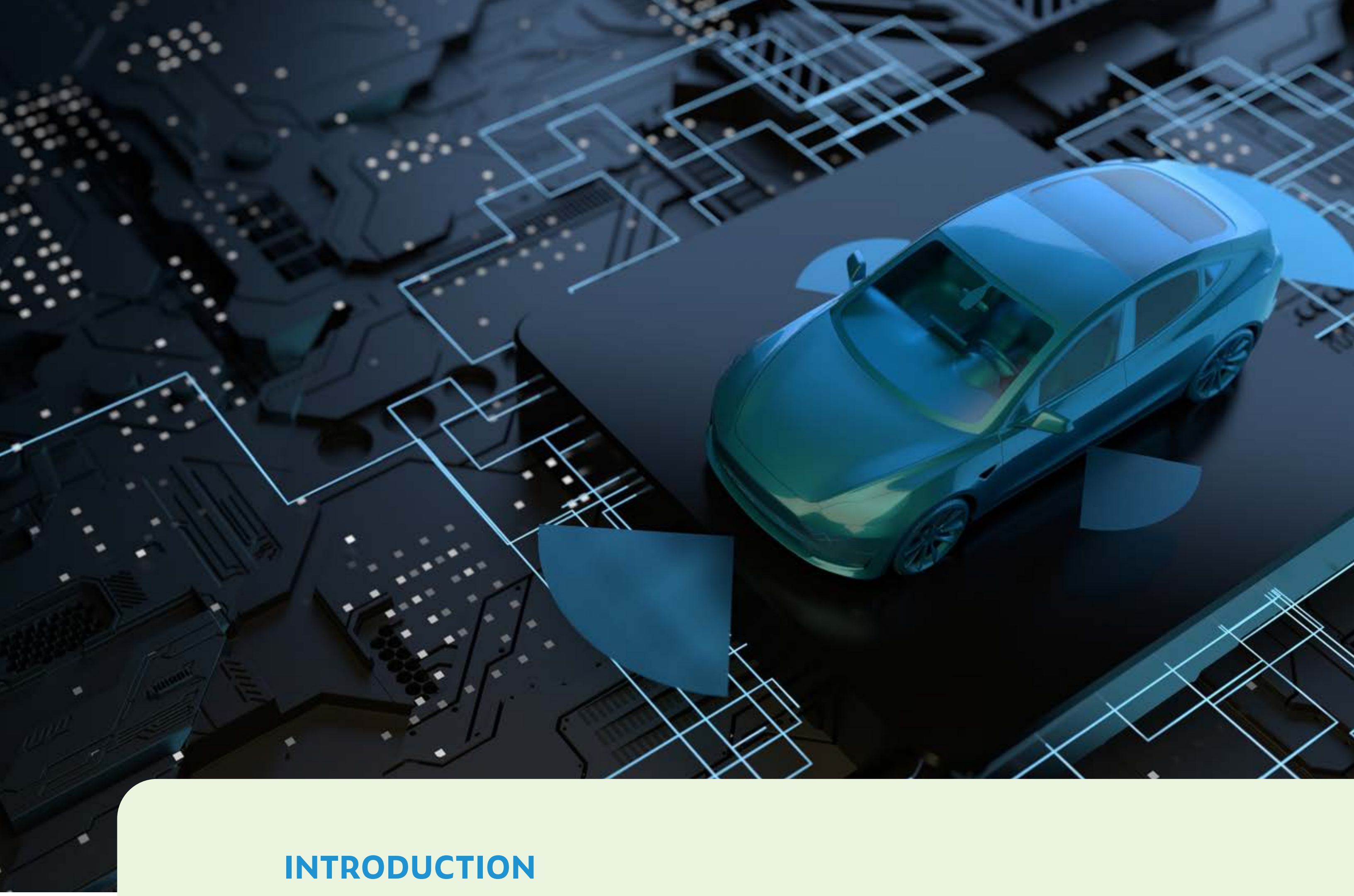
PARASOFT®

TECHNICAL WHITEPAPER

How to Satisfy ISO 26262 ASIL Requirements

Guide to Achieving Functional Safety in Automotive



INTRODUCTION

Safety functions are increasingly being carried out by electrical, electronic, or programmable electronic systems. These systems are usually complex, making it impossible in practice to fully determine every failure mode or to test all possible behavior. Although it is difficult to predict the safety performance, testing is still essential. The challenge is to design the system in such a way as to prevent dangerous failures or to control them when they arise.

Safety is one of the key issues of future automobile development. New functionality—not only in the area of driver assistance, but also in vehicle dynamics control and active and passive safety systems—increasingly touches the domain of safety engineering. Future development and integration of these functionalities will further strengthen the need to have safe system development processes and to provide evidence that all reasonable safety objectives are satisfied.

With the trend of increasing complexity, software content, and mechatronic implementation, there are rising risks of systematic failures and random hardware failures. ISO 26262 includes guidance to reduce these risks to a tolerable level by providing feasible requirements and processes.

The purpose of this document is to detail how the use of Parasoft C/C++test can help automotive software development teams meet requirements for particular ASIL levels. It first introduces the idea of ASIL as defined by the ISO 26262 standard. Next, it describes Parasoft C/C++test, a unified, fully integrated solution for automating best practices in software development and testing. Finally, it presents how teams can use Parasoft C/C++test to fully or partially satisfy software development process requirements for particular ASILs.

AUTOMOTIVE SOFTWARE INTEGRITY LEVELS

Automotive Safety Integrity Level (ASIL)—as defined by the ISO 26262 standard—is one of the four levels (1-4 in, A-D in ISO 26262) to specify the necessary safety measures for avoiding an unreasonable residual risk D representing the most stringent 1 or A the least stringent level. Note that safety integrity level is a property of a given safety function, not the property of the whole system or a system component.

Each safety function in a safety-related system needs to have an appropriate safety integrity level assigned. According to ISO 26262, the risk of each hazardous event is evaluated based on the following attributes:

» Frequency of the situation, a.k.a. “exposure”

» Impact of possible damage, a.k.a. “severity”

» Controllability

Depending on the values of these three attributes, the appropriate safety integrity level for a given functional defect is evaluated. This determines the overall ASIL for a given safety function by C/C++test functionality, which has been captured in the tables near the end of this whitepaper.

The ISO 26262 standard specifies the requirements (safety measures) for achieving each automotive safety integrity level. These requirements are more rigorous at higher levels of safety integrity in order to achieve the required lower likelihood of dangerous failures.

ABOUT PARASOFT C/C++TEST

Parasoft C/C++test is an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality. C/C++test facilitates the following.

» **Static analysis.** Performing static code analysis, data flow static analysis, and metrics analysis.

» **Unit testing.** Creating, executing, optimizing, and maintaining unit tests.

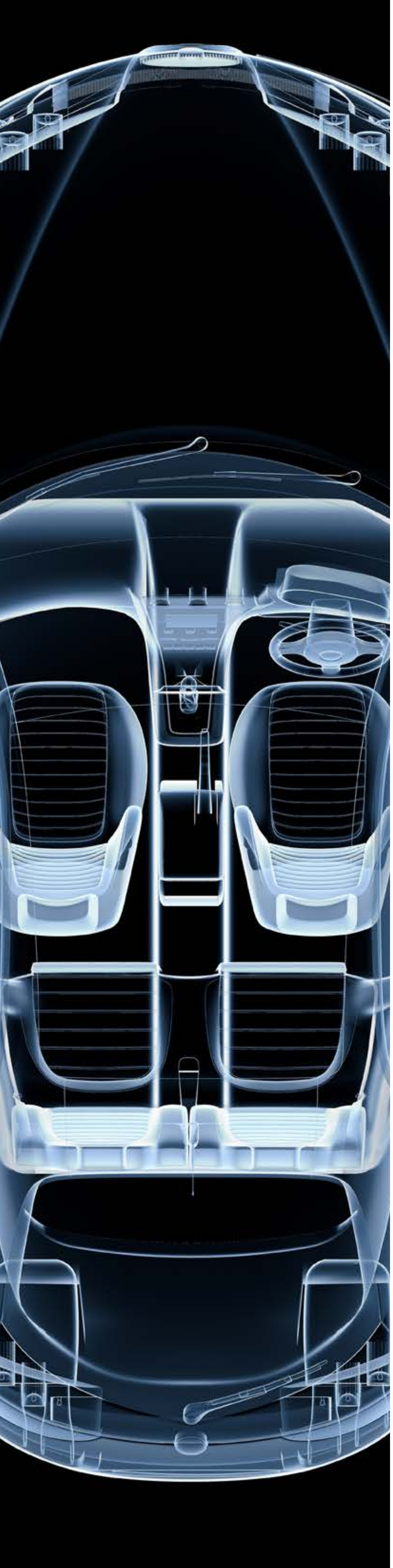
» **Code coverage.** Determines what code has been executed through a test run.

» **Requirements traceability.** Linking requirements to tests and code.

» **Runtime error.** Finding memory access errors, leaks, corruptions, and more.

This provides teams a practical way to prevent, expose, and correct errors to ensure that their C and C++ code works as expected. To promote rapid remediation, each problem detected is prioritized based on configurable severity assignments, automatically assigned to the developer who wrote the related code, and distributed to his or her IDE with direct links to the problematic code and a description of how to fix it.

For embedded and cross-platform development, C/C++test can be used in both host-based and target-based code analysis and test flows.



AUTOMATE CODE ANALYSIS FOR MONITORING COMPLIANCE

A properly implemented coding policy can eliminate entire classes of programming errors by establishing preventive coding conventions. C/C++test statically analyzes code to check compliance with such a policy. To configure C/C++test to enforce a coding standards policy specific to their group or organization, users can define their own rule sets with built-in and custom rules. Code analysis reports can be generated in a variety of formats, including HTML and PDF.

Hundreds of built-in rules including implementations of AUTOSAR C++14, MISRA C++ 2023, MISRA C 2023 standards, HIS source code metrics as well as guidelines from Meyers' Effective C++ and Effective STL books, and other popular sources—help identify potential bugs from improper C/C++ language usage, enforce best coding practices, and improve code maintainability and reusability. Custom rules, which are created with a graphical RuleWizard editor, can enforce standard API usage and prevent the recurrence of application-specific defects after a single instance has been found.

IDENTIFY RUNTIME BUGS WITHOUT EXECUTING SOFTWARE

Flow Analysis, the advanced interprocedural static analysis module of C/C++test, simulates feasible application execution paths—which may cross multiple functions and files—and determines whether these paths could trigger specific categories of runtime bugs. Defects detected include using uninitialized or invalid memory, null pointer dereferencing, array and buffer overflows, division by zero, memory and resource leaks, and various flavors of dead code. The ability to expose bugs without executing code is especially valuable for embedded code, where detailed runtime analysis for such errors is often not effective or possible.

C/C++test greatly simplifies defect analysis by providing a complete path trace for each potential defect in the developer's IDE. Automatic cross-links to code help users quickly jump to any point in the highlighted analysis path.

STREAMLINE CODE REVIEW

Code review is known to be the most effective approach to uncover code defects. Unfortunately, many organizations underutilize code review because of the extensive effort it is thought to require. Parasoft DTP Change Explorer enables convenient analyses of source code deltas between specific milestones or points in development. Overlapping code delta information with static analysis or unit testing results elevates the traditional code review process to a completely new level.

MONITOR THE APPLICATION FOR MEMORY PROBLEMS

Application memory monitoring is the best known approach to eliminating serious memory-related bugs with zero false positives. The running application is constantly monitored for certain classes of problems—like memory leaks, null pointers, uninitialized memory, and buffer overflows—and results are visible immediately after the testing session is finished.

Without requiring advanced and time-consuming testing activities, the instrumented application—additional code is added for the monitoring purposes—goes through the standard functional testing and all existing problems are flagged. The application can be executed on the target device, simulated target, or host machine. The collected problems are presented directly in the developer's IDE with the details required to understand and fix the problem, including memory block size, array index, and allocation/deallocation stack trace.

Coverage metrics are collected during application execution. These can be used to see what part of the application was tested and to fine tune the set of regression unit tests (complementary to functional testing).

This runtime error detection allows you to:

» Identify complex memory-related problems through simple functional testing. For example memory leaks, null pointers, uninitialized memory, and buffers overflows.

» Collect code coverage from application runs.

» Increase the testing results accuracy through execution of the monitored application in a real target environment.

UNIT AND INTEGRATION TEST WITH COVERAGE ANALYSIS

C/C++test's automation greatly increases the efficiency of testing the correctness and reliability of newly- developed or legacy code. C/C++test automatically generates complete tests, including test drivers and test cases for individual functions, purely in C or C++ code in a format similar to CppUnit. These tests, with or without modifications, are used for initial validation of the functional behavior of the code. By using corner case conditions, these automatically generated test cases also check function responses to unexpected inputs, exposing potential reliability problems.

Test creation and management is simplified via a set of specific GUI widgets. A graphical Test Case Wizard enables developers to rapidly create black box functional tests for selected functions without having to worry about their inner workings or embedded data dependencies. A Data Source Wizard helps parameterize test cases and stubs—enabling increased test scope and coverage with minimal effort. Stub analysis and generation is facilitated by the Stub View, which presents all functions used in the code and allows users to create stubs for any functions not available in the test scope—or to alter existing functions for specific test purposes. Test execution and analysis are centralized in the Test Case Explorer, which consolidates all existing project tests and provides a clear pass/fail status. These capabilities are especially helpful for supporting automated continuous integration and testing as well as "test as you go" development.

Both automatically generated and handwritten test cases can be used to produce a regression test base by capturing the existing software behavior via test assertions produced by automatically recording runtime test results. As the code base evolves, C/C++test reruns these tests and compares the current results with those from the originally captured "golden set." It can easily be configured to use different execution settings, test cases, and stubs to support testing in different contexts like different continuous integration phases, testing incomplete systems, or testing specific parts of complete systems.

A multi-metric test coverage analyzer, including statement, branch, function, call and MC/DC coverage, helps users gauge the efficacy and completeness of the tests. Test coverage is presented via code highlighting for all supported coverage metrics in the GUI or color-coded code listing reports. Summary coverage reports including file, class, and function data can be produced in a variety of formats.

CONFIGURABLE DETAILED REPORTING

C/C++test's HTML, PDF, and custom format reports can be configured via GUI controls or an options file. The standard reports include a pass/fail summary of code analysis and test results, a list of analyzed files, and a code coverage summary. The reports can be customized to include a listing of active static analysis checks, expanded test output with pass/fail status of individual tests, parameters of trend graphs for key metrics, and full code listings with color-coding of all code coverage results. Generated reports can be automatically sent via email, based on a variety of role-based filters. In addition to providing data directly to the developers responsible for the code flagged for defects, C/C++test sends summary reports to managers and team leads.

DTP

Code analysis and test results, coverage analysis, and other C/C++test data can be sent to Parasoft DTP where it's correlated with data generated by third-party analyzers, source control, defect tracking, and other infrastructure components and processed by DTP. The result is actionable, intelligent analytics that not only provide visibility into the risk associated with the application under test, but also the traceability required to achieve ISO 26262 compliance.

Defect review and correction are facilitated through automated task assignment and distribution. Each defect detected is prioritized, assigned to the developer who wrote the related code, and distributed to his or her IDE with full data and cross-links to code. To help managers assess and document trends, centralized reporting ensures real-time visibility into quality status and processes. This data also helps determine if additional actions are needed to satisfy internal goals or demonstrate regulatory compliance.

ACHIEVING ASIL REQUIREMENTS WITH C/C++TEST

ISO 26262 requires number of different methods to be used in the software development lifecycle of the safety functions determined to have given ASIL. Parasoft C/C++test capabilities that can be used to effectively implement these methods are described below. Please note that the information presented here is meant to briefly introduce C/C++test usage in the ASIL-related software verification process. Please refer to the standard and consult functional safety experts for clarification of any requirements defined by the ISO 26262 standard. If you have any additional questions regarding how to use C/C++test in the ISO 26262 software verification process, please contact your Parasoft representative.

The following markers are used in the tables presented below to indicate:

» (+) – functionalities matching methods recommended by the ISO 26262-6

» (++) – functionalities matching methods highly recommended by the ISO 26262-6

» o – functionalities matching methods have no recommendation by the ISO 26262-6

Methods defined in ISO 26262-6:2018 second edition that are satisfied by C/C++test functionality has been captured in the tables below.

C/C++Test Functionality	**ASIL**			
	A	**B**	**C**	**D**
Coding Standards Compliance				
Static Code Analysis *(Table 7: 1h)*	++	++	++	++
Analysis Types				
Enforcement of Low Complexity *(Table 1: 1a)*	++	++	++	++
Use of Language Subsets *(Table 1: 1b)*	++	++	++	++
Enforcement of Strong Typing *(Table 1: 1c)*	++	++	++	++
Use of Defensive Implementation Techniques *(Table 1:1g)*	+	+	++	++
Use of Well-Trusted Design Principles *(Table 1:1f)*	+	++	++	++
Use of Style Guides *(Table 1: 1g)*	+	++	++	++
Use of Naming Conventions *(Table 1: 1h)*	++	++	++	++
Specific Coding Standards Guidelines				
One Entry and One Exit Point in Subprograms and Functions *(Table 6: 1a)*	++	++	++	++

No Dynamic Objects or Variables, or Else Online Test During Their Creation *(Table 6: 1b)*	+	++	++	++
Initialization of Variables *(Table 6: 1c)*	++	++	++	++
No Multiple Use of Variable Names *(Table 6: 1d)*	++	++	++	++
Avoid Global Variables or Else Justify Their Usage *(Table 6: 1e)*	+	+	++	++
Restricted Use of Pointers *(Table 6: 1f)*	+	++	++	++
No Implicit Type Conversions *(Table 6: 1g)*	+	++	++	++
No Hidden Data Flow or Control Flow *(Table 6: 1h)*	+	++	++	++
No Unconditional Jumps *(Table 6: 1i)*	++	++	++	++
No Recursions *(Table 6: 1j)*	+	+	++	++

C/C++Test Functionality	**ASIL**			
	A	**B**	**C**	**D**
Flow Analysis: Static Data & Execution Flow Analysis				
Semi-Formal Verification *(Table 7: 1d)*	+	+	++	++
Formal Verification *(Table 7: 1e)*	o	o	+	+
Control Flow Analysis *(Table 7: 1f)*	+	+	++	++
Data Flow Analysis *(Table 7: 1g)*	+	+	++	++
Static Code Analysis *(Table 7: 1h)*	++	++	++	++
Static Analyses Based on Abstract Interpretation *(Table 7: 1i)*	+	+	+	+

C/C++Test Functionality	**ASIL**			
	A	**B**	**C**	**D**
Unit & Integration Testing				
Fault Injection Test *(Table 7: 1l - Table 10:1a - Table 14:1b)*	+	+	+	++
Resource Usage Evaluation *(Table 7: 1m - Table 10:1d)*	+	+	+	++
Analysis of Requirements *(Table 8: 1a - Table 11:1a - Table 15:1a)*	++	++	++	++
Generation and Analysis of Equivalence Classes *(Table 8: 1b - Table 11:1b - Table 15:1b)*	+	++	++	++
Analysis of Boundary Values *(Table 8: 1c - Table 11:1c - Table 15:1c)*	+	++	++	++
Error Guessing Based on Knowledge or Experience *(Table 8: 1d - Table 11:1d - Table 15:1d)*	+	+	+	+
Structural Code Coverage				
Statement Coverage *(Table 9: 1a)*	++	++	+	+
Branch Coverage *(Table 9: 1b)*	+	++	++	++
MC/DC (Modified Condition/Decision Coverage) *(Table 9: 1c)*	+	+	+	++
Function Coverage *(Table 12: 1a)*	+	+	++	++
Call Coverage *(Table 12: 1b)*	+	+	++	++

C/C++Test Functionality	ASIL			
	A	B	C	D
Test Environment: On-Target Hardware				
Hardware-in-the-Loop *(Table 13: 1a)*	++	++	++	++

C/C++Test Functionality	ASIL			
	A	B	C	D
Parasoft DTP: Reporting & Analytics Dashboard				
Walk Through *(Table 7: 1a)*	++	+	o	o
Paired Programming *(Table 7: 1b)*	+	+	+	+
Inspection *(Table 7: 1c)*	+	++	++	++
Requirements-Based Test *(Table 7: 1j - Table 10,:1a - Table 14:1a)*	++	++	++	++

SUMMARY

Parasoft C/C++test helps automotive software development teams achieve ISO 26262 verification and validation requirements. A broad range of test methods like static analysis, data and control flow analysis, unit testing, application monitoring, workflow components, and peer code review process—together with the configurable test reports containing a high level of details—significantly facilitates the work required for the software verification process.

TAKE THE NEXT STEP

Request a demo to see how your automotive software development team can satisfy ASIL requirements and achieve functional safety.

ABOUT PARASOFT