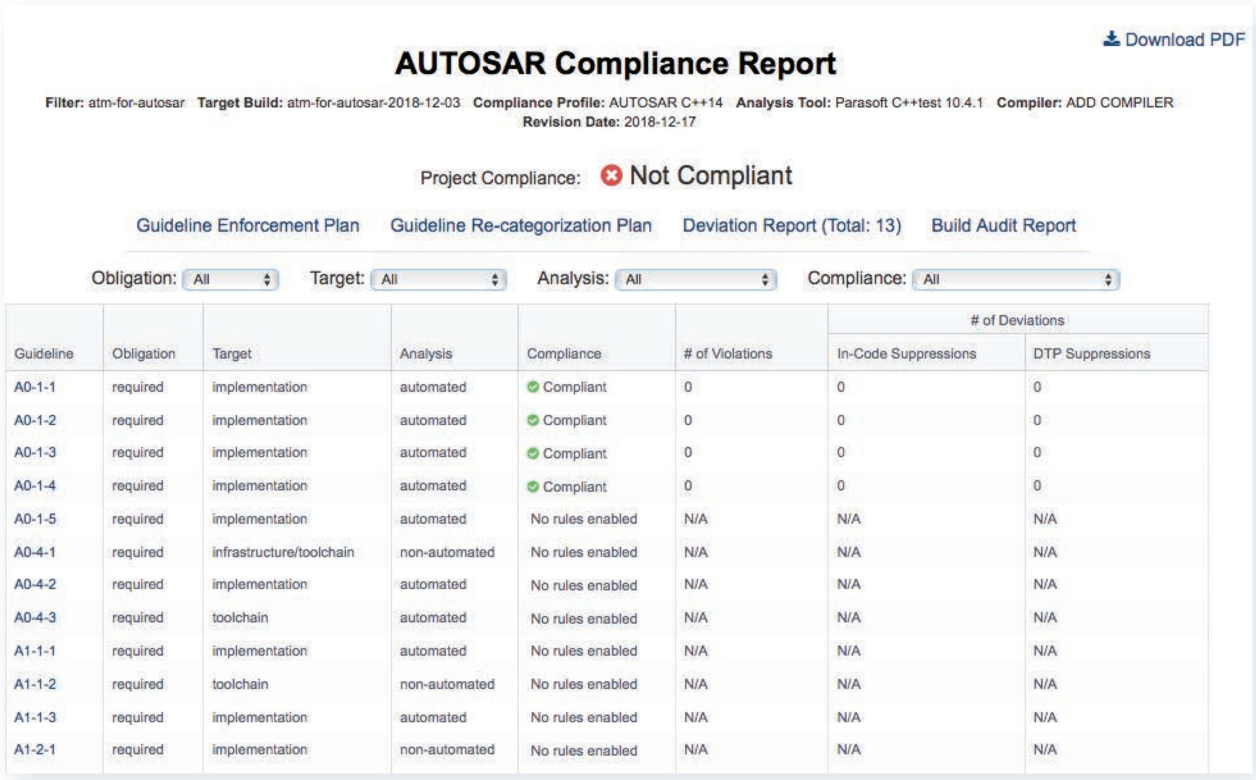
This is a particularly important piece of the puzzle. It prevents abusing the deviation concept by developers deviating at will. Effectively, you'll need formal tickets stored in your system that document every deviation in the source code. The same pertains to the compliance matrix and any additional configurations and process descriptions created to enforce compliance. MISRA C++2008 is truly clear here and requires "formalization within quality system."

Finally, if all the procedures described in MISRA C++ 2008 point 4.3 are in place for AUTOSARC++14, we can claim compliance by demonstrating:

» A compliance matrix has been completed that shows how compliance has been enforced.

» All the C++ code in the product is compliant with the rules of AUTOSARC++ 14 document. If not, they are documented deviations.

» A list of all instances of the rules not being followed is maintained and for each instance there is an appropriately signed-off deviation.



Download PDF

AUTOSAR Compliance Report

Filter: atm-for-autosar **Target Build:** atm-for-autosar-2018-12-03 **Compliance Profile:** AUTOSAR C++14 **Analysis Tool:** Parasoft C++test 10.4.1 **Compiler:** ADD COMPILER
Revision Date: 2018-12-17

Project Compliance: ✗ Not Compliant

Guideline Enforcement Plan | Guideline Re-categorization Plan | Deviation Report (Total: 13) | Build Audit Report

Obligation: All Target: All Analysis: All Compliance: All

						# of Deviations	
Guideline	Obligation	Target	Analysis	Compliance	# of Violations	In-Code Suppressions	DTP Suppressions
A0-1-1	required	implementation	automated	✓ Compliant	0	0	0
A0-1-2	required	implementation	automated	✓ Compliant	0	0	0
A0-1-3	required	implementation	automated	✓ Compliant	0	0	0
A0-1-4	required	implementation	automated	✓ Compliant	0	0	0
A0-1-5	required	implementation	automated	No rules enabled	N/A	N/A	N/A
A0-4-1	required	infrastructure/toolchain	non-automated	No rules enabled	N/A	N/A	N/A
A0-4-2	required	implementation	automated	No rules enabled	N/A	N/A	N/A
A0-4-3	required	toolchain	automated	No rules enabled	N/A	N/A	N/A
A1-1-1	required	implementation	automated	No rules enabled	N/A	N/A	N/A
A1-1-2	required	toolchain	non-automated	No rules enabled	N/A	N/A	N/A
A1-1-3	required	implementation	automated	No rules enabled	N/A	N/A	N/A
A1-2-1	required	implementation	non-automated	No rules enabled	N/A	N/A	N/A

Figure 3-3: AUTOSAR Compliance Report

A formal process for handling deviations must document enforcement methods for every applicable guideline. This document is called the Guidelines Enforcement Plan (GEP). Also, a Guidelines Recategorization Plan (GRP) is needed, which documents in a formal way any changes that are introduced to rule categories. And the Guidelines Compliance Summary (GCS) is a final artifact from the compliance process that presents the level of compliance that was achieved for every guideline.

Support for AUTOSAR C++14 in Parasoft C/C++test

The only practical way to enforce compliance with a coding standard like AUTOSAR C++14 is with a static analysis tool, like Parasoft C/C++test, a code quality tool supporting multiple testing technologies. Parasoft C/C++test support for AUTOSAR C++14, provides a set of built-in checkers (rules) for verifying compliance with standards including MISRA C 2023, MISRA C++ 2023, JSF AV C++, SEI CERT C/C++, HIC++, CWE Top 25, CWE On the Cusp, OWASP, and more.

Parasoft compliance packs provide users with standard specific configurations, automatic generation of compliance documentation, risk assessment framework, and dynamic compliance reporting dashboards (DTP) to help stakeholders easily aggregate, correlate, and apply analytics to centralize reporting for each step along the complex software supply chain.

OBLIGATION LEVEL	ENFOCEMENT BY STATIC ANALYSIS	SUPPORTED	UNSUPPORTED	ALL	PERCENT COVERAGE
Required	Automated	301	0	301	100%
Required	Partially	18	2	20	90%
Required	Non-Automated	19	22	41	46%
Required	All	338	24	362	93%

Figure 3-4: Parasoft covers 100% of all required and automated AUTOSAR rules.

With the 100% coverage for Required & Automated rules, Parasoft testing tool suite ensures AUTOSAR compliance throughout the software development life cycle, improves code quality, and reduces cost associated with resources and time to market.

Parasoft provides comprehensive support for CERT C and CERT C++ secure coding standards with complete coverage of all the CERT C/C++ guidelines including both 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.

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 form 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: rules and 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.

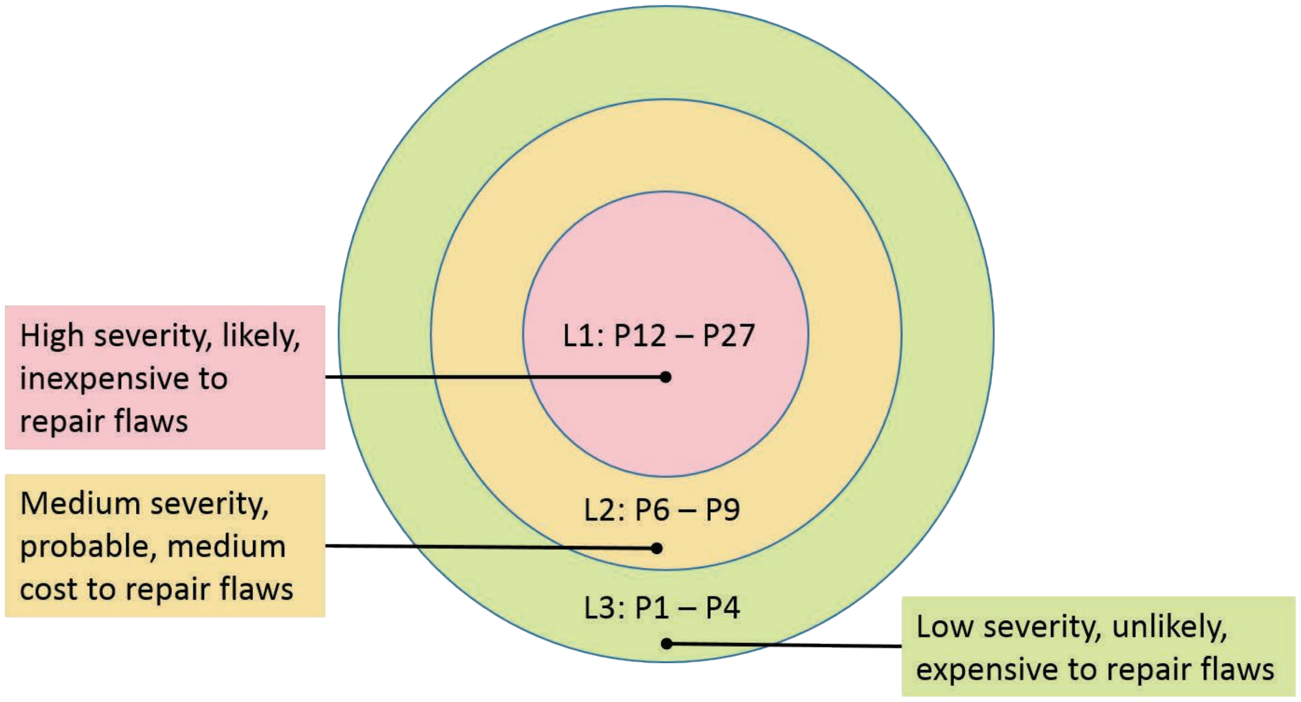

High severity, likely, inexpensive to repair flaws
L1: P12 – P27
Medium severity, probable, medium cost to repair flaws
L2: P6 – P9
L3: P1 – P4
Low severity, unlikely, expensive to repair flaws

Figure 3-5: SEI CERT severity bullseye diagram

SEI CERT C/C++ Conformance

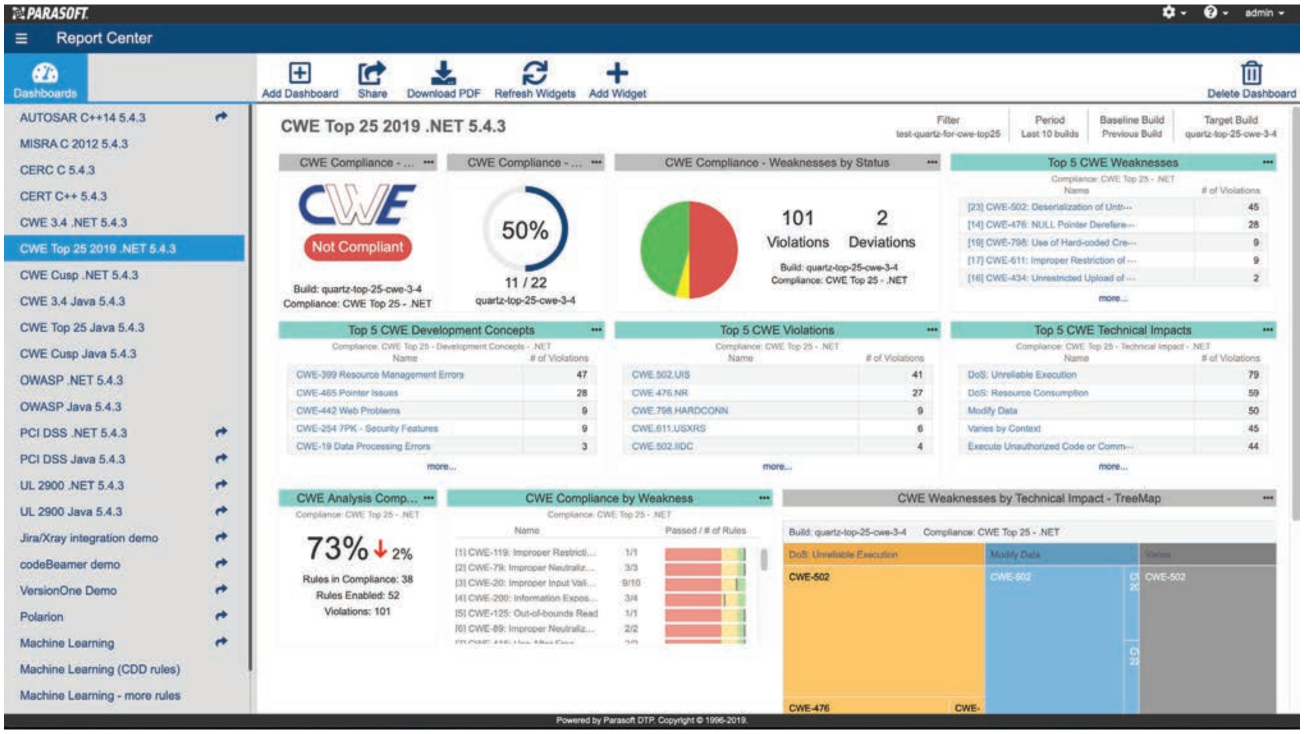
According the SEI CERT C documentation, conformance "requires that the code not contain any violations of the rules specified in this standard. If an exceptional condition is claimed, the exception must correspond to a predefined exceptional condition, and the application of this exception must be documented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

Although conformance is less specific than standards such as MISRA, the principles remain similar. Rules should be followed and deviations rare and well documentation. Recommendations should be used when possible and those that aren’t needed to 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 provided 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. Also, a conformance test plan is available to correlate the CERT guideline with the appropriate Parasoft static analysis checker is an important tool if conformance documentation is needed for audit purposes. In addition, all the interesting reports as specified by the team are available in a single PDF available for download for auditors.



CWE Top 25 2019 .NET 5.4.3

Filter	Period	Baseline Build	Target Build
test-quartz-for-cwe-top25	Last 10 builds	Previous Build	quartz-top-25-cwe-3-4

CWE Not Compliant
Build: quartz-top-25-cwe-3-4
Compliance: CWE Top 25 - .NET

50%
11 / 22
quartz-top-25-cwe-3-4

101 Violations 2 Deviations
Build: quartz-top-25-cwe-3-4
Compliance: CWE Top 25 - .NET

Top 5 CWE Weaknesses
Compliance: CWE Top 25 - .NET

Name	# of Violations
[23] CWE-502: Deserialization of Untr···	45
[14] CWE-476: NULL Pointer Derefere···	28
[19] CWE-798: Use of Hard-coded Cre···	9
[17] CWE-611: Improper Restriction of ···	9
[16] CWE-434: Unrestricted Upload of ···	2

more...

Top 5 CWE Development Concepts
Compliance: CWE Top 25 - Development Concepts - .NET

Name	# of Violations
CWE-399 Resource Management Errors	47
CWE-465 Pointer Issues	28
CWE-442 Web Problems	9
CWE-254 7PK - Security Features	9
CWE-19 Data Processing Errors	3

more...

Top 5 CWE Violations
Compliance: CWE Top 25 - .NET

Name	# of Violations
CWE.502.UIS	41
CWE.476.NR	27
CWE.798.HARDCONN	9
CWE.611.USXRS	6
CWE.502.IIDC	4

more...

Top 5 CWE Technical Impacts
Compliance: CWE Top 25 - Technical Impact - .NET

Name	# of Violations
DoS: Unreliable Execution	79
DoS: Resource Consumption	59
Modify Data	50
Varies by Context	45
Execute Unauthorized Code or Comm···	44

more...

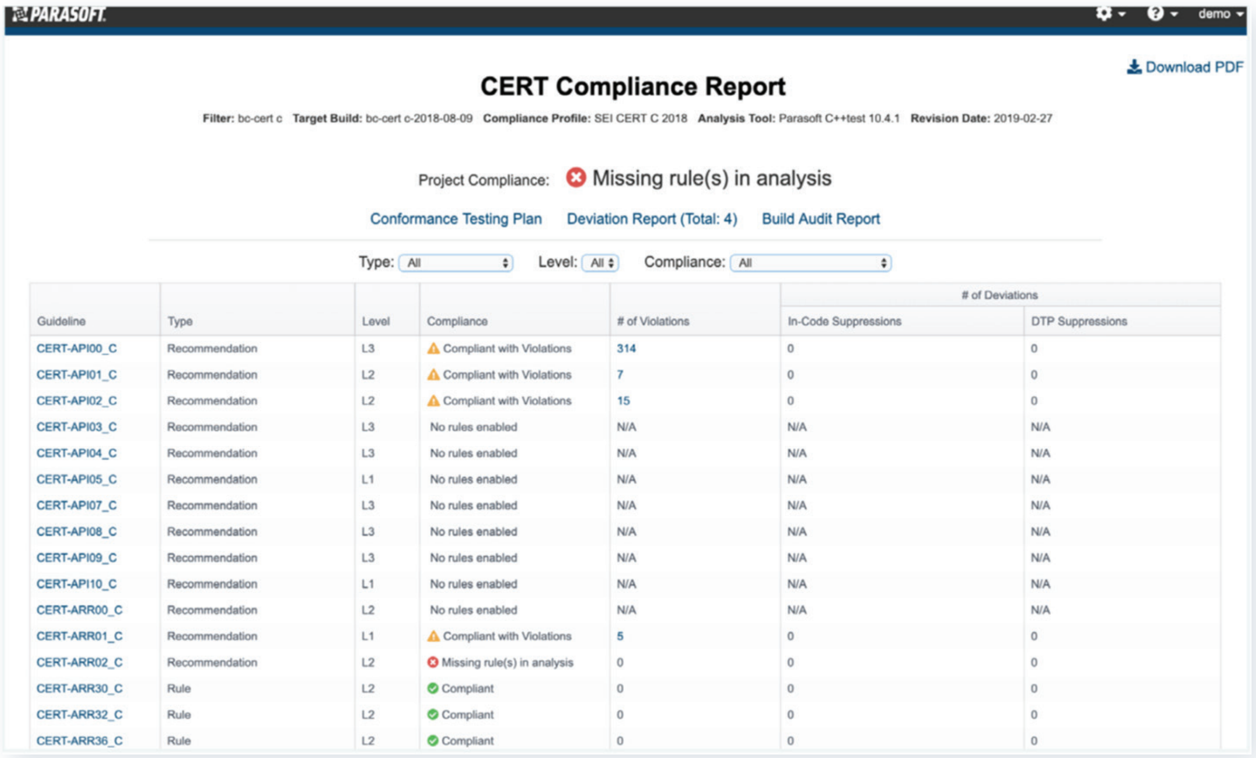
CWE Analysis Comp...
Compliance: CWE Top 25 - .NET
73% ↓ 2%
Rules in Compliance: 38
Rules Enabled: 52
Violations: 101

CWE Compliance by Weakness
Compliance: CWE Top 25 - .NET

Name	Passed / # of Rules
[1] CWE-119: Improper Restricti...	1/1
[2] CWE-79: Improper Neutraliz...	3/3
[3] CWE-20: Improper Input Vali...	9/10
[4] CWE-200: Information Expos...	3/4
[5] CWE-125: Out-of-bounds Read	1/1
[6] CWE-89: Improper Neutraliz...	2/2

Build: quartz-top-25-cwe-3-4 Compliance: CWE Top 25 - .NET

Figure 3-6:
CWE Top 25 compliace dashboard



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 3-7:
CERT Compliance Report

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 both 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.

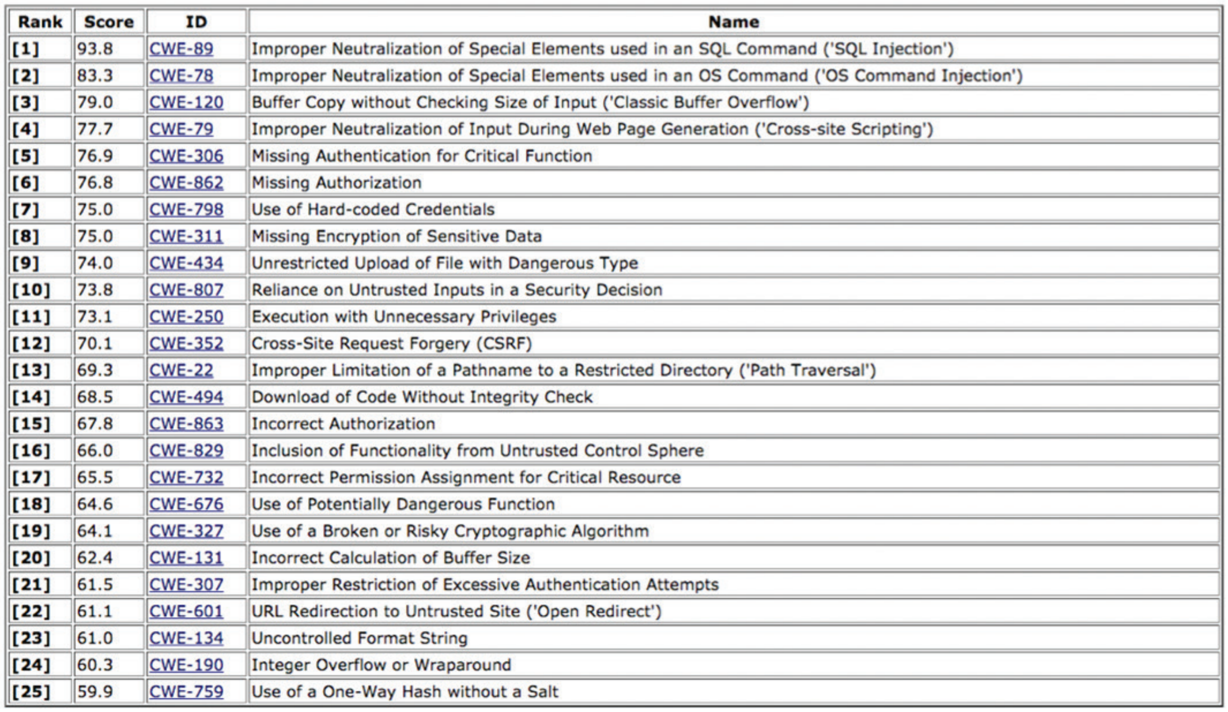
CWE - COMMON WEAKNESS ENUMERATION

CWE is a list of discovered software weaknesses based on the analysis of reported vulnerabilities (CVEs). The collection of CVEs and CWEs is a US government funded initiative developed by the software community and managed by the MITRE organization. In its entirety, the CWE list contains over 800 items.

These 800+ 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 is 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 2011. An updated Top 25 is currently in process with improved linking to CVEs and the NVD. Ranking considers real world 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.

The current CWE Top 25 is listed below.



Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-306	Missing Authentication for Critical Function
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

Table 3-8: CWE Top 25

For software teams that have a good handle on the Top 25, there is 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 so on.

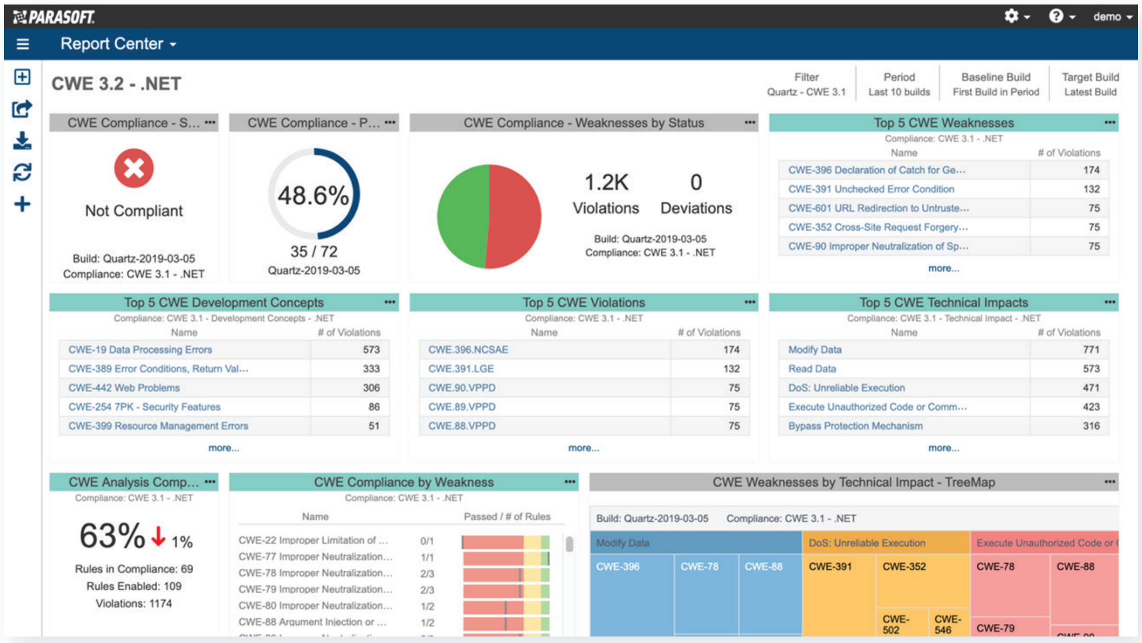
The details of these methods aren't too important, but the sorted list is useful in understanding which vulnerabilities to be concerned about the most. As an example, it's possible that your application is purely internal and DoS issues aren't critical for you. Being able to prioritize on the most important 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 is not 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 a coding standards per se, but a list of weaknesses to avoid, improving security. To be CWE compliant, a project should be able to prove that is 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 standards violations and current project status. A built in CWE Top 25 configuration is available for C, C++, .NET, and Java and has full coverage of all the 25 common weaknesses.



CWE 3.2 - .NET

Filter	Period	Baseline Build	Target Build
Quartz - CWE 3.1	Last 10 builds	First Build in Period	Latest Build

Not Compliant
Build: Quartz-2019-03-05
Compliance: CWE 3.1 - .NET

35 / 72
Quartz-2019-03-05

1.2K Violations 0 Deviations
Build: Quartz-2019-03-05
Compliance: CWE 3.1 - .NET

Top 5 CWE Weaknesses
Compliance: CWE 3.1 - .NET

Name	# of Violations
CWE-396 Declaration of Catch for Ge...	174
CWE-391 Unchecked Error Condition	132
CWE-601 URL Redirection to Untruste...	75
CWE-352 Cross-Site Request Forgery...	75
CWE-90 Improper Neutralization of Sp...	75

more...

Top 5 CWE Development Concepts
Compliance: CWE 3.1 - Development Concepts - .NET

Name	# of Violations
CWE-19 Data Processing Errors	573
CWE-389 Error Conditions, Return Val...	333
CWE-442 Web Problems	306
CWE-254 7PK - Security Features	86
CWE-399 Resource Management Errors	51

more...

Top 5 CWE Violations
Compliance: CWE 3.1 - .NET

Name	# of Violations
CWE.396.NCSAE	174
CWE.391.LGE	132
CWE.90.VPPD	75
CWE.89.VPPD	75
CWE.88.VPPD	75

more...

Top 5 CWE Technical Impacts
Compliance: CWE 3.1 - Technical Impact - .NET

Name	# of Violations
Modify Data	771
Read Data	573
DoS: Unreliable Execution	471
Execute Unauthorized Code or Comm...	423
Bypass Protection Mechanism	316

more...

CWE Analysis Comp...
Compliance: CWE 3.1 - .NET
63% ↓ 1%
Rules in Compliance: 69
Rules Enabled: 109
Violations: 1174

CWE Compliance by Weakness
Compliance: CWE 3.1 - .NET

Name	Passed / # of Rules
CWE-22 Improper Limitation of ...	0/1
CWE-77 Improper Neutralization...	1/1
CWE-78 Improper Neutralization...	2/3
CWE-79 Improper Neutralization...	2/3
CWE-80 Improper Neutralization...	1/2
CWE-88 Argument Injection or ...	1/2

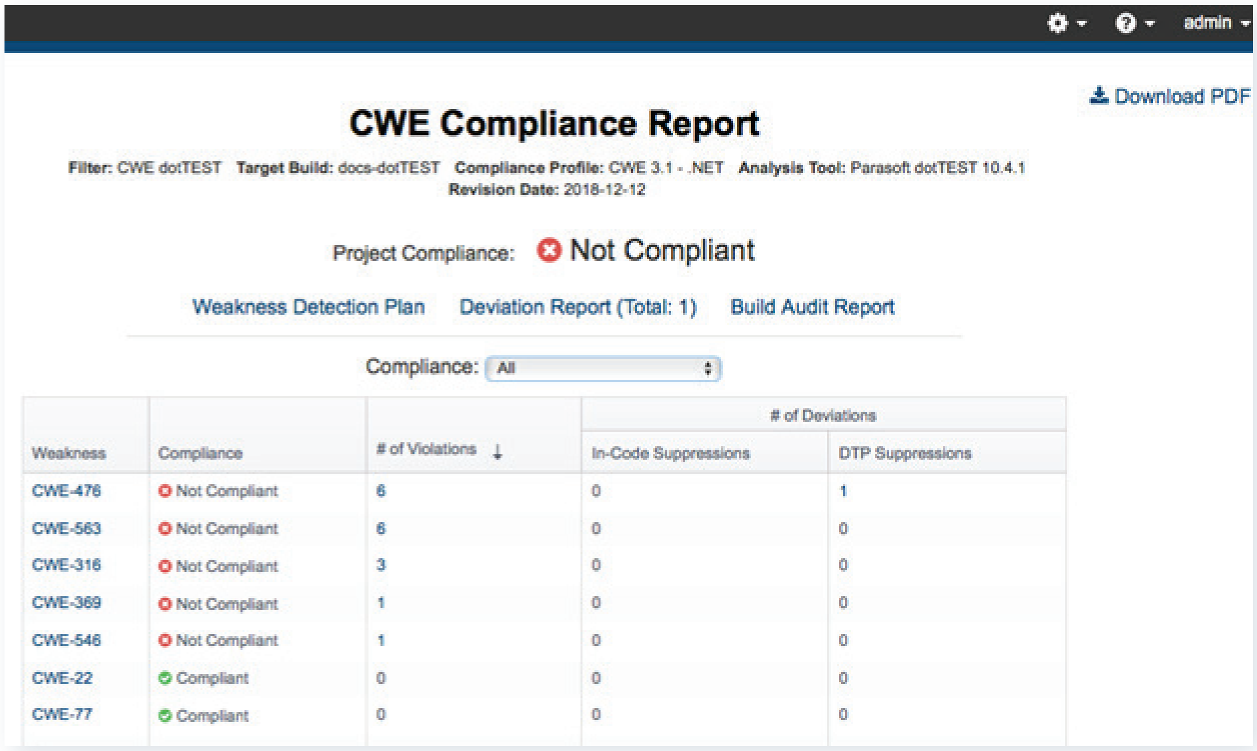
Build: Quartz-2019-03-05 Compliance: CWE 3.1 - .NET

Figure 3-9:
CWE 3.2 - .NET compliance dashboard

The Parasoft tools include information from the CWRAF risk analysis framework, 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.

The On the Cusp guidelines are also available. When enabled, they're treated the same way as the Top 25 and reports provide the same level of detail. This is useful since the UL 2900 (formerly Underwriters Laboratory) and FDA recommend the full list of guidelines (Top 25 + On the Cusp + OWASP Top 10). It's possible to integrate other guidelines from CWE lists or other standards and guidelines using Parasoft's custom checker configurations as needed.

Parasoft also supports detailed compliance reporting to streamline audit processes. The web dashboards provide the link to compliance reports that provide 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.



CWE Compliance Report

Filter: CWE dotTEST **Target Build:** docs-dotTEST **Compliance Profile:** CWE 3.1 - .NET **Analysis Tool:** Parasoft dotTEST 10.4.1
Revision Date: 2018-12-12

Project Compliance: Not Compliant

Weakness Detection Plan Deviation Report (Total: 1) Build Audit Report

Compliance: All

			# of Deviations	
Weakness	Compliance	# of Violations	In-Code Suppressions	DTP Suppressions
CWE-476	Not Compliant	6	0	1
CWE-563	Not Compliant	6	0	0
CWE-316	Not Compliant	3	0	0
CWE-369	Not Compliant	1	0	0
CWE-546	Not Compliant	1	0	0
CWE-22	Compliant	0	0	0
CWE-77	Compliant	0	0	0

Figure 3-10:
CWE 3.2 - .NET compliace report

UNIT TESTING

Software verification and validation is an inherent part of automotive software development and testing is a key way to demonstrate correct software behavior. Unit testing is the verification of module design. It ensures that each software unit does what it's required to do.

In addition, safety and security requirements may require that software units don't behave in unexpected ways and are not susceptible to manipulation with unexpected data inputs.

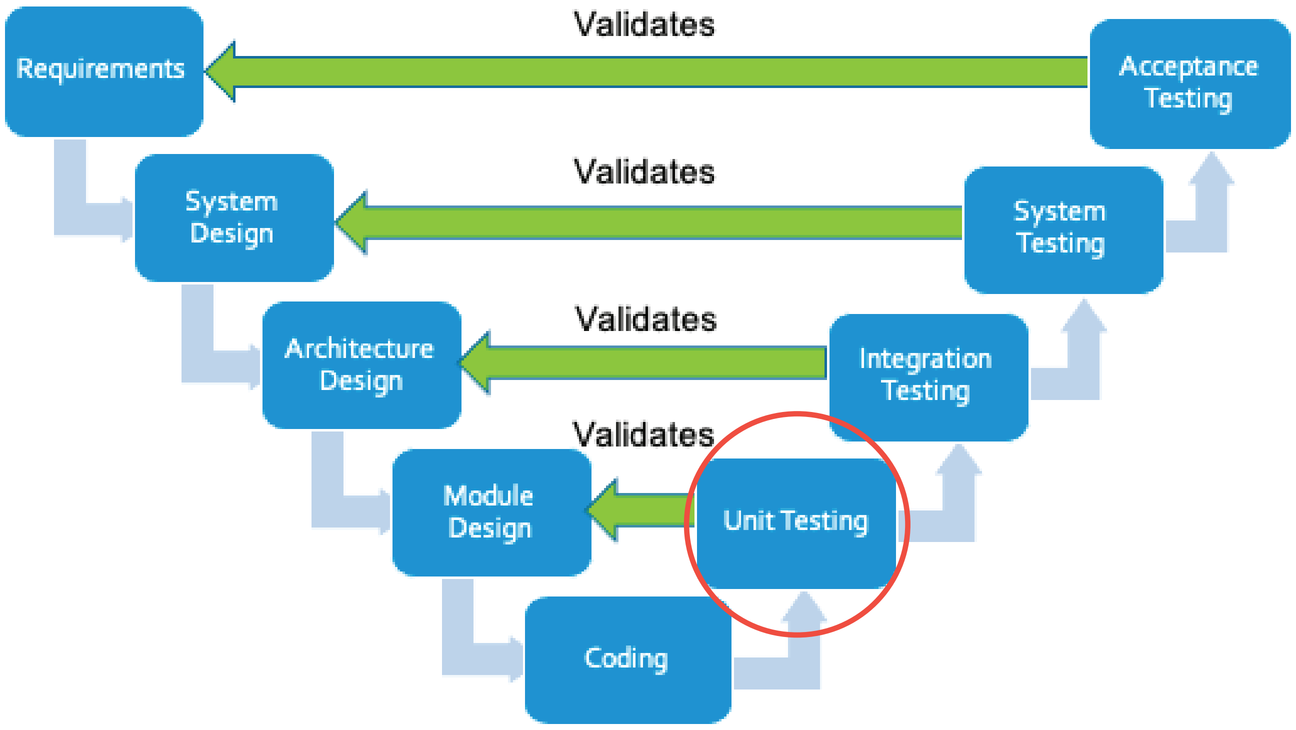


Figure 4-1:
The V-model of software development showing the relationship between each phase and the validation inferred at each stage of testing.

In terms of the classic V model of development, unit test execution is a validation practice to ensure module design is correct. ISO 26262 has specific guidelines for what needs to be tested for unit testing.

ISO 26262 has specific guidelines for testing in accordance with safety integrity level where requirements-based testing and interface testing are highly recommended for all levels. Fault injection and resource usage tests are recommended at lower integrity levels and highly recommended at ASIL (Automotive Safety Integrity Levels) D. Similarly, the method of driving test cases is also specified with recommended practices.

Table 13 - Methods for software unit verification

METHODS		ASIL			
		A	B	C	D
1a	Requirement-based test	++	++	++	++
1b	Interface test	++	++	++	++
1c	Fault injection test	+	+	+	++
1d	Resource usage evaluation	+	+	+	++
1e	Back-to-back comparison test between model and code (if applicable)	+	+	++	++

Figure 4-2: ISO 26262 Part 6, 10.4.3:2011

Table 14 - Methods for deriving test cases for software integration testing

METHODS		ASIL			
		A	B	C	D
1a	Analysis of requirements	++	++	++	++
1b	Generation and analysis of equivalence classes	+	++	++	++
1c	Analysis of boundary values	+	++	++	++
1d	Error guessing	+	+	+	+

Figure 4-3: ISO 26262 Part 6, 10.4.4:2011

Breaking these down individually, consider how each unit test requirement from ISO 26262 can be satisfied and accelerated with test automation tools like Parasoft C/C++test.

UNIT TEST METHODS

Requirement-Based Test

These tests directly test functionality 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.

Interface Test

These 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 Test

These 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, max, and heuristic values.

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 do not 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, such as heuristic values, boundary values, employ data sources to use a wide range of input values in tests.

Error Guessing

This 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 AND TEST CASE GENERATION

Test automation provides large benefits to embedded automotive 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 for Automotive Systems

Automating testing for automotive 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.

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, test results and code coverage data collection from the target system are essential for validation and standards compliance.

Traceability between test cases, test results, source code, and requirements must be recorded and maintained. So 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.

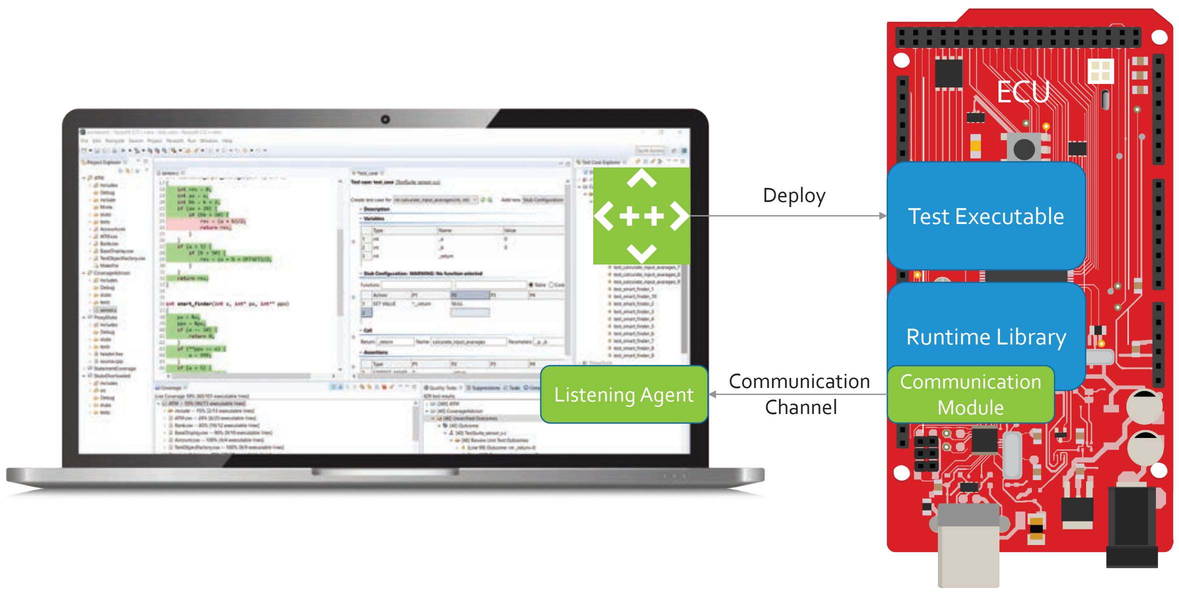


Figure 4-4: A high-level view of deploying, executing, and observing tests from host to target.

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 eclipse, VS Code, Green Hills Multi, Wind River Workbench, IAR EW, ARM MDK, ARM DS-5, TI CCS, Visual Studio, and many others.

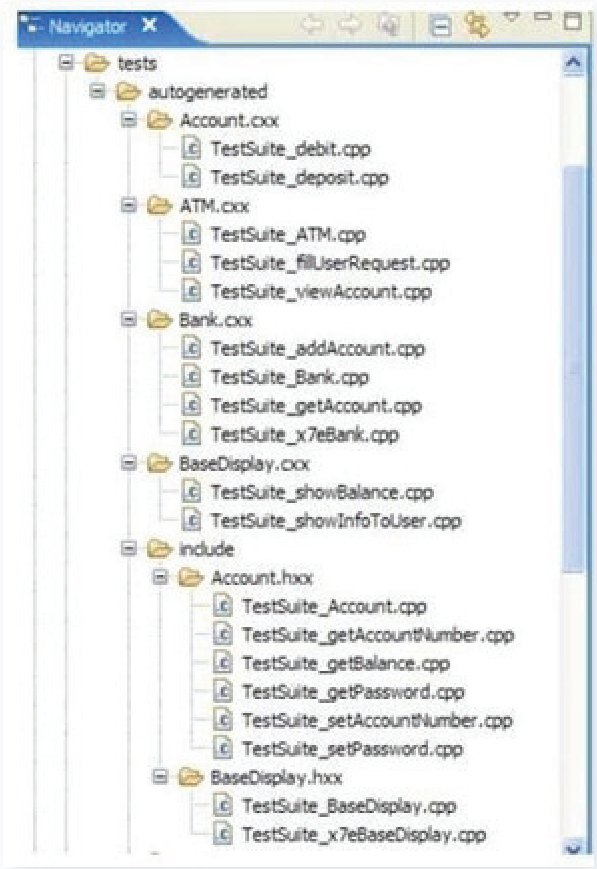
AUTOMATED TEST CASE GENERATION

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.

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 includes 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.



```
tests
  autogenerated
    Account.cxx
      TestSuite_debit.cpp
      TestSuite_deposit.cpp
    ATM.cxx
      TestSuite_ATM.cpp
      TestSuite_fillUserRequest.cpp
      TestSuite_viewAccount.cpp
    Bank.cxx
      TestSuite_addAccount.cpp
      TestSuite_Bank.cpp
      TestSuite_getAccount.cpp
      TestSuite_x7eBank.cpp
    BaseDisplay.cxx
      TestSuite_showBalance.cpp
      TestSuite_showInfoToUser.cpp
    include
      Account.hxx
        TestSuite_Account.cpp
        TestSuite_getAccountNumber.cpp
        TestSuite_getBalance.cpp
        TestSuite_getPassword.cpp
        TestSuite_setAccountNumber.cpp
        TestSuite_setPassword.cpp
      BaseDisplay.hxx
        TestSuite_BaseDisplay.cpp
        TestSuite_x7eBaseDisplay.cpp
```

Figure 4-5: Parasoft C/C++ automated test case generation, in this case, one test suite per function.

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.

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.

REGRESSION TESTING IN SAFETY-CRITICAL SOFTWARE

In safety-critical software development, validation is critical in proving correct functionality, safety, and security. Tests are needed to confirm any changes to the application to ensure functionality and to verify there are no unforeseen impacts on the rest of the system.

If a test case passed in the past and 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 inputs and outputs values.

Regression testing of embedded systems also includes the execution of:

- Integration test cases
- System test cases
- Performance test cases
- Stress test cases and more

In fact, all previously created test cases may 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 or developed upon. If you do not 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 will automatically verify 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++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 & C++
- dotTEST for C#
- Jtest for Java

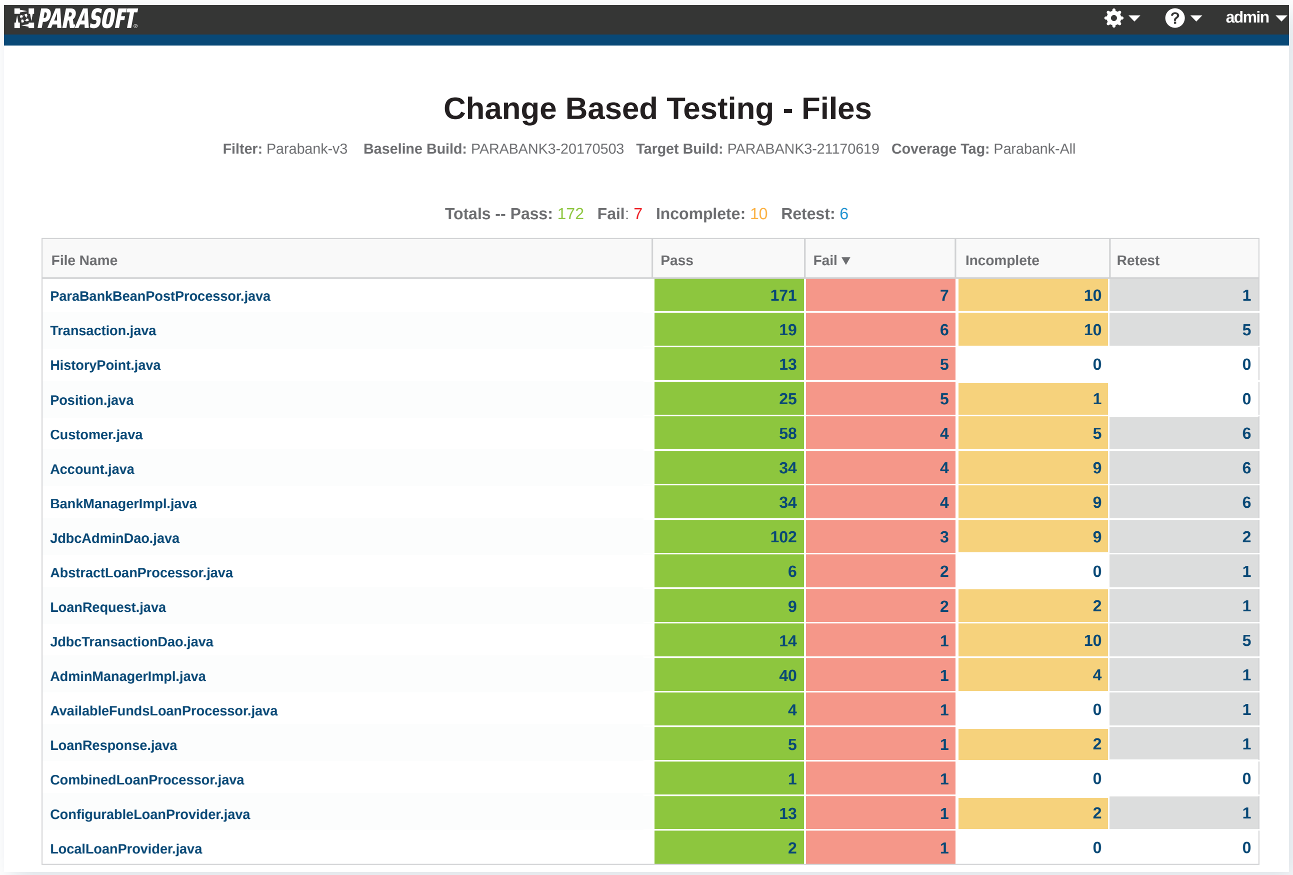
With this combination, teams can improve efficiency and achieve the promise of Agile.

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 and identify the subset of regression tests that need to be executed. It also understands the dependencies on the units modified to determine what ripple effect the changes have made on other units.

Parasoft C/C++test, 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. See the example change based testing report below.



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 which are and are not tested.

SOFTWARE INTEGRATION TESTING

Integration testing follows unit testing with the goal of validating the architectural design. Integration testing can be done bottom up and top down with a combination of approaches likely in many software organizations.

BOTTOM-UP INTEGRATION

This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds. The approach follows a version of the testing pyramid where unit testing forms the foundation of a thorough testing regime. Integration tests follow the integration of units into larger architectural blocks.

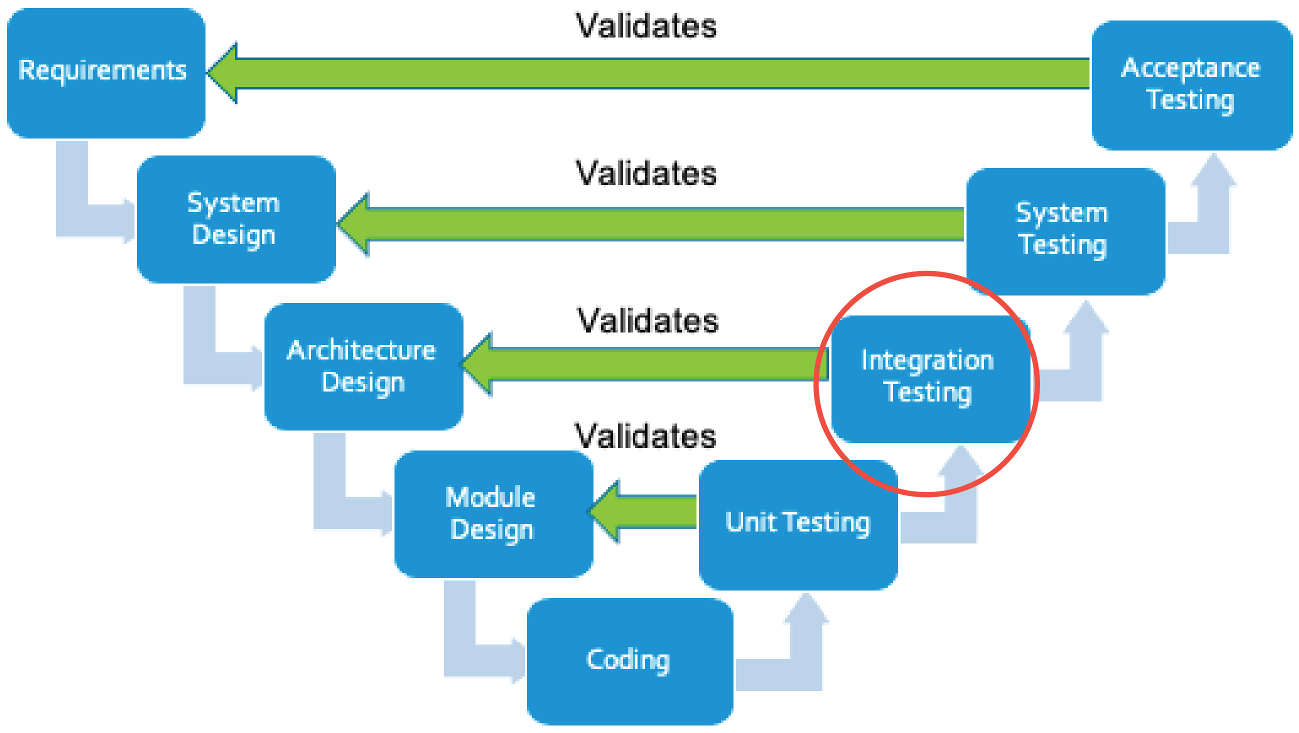
TOP-DOWN INTEGRATION

In this testing, the highest level modules are tested first. Progressively, testing of lower-level modules follows. 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.

The V-model might imply a waterfall development method. However, there are ways to incorporate Agile, DevOps, and CI/CD into this type of product development while still being standards compliant.

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



While the act of performing tests 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

» Analysis

» Traceability

» Test

» Code coverage and more

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

INTEGRATION AND 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, which 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 real-time, objective assessment of the risks associated with a system under development. Applied uniformly, it allows both business and technical managers to make better trade-off decisions between release scope, time, and quality.

Continuous testing isn't just more automation. It's a larger reassessment of software quality practices that's driven by an organization's cost of quality and balanced for speed and agility. Even within the V-model used in safety-critical software development, continuous testing is still a viable approach, particularly during phases of testing, for example, during unit testing and integration testing.

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.

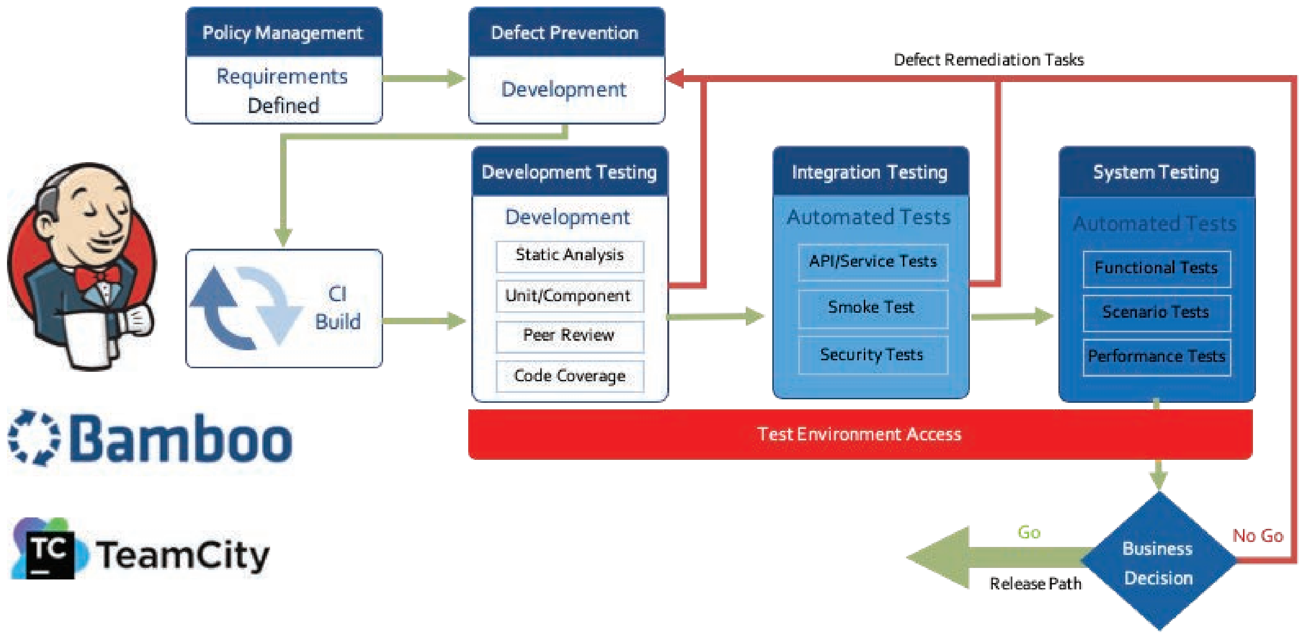


Figure 6-2: A continuous testing cycle

PARASOFT ANALYSIS AND REPORTING IN SUPPORT OF INTEGRATION AND SYSTEM TESTING

Parasoft test automation tools support the validation (actual 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 standards requirements. Key aspects of safety-critical automotive software development are requirements traceability and code coverage.

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 lifecycle 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 such as Intland codebeamer, Polarion from Siemens, Atlassian Jira, Jama Connect, and others. As shown in the image below, each of Parasoft's test automation solutions (C/C++test, Jtest, dotTEST, SOAtest, and Selenic) used within the development life cycle support the association of tests with work items defined in these systems, such as requirements, defects, test case/test runs. Traceability is managed through Parasoft DTP, central reporting and analytics dashboard.

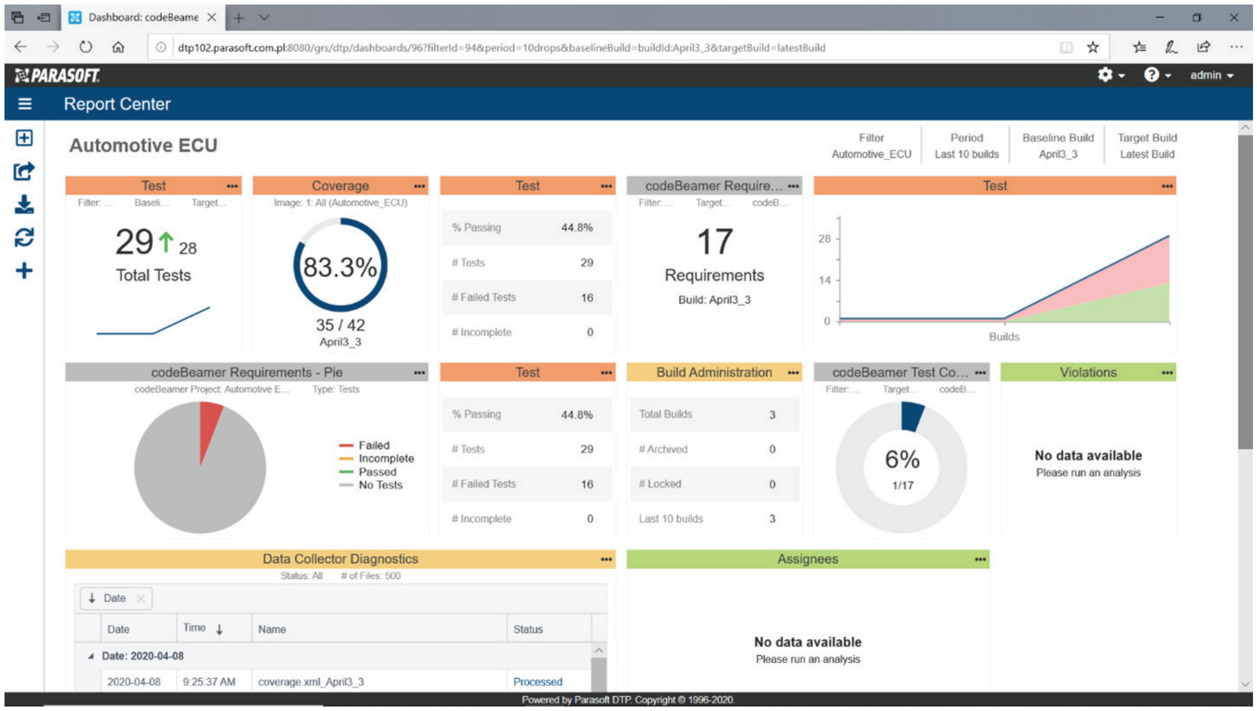


Figure 6-3: Parasoft provides a reporting dashboard that capture the project's testing status, correlation to requirements and progress towards completion.

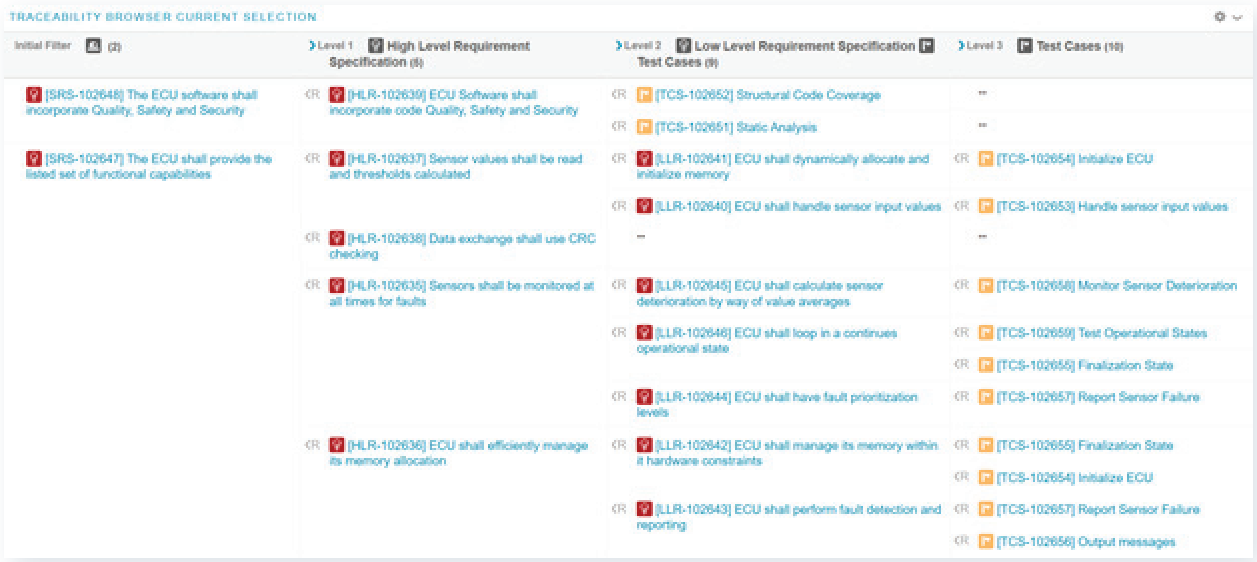
Parasoft DTP correlates the unique identifiers from the management system with:

» 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.

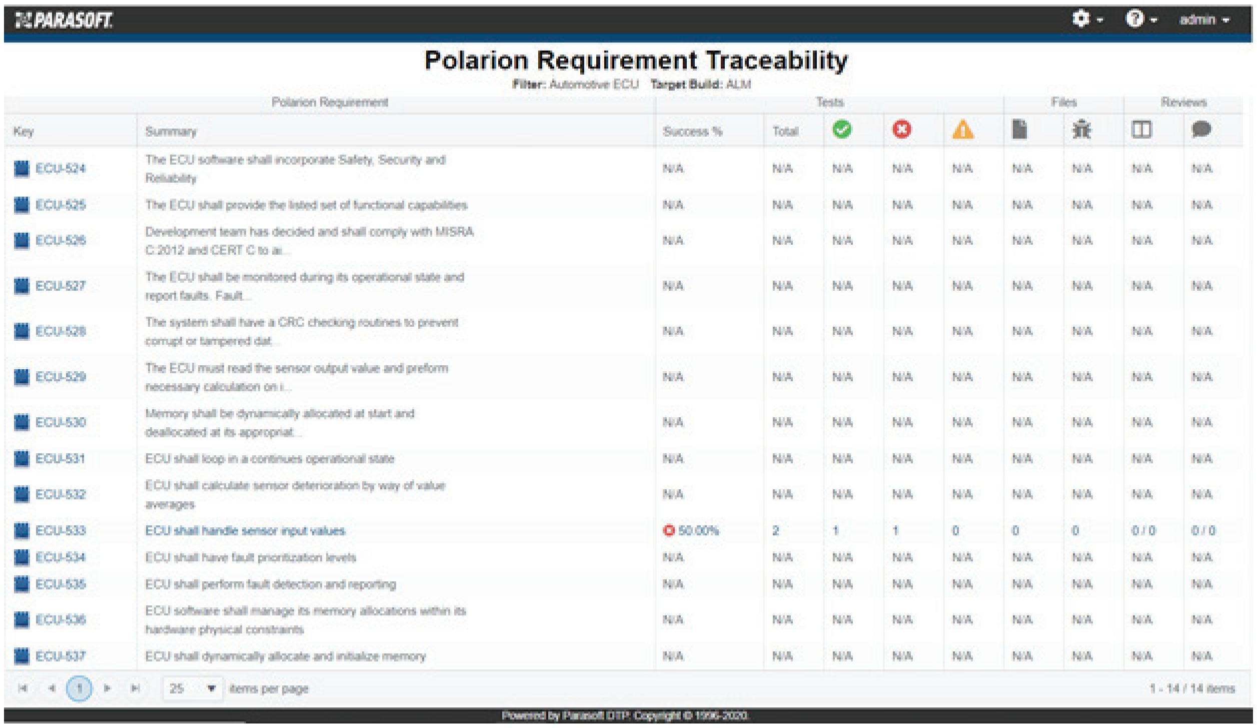


TRACEABILITY BROWSER CURRENT SELECTION

Initial Filter (2)	Level 1 High Level Requirement Specification (6)	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 6-4: codebeamer traceability matrix. System requirements to high level requirements to low level requirement to test cases and test results.

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



PARASOFT admin

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. Depending on the safety integrity level (ASIL in ISO 26262), the depth and completeness of the code coverage will vary.

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 part of 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 is 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.

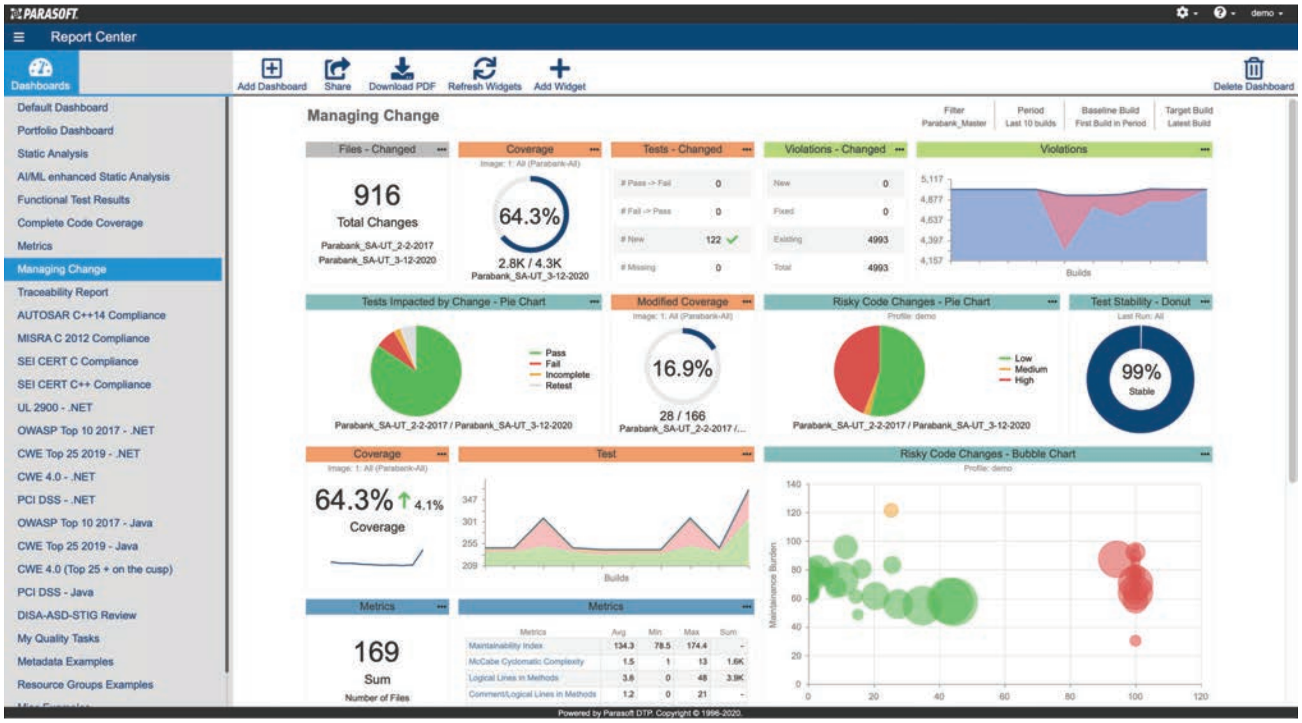


Figure 6-6: Parasoft DTP reporting and analytics dashboard

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 C/C++test, 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 AND SYSTEM TESTING WITH PARASOFT 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 it meets the specified functional, safety, security, and other nonfunctional requirements.

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 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 validated and system wide security testing is run.

AUTOMOTIVE SYSTEM TESTING AT THE SERVICE LEVEL

Individual systems within an automobile may not be considered part of a service. However, connectivity into larger systems means they should be. For example, in an automobile, the role of the engine control unit (ECU) alone is to ensure proper combustion and emissions in the engine, but the car is tracking fuel economy, using the ECU, and reporting it to a central server over a wireless connection.

This mileage data is then used to plan routes and estimate operating costs. Suddenly, the ECU is a critical leaf node in a business decision making process.

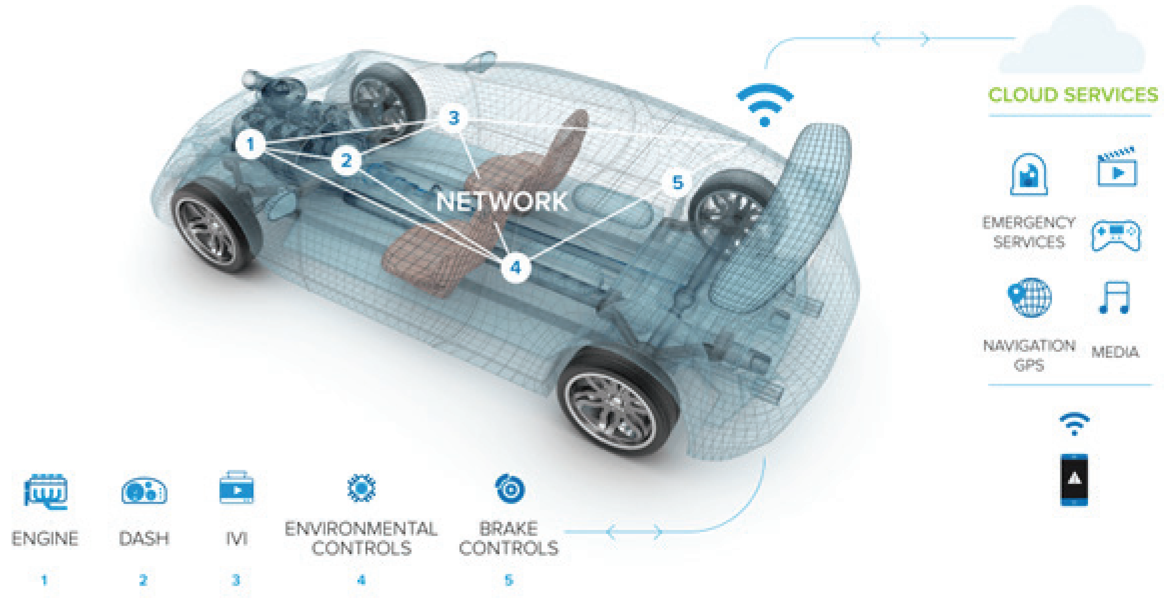


Figure 7-1: Engine Control Unit (ECU) and user services all connected to the cloud

Instead of viewing system quality in terms of meeting individual device 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. In the HVAC example, the new temperature sensor can be tested with varying request rates to see if it meets performance requirements.

Security is a significant concern in automotive systems. Cyber attacks most 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. A new temperature sensor in the HVAC example might operate correctly with expected requests, but crash when overloaded. An attacker might be able to exploit this to overload the system and cause an outage.

VIRTUAL TEST ENVIRONMENT AND SERVICE LEVEL TESTING

A real test lab requires the closest physical manifestation of the environment in which an automobile 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 non-existent) hardware dependencies. They use sophisticated service virtualization with other key test automation tools.

Service Virtualization

Simulates all 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 that an automobile communicates. Similarly, virtualization can control simulate a dependent system, like traffic or weather data, in a realistic manner.

Service and API Testing

Provide a way to drive the system under test in a manner that ensures the services it provides (and APIs provided) are performing flawlessly. These tests can be manipulated via the automation platform to perform performance and security tests as needed.

Runtime Monitoring

Detects errors in real time on the system under test and captures important trace information.

Test Lab Management and Analytics

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

PARASOFT SOATEST AND VIRTUALIZE FOR SERVICE LEVEL TESTING OF AUTOMOTIVE 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 automotive software development. Code coverage measures the completion of test cases and executed tests. It provides evidence that validation is complete, at least as specified by the software design.

It also identifies dead code. This is code that can logically never be reached. It demonstrates the absence of unintended behavior. Code that isn't covered by any test is a liability because its behavior and functionality are unknown.

The amount and extent of code coverage depends on the safety integrity level. The higher the integrity level, the higher the rigor used, and inevitably the number and complexity of test cases.

The following table shows the recommendations for types of code coverage at each ISO 26262 ASIL.

Table 12 - Structural coverage metrics at the software level

METHODS		ASIL			
		A	B	C	D
1a	Statement coverage	++	++	+	+
1b	Branch coverage	+	++	++	++
1c	MC/DC (Modified Condition/Decision Coverage	+	+	+	++

Figure 8-1: ISO 26262 Part 6, 9.4.5:2011

Statement coverage requires that each program statement be executed at least once and is recommended at the lower ASIL levels. 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. See this tutorial from NASA.

Advanced unit test automation tools such as Parasoft C/++test provide all these code coverage metrics. C/C++test 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.

INCREASING CODE COVERAGE WITH AUTOMATED UNIT TEST CASE CREATION

The creation of productive unit tests has always been a challenge. Functional safety standards compliance demands high-quality software, which drives a need for test suites that affect and produce high code coverage statistics. Teams require unit test cases that help them achieve 100% code coverage.

This is easier said than done. Analyzing branches in the code and trying to find reasons why certain code sections are not covered continues to steal cycles from development teams.

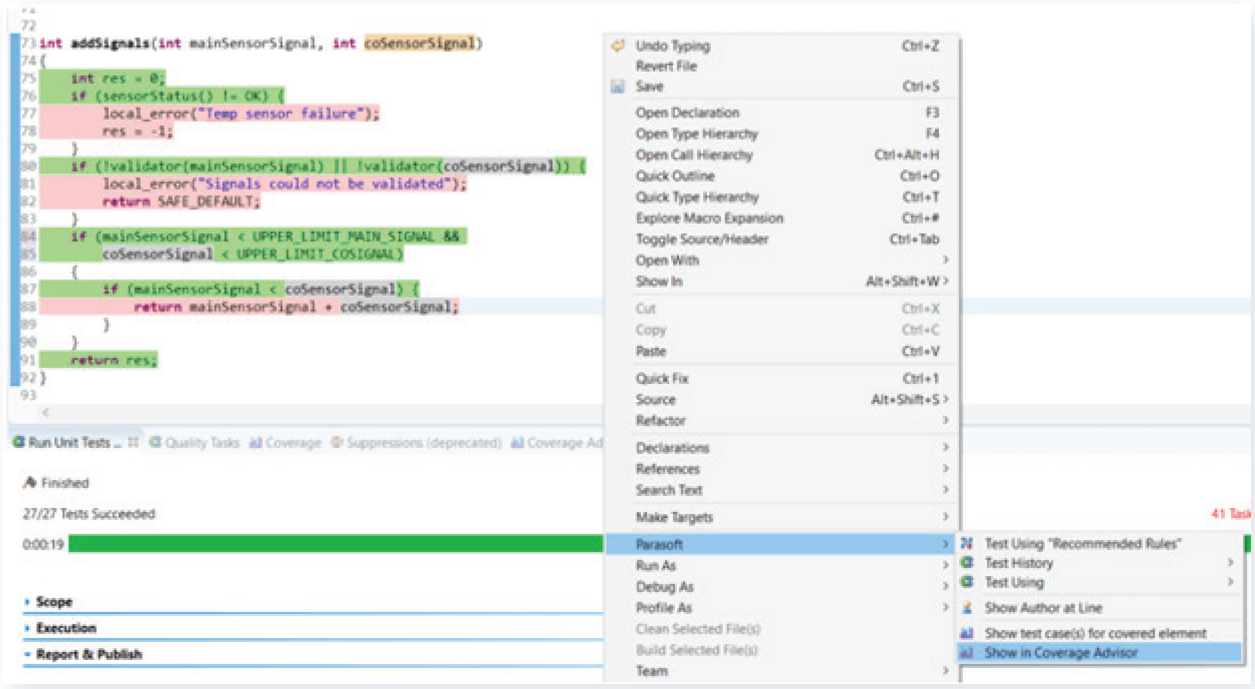
Parasoft Coverage Advisor

Parasoft C/C++test resolves the 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 of 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.

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

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

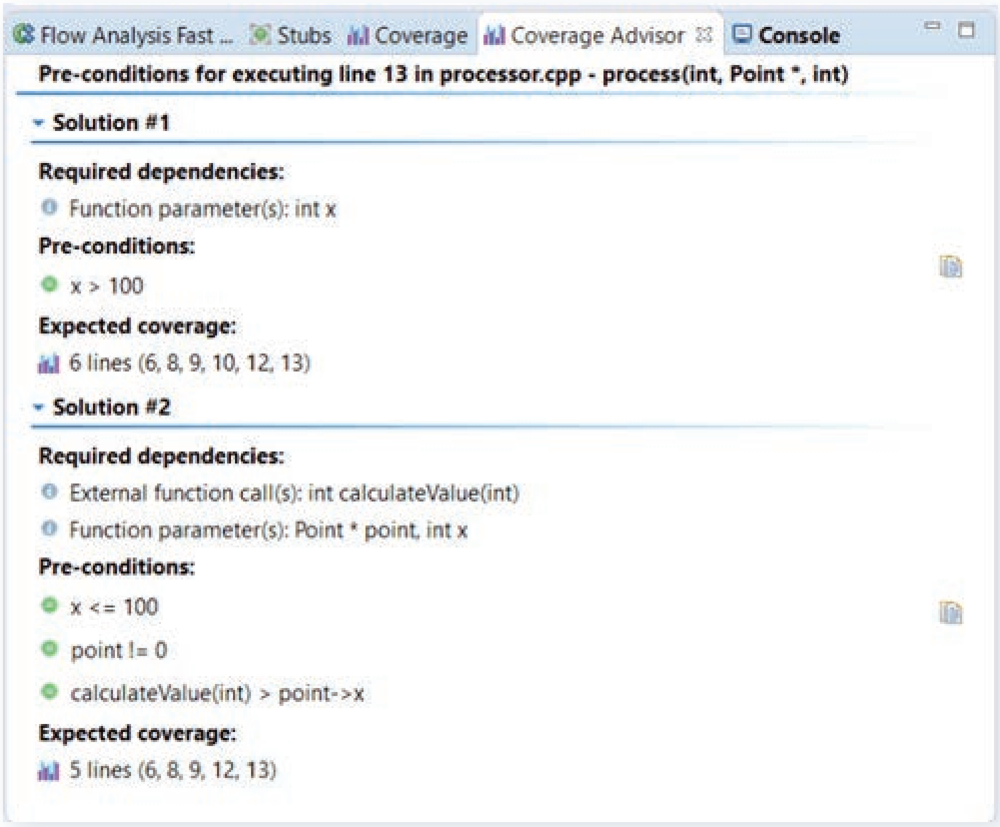


```
int addSignals(int mainSensorSignal, int coSensorSignal)
{
    int res = 0;
    if (sensorStatus() != OK) {
        local_error("Temp sensor failure");
        res = -1;
    }
    if (!validator(mainSensorSignal) || !validator(coSensorSignal)) {
        local_error("Signals could not be validated");
        return SAFE_DEFAULT;
    }
    if (mainSensorSignal < UPPER_LIMIT_MAIN_SIGNAL &&
        coSensorSignal < UPPER_LIMIT_COSIGNAL)
    {
        if (mainSensorSignal < coSensorSignal) {
            return mainSensorSignal + coSensorSignal;
        }
    }
    return res;
}
```

The figure below shows an analysis report providing the user with a solution. The Pre-conditions 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, the user will obtain 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.

REQUIREMENTS AND THE TRACEABILITY MATRIX

In ISO 26262, requirements management is a mandatory part of the software development process and the traceability of those requirements to implementation—and subsequently, proof of correct implementation need to be ensured.

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)."[1]

In the simplest sense, requirements traceability is needed to keep track of 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 is needed.

Traceability works both to prove you satisfied the requirements and to identify what doesn't. 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 go beyond providing proof of the implementation. Disciplined traceability is an important visibility into development progress.

It's important to realize that many requirements in safety-critical software are derived from safety analysis and risk management. The system must perform it's 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 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 diagram of software.

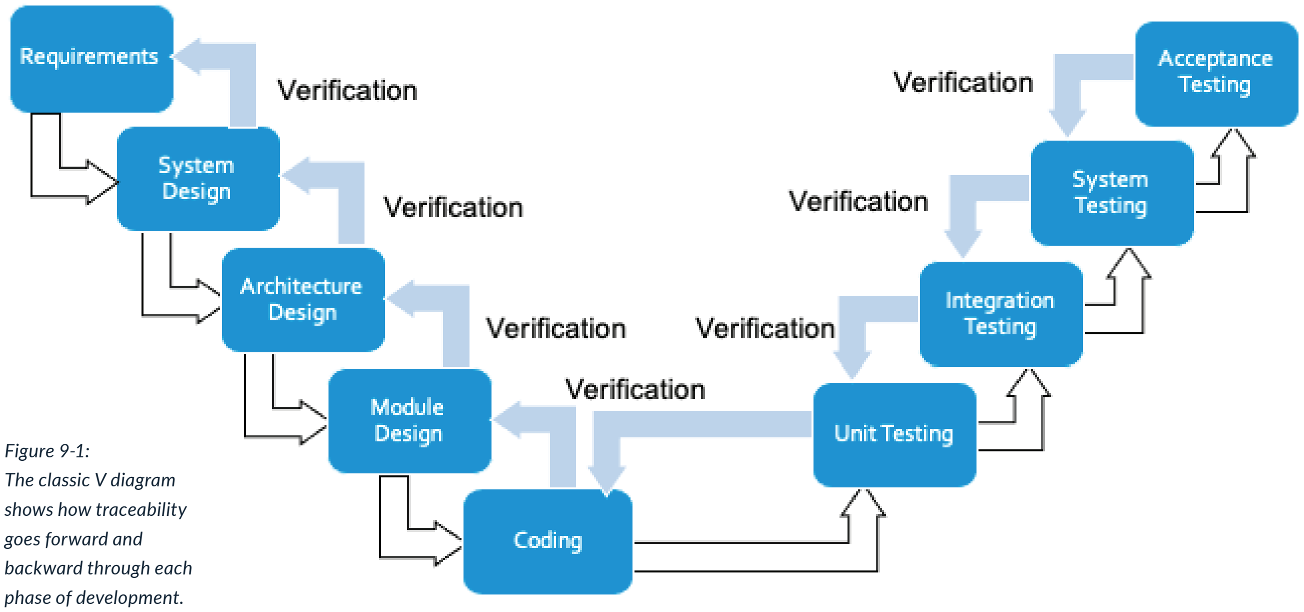


Figure 9-1: The classic V diagram shows how traceability goes forward and backward through each phase of development.

[1]Gotel O.C.Z and Finklestein A.C.W., "An analysis of the requirements traceability problem", in Proceedings of ICRE94, 1st International Conference on Requirements Engineering, Colorado Springs, Co, IEEE CS Press, 1994

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.

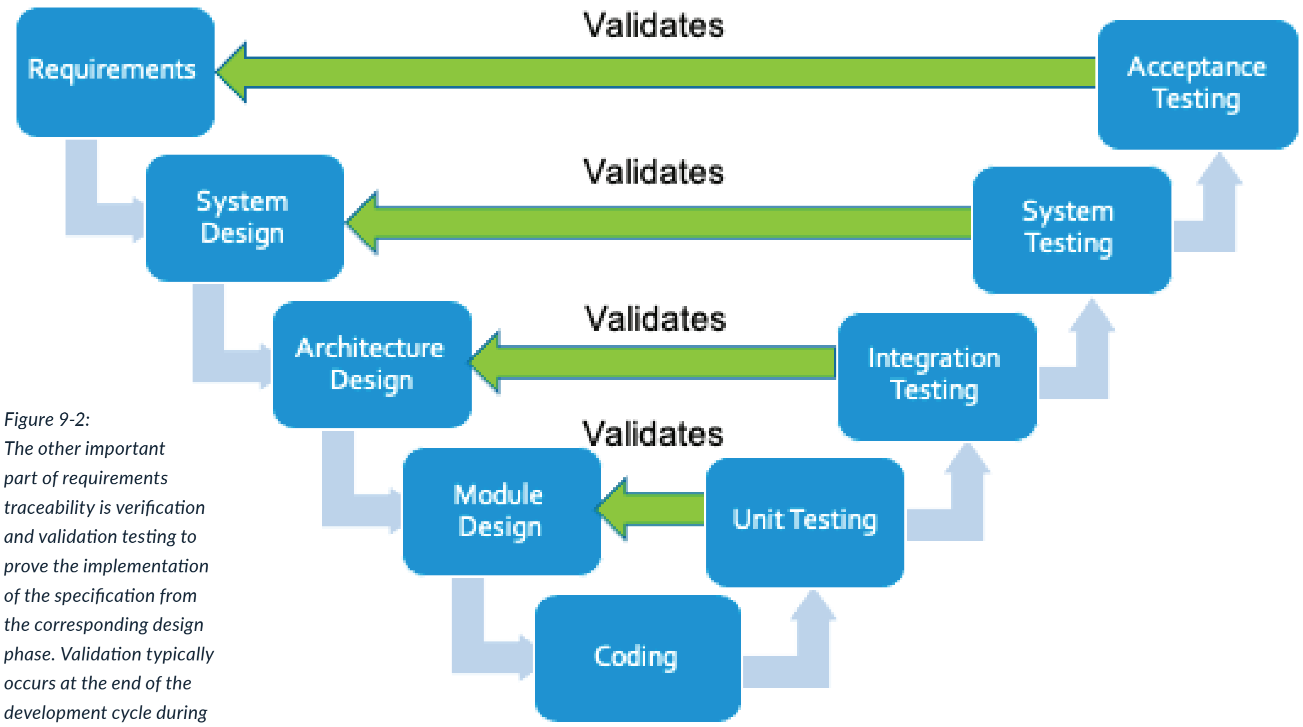


Figure 9-2:
The other important part of requirements traceability is verification and validation testing to prove the implementation of the specification from the corresponding design phase. Validation typically occurs at the end of the development cycle during final acceptance testing with the customer.

In the V diagram shown above, each testing phase verifies the satisfaction of the specifications associated with the corresponding design/implementation phase. In the example, you see:

» Acceptance testing validates requirements.

» Integration testing verifies architecture design.

» Unit testing verifies module design, and so on.

Validation typically occurs at the end of the development lifecycle during acceptance testing with the customer.

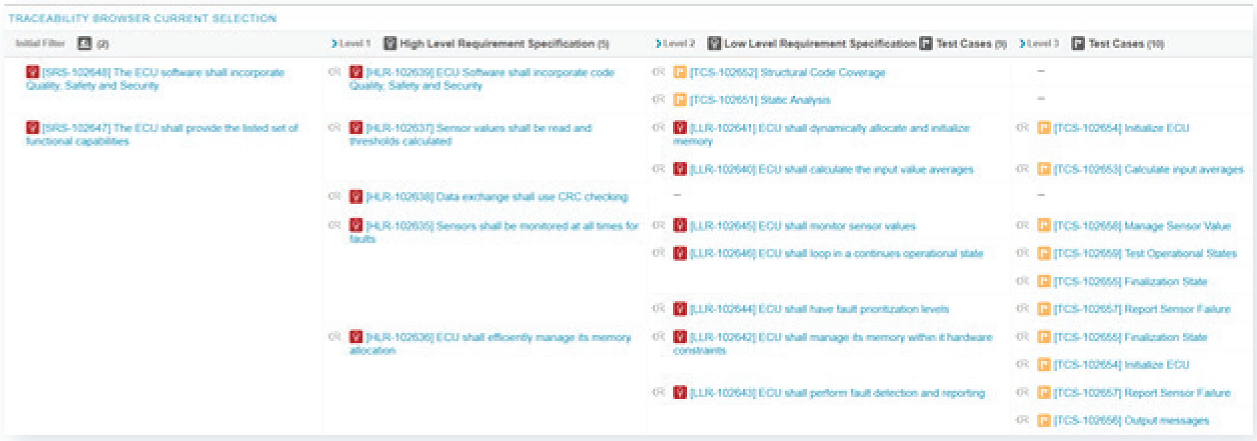
Requirements traceability needs both the link to implementation and verification, plus all the associated artifacts from the development process. Software development on any realistic scale will have many requirements, complex design and architecture, and possibly thousands of units and unit tests. Automation 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 a document that illustrates the satisfaction of requirements with a corresponding work item, like a unit test, module source code, architecture design element, 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 calculate the input value averages	[TCS-102653] Calculate input averages
	[HLR-102638] Data exchange shall use CRC checking	–	–
	[HLR-102635] Sensors shall be monitored at all times for faults	[LLR-102645] ECU shall monitor sensor values	[TCS-102658] Manage Sensor Value
		[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 lifecycle 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 such as Intland codebeamer, Polarion from Siemens, Jama Connect, Atlassian Jira, CollabNet VersionOne, and TeamForge.

As shown in the image below, each of Parasoft's test automation tools, C/C++test, 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.

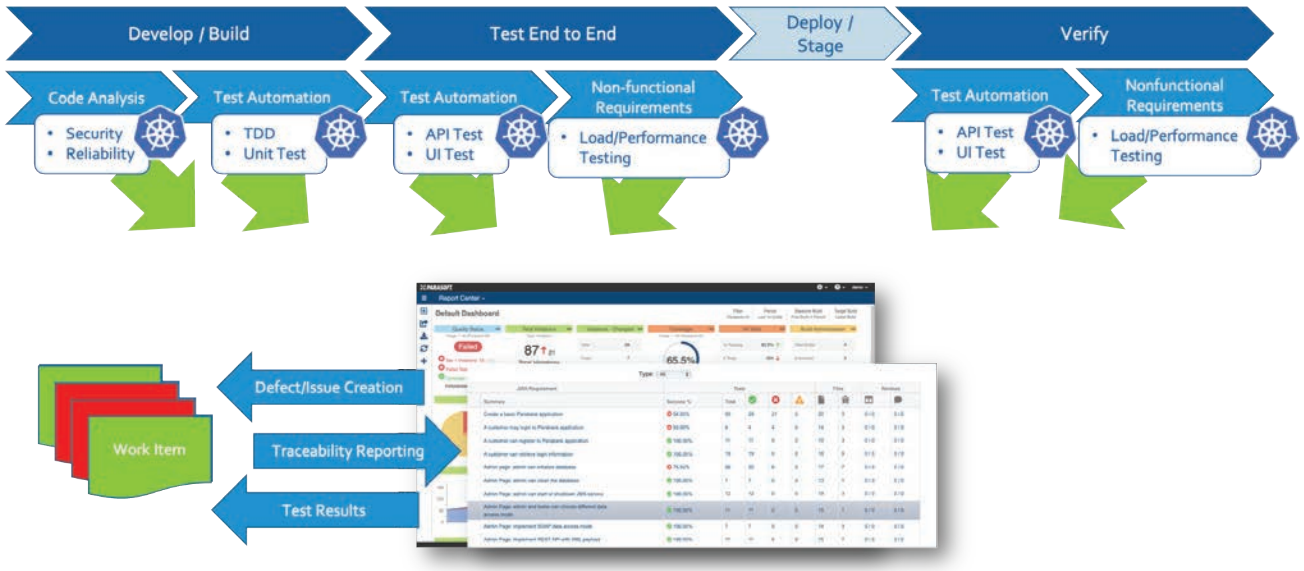


Figure 9-4: Parasoft provides bidirectional traceability from work items to test cases and test results, displaying traceability reports with Parasoft DTP and reporting results back to the requirements management system.

Parasoft DTP correlates the unique identifiers from the management system with static analysis findings, code coverage, and 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 test cases, static analysis findings, the source code files, and the manual code reviews.

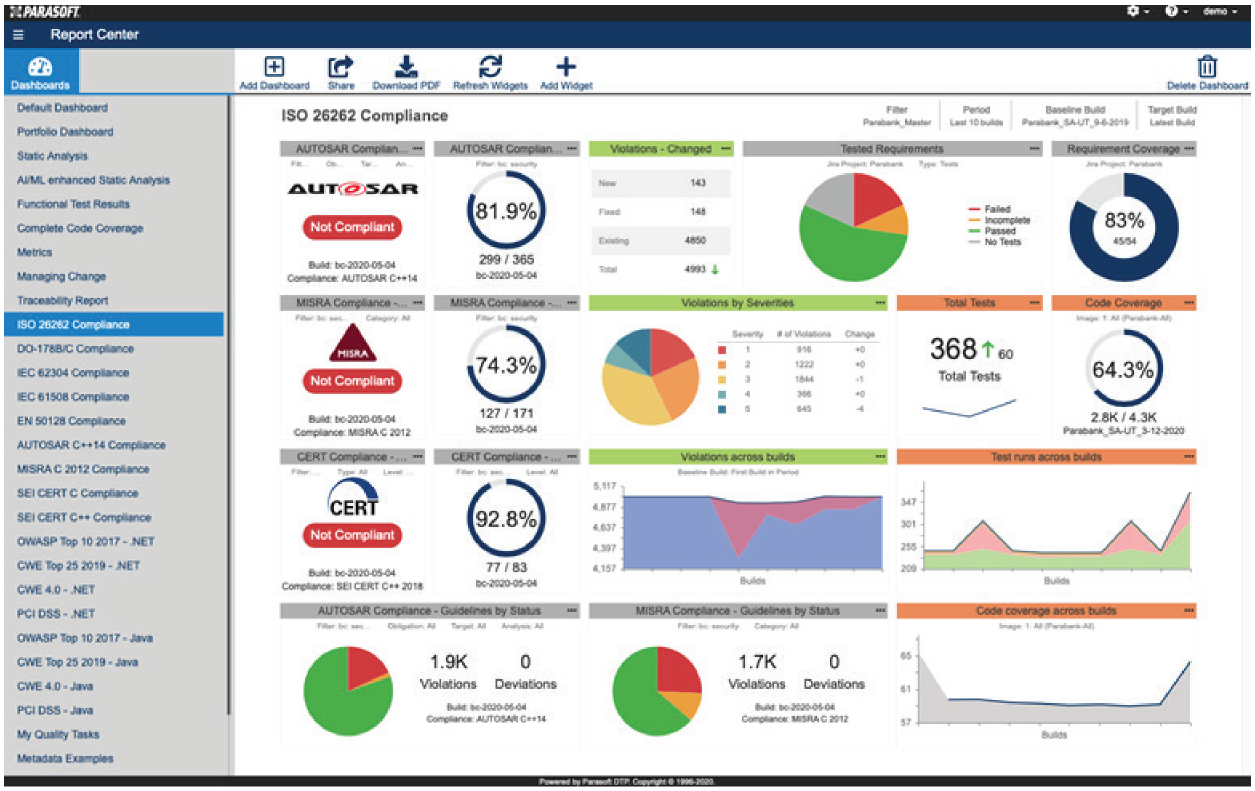


Figure 9-5: Parasoft DTP integrated with Jira for requirements traceability

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 lifecycle 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 into test automation results and complete the software test verification and validation of requirements.

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

TOOL QUALIFICATION FOR SAFETY-CRITICAL AUTOMOTIVE SYSTEMS

Safety-critical software development standards like ISO 26262 require that manufacturers prove that the tools they're using to develop software provide correct and 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 that many organizations fail to plan for.

The end deliverable is proof in the form of documentation, but there is more to the qualification process than just delivering a big pile of static documentation. Parasoft's Qualification Kits for C/C++test, which include a convenient tool wizard that brings automation into the picture and reduces the time and effort required for tool qualification.

PRE-CERTIFIED TOOLS

Tool qualification needs to start with tool selection, ensuring you are using a development tool that is certified by an organization, such as TÜV SÜD. This will significantly reduce the effort when it comes to tool qualification.

Parasoft C/C++test is certified by TÜV SÜD for functional safety according to IEC 61508 and ISO 26262 standards for both host based and embedded target applications. The fully integrated testing solution for C/C++ software development paves the way for a streamlined qualification of static analysis, unit testing, and coverage requirements for the safety-critical standards.

Pre-certified tools are often enough for lower safety integrity levels such as ASIL A and B. However, for ASIL C and D, tool qualification requires further validation, usually requiring verification and validation of the tool itself on target system hardware.

TOOL QUALIFICATION REQUIRES MORE 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.

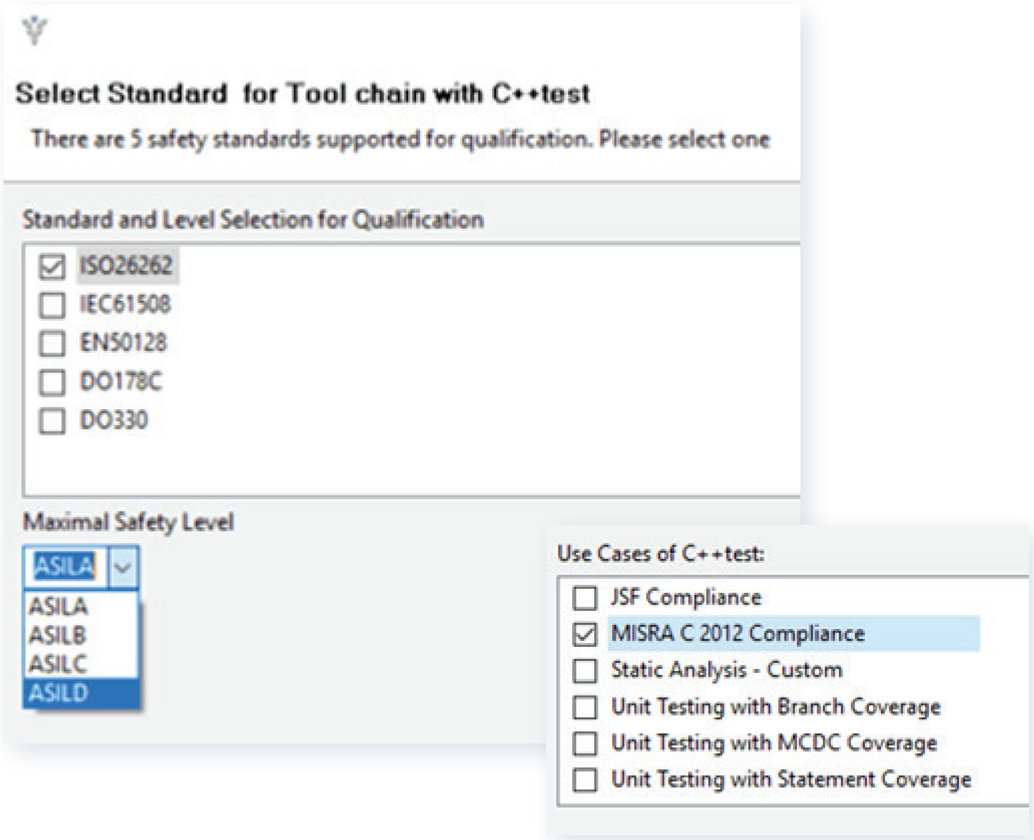
Benefits of Using the Qualification Kits

- 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.

The example below shows the use case of C/C++ static code analysis being used to check compliance to the MISRA C 2012 standard, as part of ISO 26262 qualification. The tool then selects only the parts of the qualification suite needed for this function.



Select Standard for Tool chain with C++test

There are 5 safety standards supported for qualification. Please select one

Standard and Level Selection for Qualification

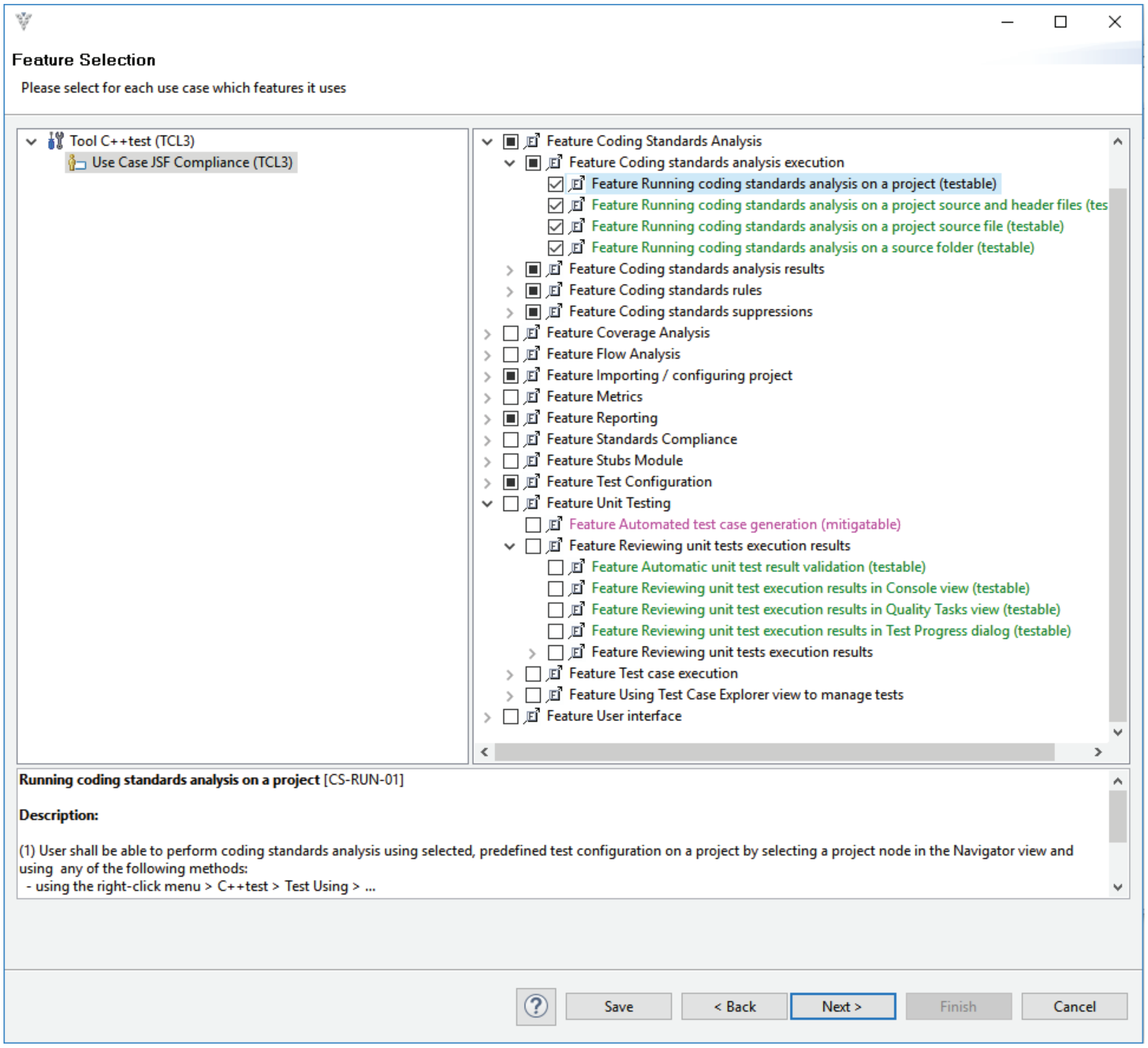
- [✓] ISO26262
- [ ] IEC61508
- [ ] EN50128
- [ ] DO178C
- [ ] DO330

Maximal Safety Level: ASILA (options: ASILA, ASILB, ASILC, ASILD)

Use Cases of C++test:

- [ ] JSF Compliance
- [✓] MISRA C 2012 Compliance
- [ ] Static Analysis - Custom
- [ ] Unit Testing with Branch Coverage
- [ ] Unit Testing with MCDC Coverage
- [ ] Unit Testing with Statement Coverage

Figure 10-1: Functional compliance selection with additional use case settings



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:

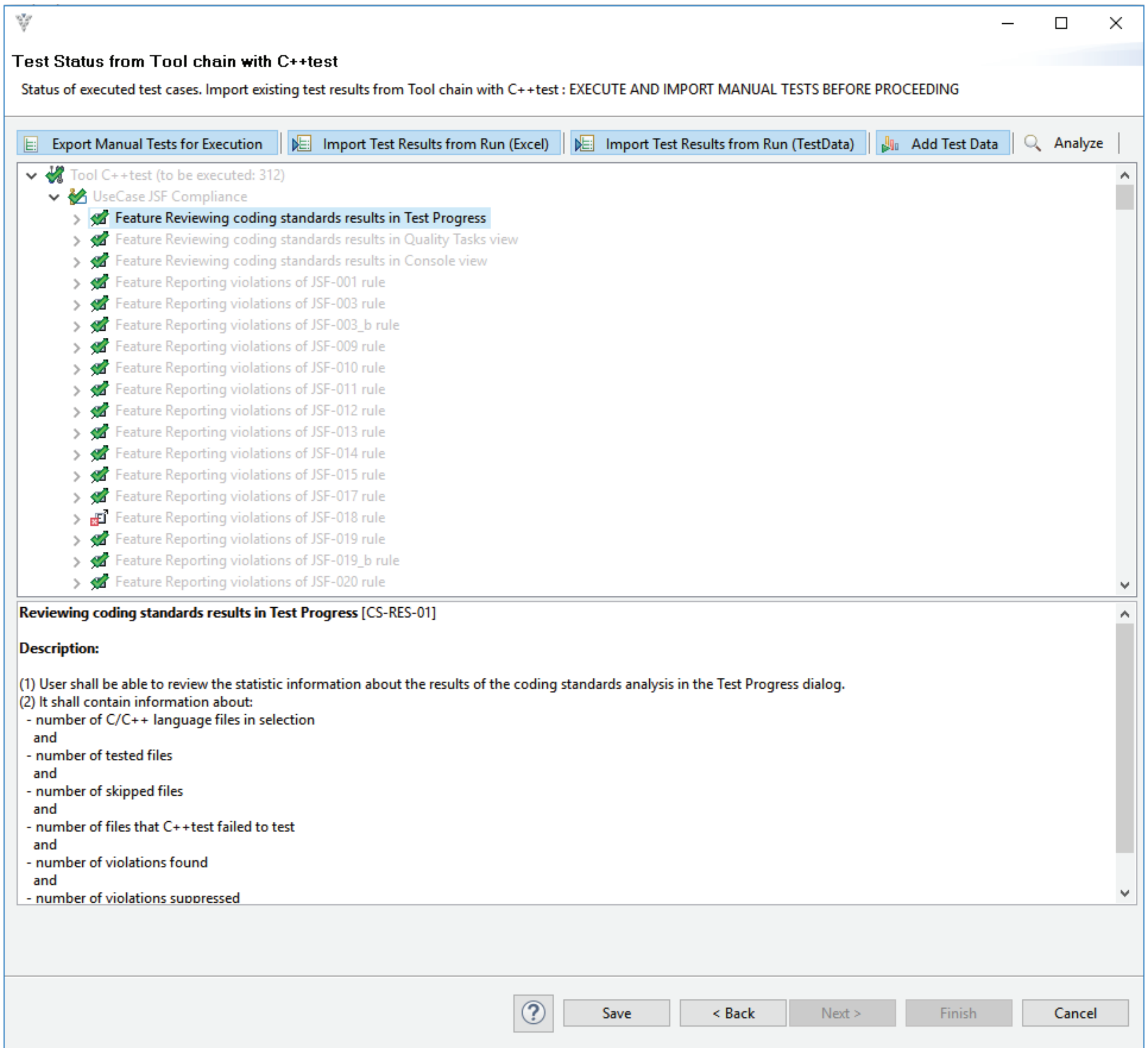
(1) User shall be able to perform coding standards analysis using selected, predefined test configuration on a project by selecting a project node in the Navigator view and using any of the following methods:
 - using the right-click menu > C++test > Test Using > ...

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

LEVERAGE TEST AUTOMATION AND 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, and 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

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

Export Manual Tests for Execution
Import Test Results from Run (Excel)
Import Test Results from Run (TestData)
Add Test Data
Analyze

- Tool C++test (to be executed: 312)
  - 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-003_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:

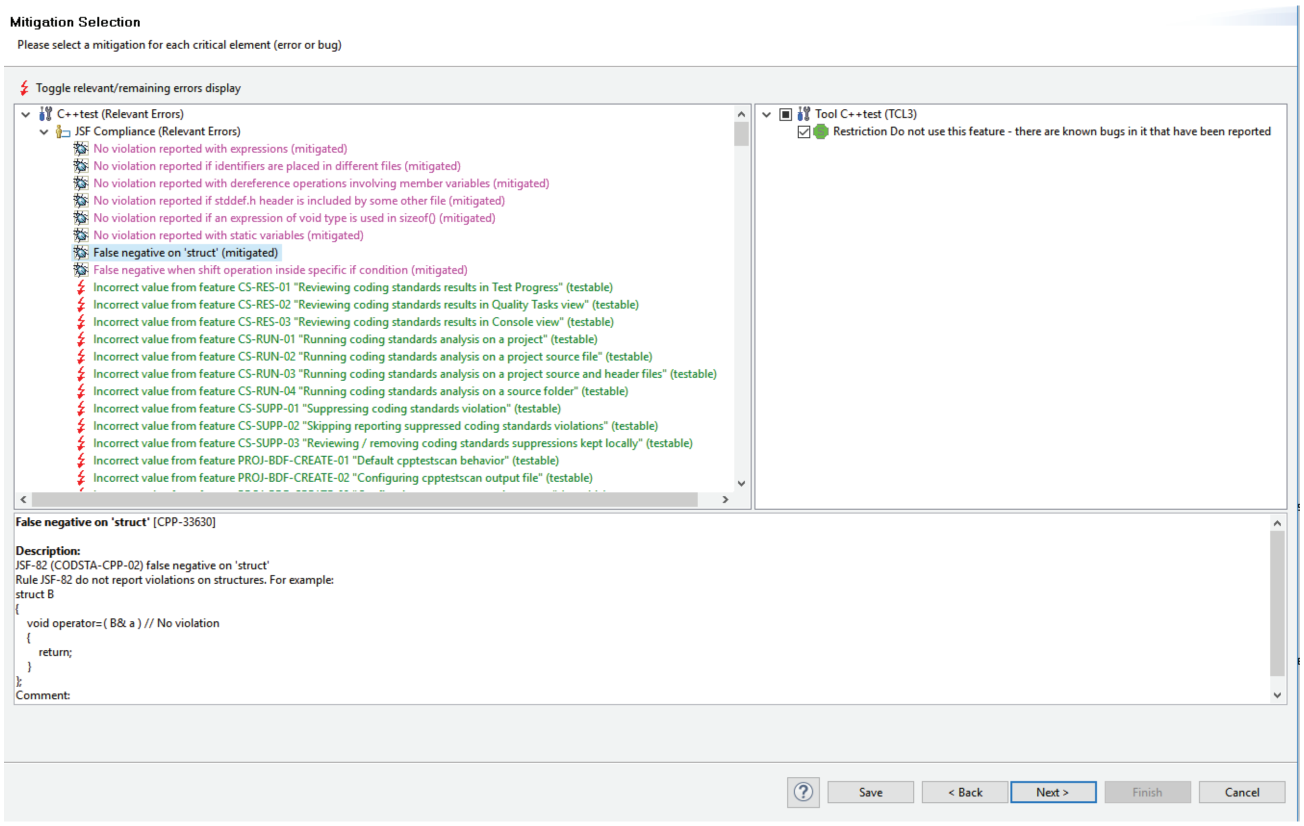
(1) User shall be able to review the statistic information about the results of the coding standards analysis in the Test Progress dialog.
(2) It shall contain information about:
- number of C/C++ language files in selection
 and
- number of tested files
 and
- number of skipped files
 and
- number of 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 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 based on the use cases selected by the user.

» **Tool Qualification Plan** describes how C/C++test is going to be qualified for use in a safety relevant development project.

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

» **Tool Safety Manual** describes how C/C++test should be used safely, for example compliant to safety standards, like ISO 26262 and IEC 61508, in safety-critical projects.

In each of these documents, only the documentation required for the tool features in use is generated because the scope of the qualification was narrowed down at the beginning of the project. Automation and narrowing the scope of qualification greatly reduces the documentation burden.

REPORTING AND ANALYTICS FOR AUTOMOTIVE SOFTWARE

Parasoft's extensive reporting capabilities bring the results of Parasoft C/C++test 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.

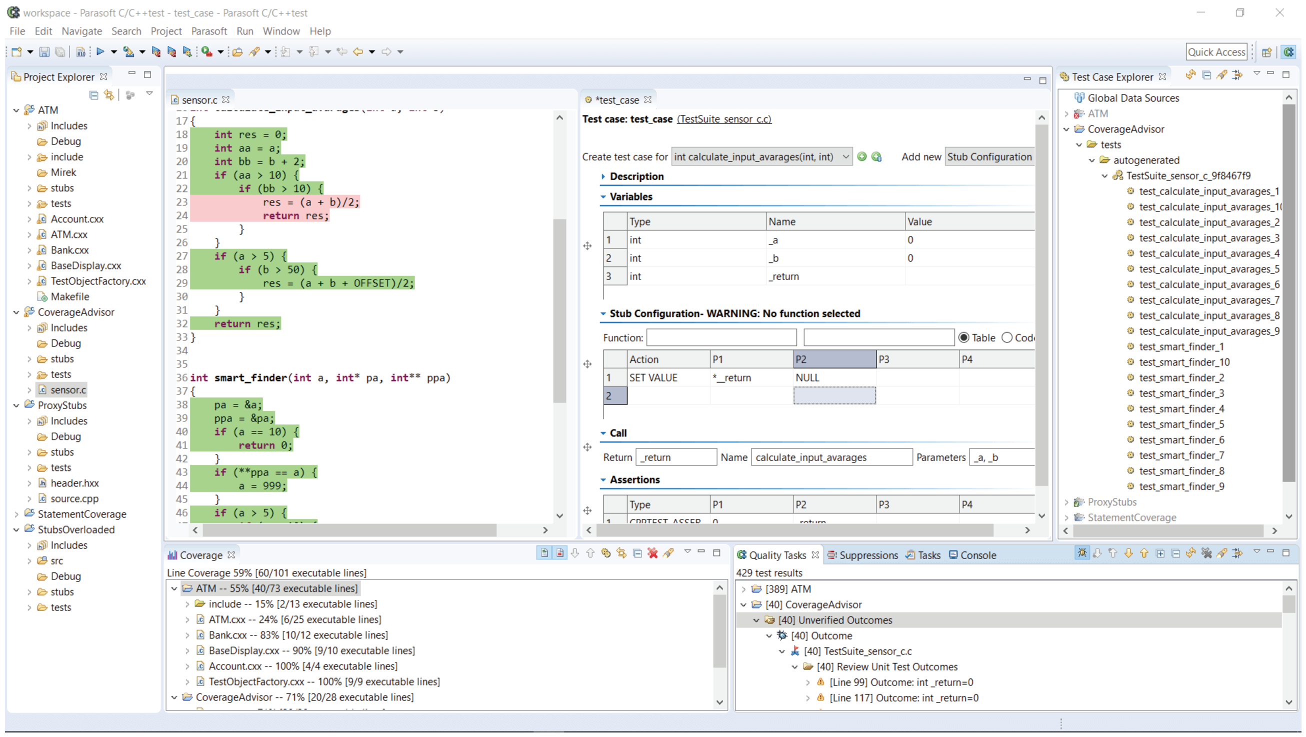
DEVELOPER'S VIEW IN THE IDE

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

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

The Quality Tasks view in the IDE makes it easy for developers to sort and filter the results, for example group per file, per rule, or per 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)

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 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:

» Static analysis findings

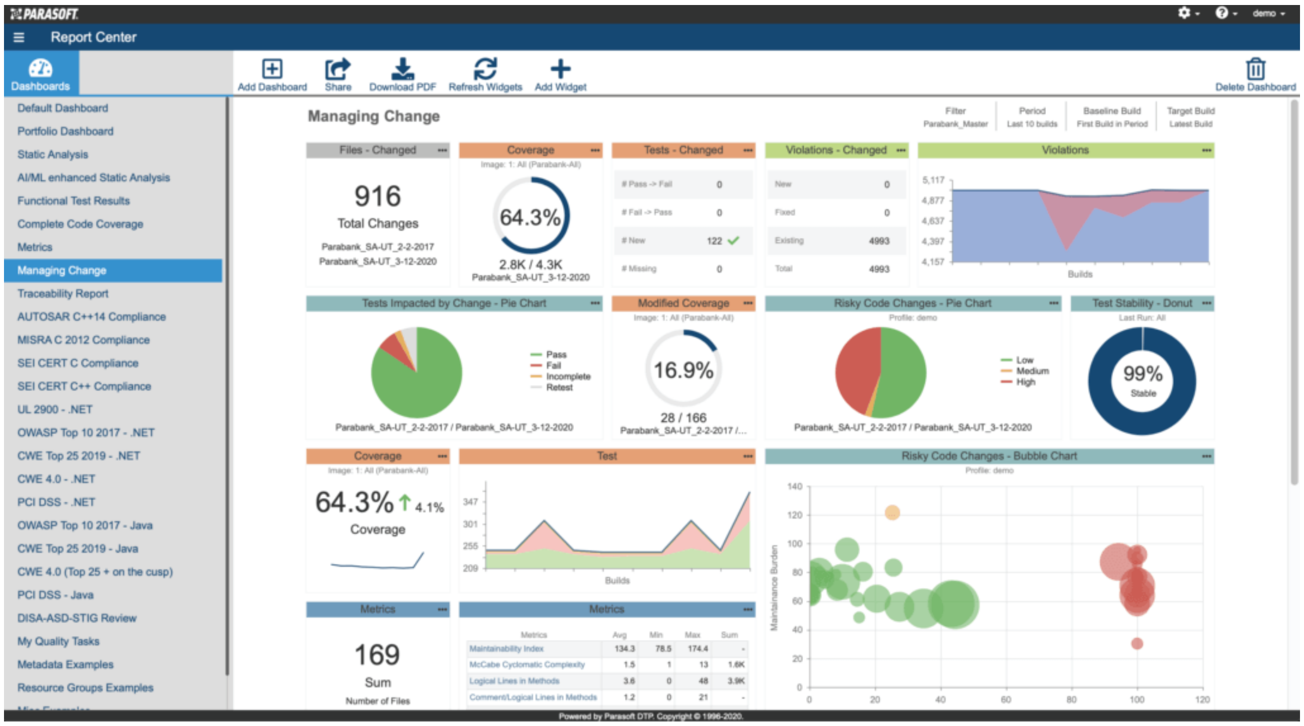
» Metric analysis details

» Unit testing details

» Code coverage details

» Source code details

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, and advanced analytics from Parasoft's 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 for 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 provides specific reporting capabilities suited to functional safety development. Here are two report examples:

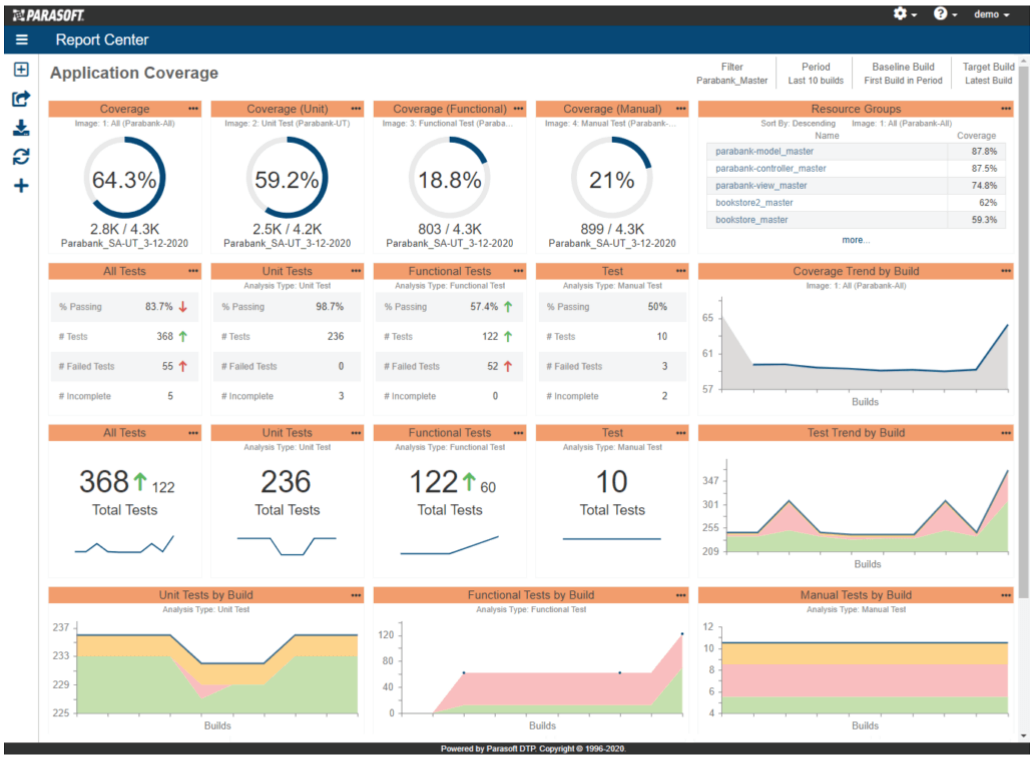
- Unit Testing Execution Details Tests to Requirements Traceability
- Test to Code Coverage Traceability

The ISO26262 Compliance Pack provides a dedicated, standard-driven report template to help teams comply with industry standards and provide automatically generated reports required for code audits.

CODE COVERAGE METRICS

There are various coverage metrics to consider. Knowing which specific type to apply depends on the software integrity level (ASIL) as defined in ISO 26262.

For automotive systems, the control metrics referenced are statements, branch, modified condition decision coverage (MC/DC). For the strictest requirements, there's object/assembly code. Parasoft supports gathering all these coverage metrics, including terms other industries use like block, call, function, path, decision, and more.



Filter	Period	Baseline Build	Target Build
Parabank_Master	Last 10 builds	First Build in Period	Latest Build

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

Resource Groups
Sort By: Descending Image: 1: All (Parabank-All)

Name	Coverage
parabank-model_master	87.8%
parabank-controller_master	87.5%
parabank-view_master	74.8%
bookstore2_master	62%
bookstore_master	59.3%

more...

	All Tests	Unit Tests	Functional Tests	Test
		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

All Tests	Unit Tests	Functional Tests	Test
	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

Powered by Parasoft DTP. Copyright © 1996-2020.

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

CUSTOM ANALYTICS, REPORTS AND DASHBOARDS

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

Benefits From Centralized, Aggregated Data Analysis and Reporting

Manage Compliance With Efficiency, Visibility, and 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:

» 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 and Risk Between New and 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 but by 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

Learn how your embedded software development team can accelerate the delivery of high-quality and compliant software. Contact one of our experts today to request a demo.

ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's 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 delivers 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 AUTOMOTIVE SOFTWARE DEVELOPMENT ASSETS FOR DOWNLOAD

CASE STUDY

- » Renovo - Overcoming the Challenges of Safety & Security in the Renovo Automotive Data Platform

WEBSITE

- » Automotive Software Testing That Drives Success
- » Easily Automate the Tool Qualification Process
- » Conquer Cumbersome Functional Safety Compliance Standards
- » AUTOSAR Compliance With Parasoft
- » ISO 26262 Compliance With Parasoft
- » MISRA Compliance With Parasoft
- » Integrate Codebeamer and Parasoft
- » Integrate Siemens and Parasoft

WHITEPAPERS

- » ISO 26262 Software Compliance: Achieving Functional Safety in the Automotive Industry
- » Satisfying ASIL Requirements With Parasoft C/C++test – Achieving Functional Safety in the Automotive Industry
- » A Practical Guide to Accelerate MISRA C 2023 Compliance With Test Automation
- » A Practical Guide to Accelerating MISRA C 2012 Compliance With Test Automation
- » Accelerating MISRA C & SEI CERT C Compliance With Dedicated Reporting and Workflow Management
- » Using AUTOSAR C++ Coding Guidelines to Streamline ISO 26262 Compliance

» Streamlining Unit Testing for Embedded and Safety critical systems

» Embedded Cybersecurity Through Secure Coding Standards CWE and CERT

BLOG POSTS

» Expedite Your Code Coverage Task With a Coverage Advisor

» Regression Testing of Embedded Systems

» Verification vs Validation in Embedded Software

» Reducing the Risk and Cost of Achieving Compliant Software

» Qualifying a Software Testing Tool With the TÜV Certificate

» Breaking Down the AUTOSAR C++14 Coding Guidelines for Adaptive AUTOSAR

» A Smoother Road to MISRA Compliance

» The Two Big Traps of Code Coverage

» How to Select the Right Tool for AUTOSAR C++ Compliance in Support of ISO 26262

» Shift-Left Your Safety-Critical Software Testing With Test Automation

» Automotive Software Engineering Defects on the Rise

» Quantifying the Risk of Automotive Software Failures: The SRR Warranty and Recall Report

» Requirements Management and the Traceability Matrix

» Static Analysis & Coding Standards Compliance for Autonomous Driving Software

» A Practical Guide to Make Your Legacy Codebase MISRA C 2012 Compliant

WEBINARS

» Addressing ISO 26262 Compliance in Testing Automotive Software

» How Do You Develop Critical ADAS Infrastructure Systems?

» Make Your C/C++ Applications Safe and Secure With MISRA and CERT

» Requirement Traceability for Safety-Critical Applications