





PARASOFT WHITE PAPER

Topic: Satisfying ASIL Requirements with Parasoft C++test
Achieving Functional Safety in the Automotive Industry

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/DIS 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++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/DIS 26262 standard. Next, it describes Parasoft C++test: an integrated solution for automating best practices in software development and testing. Finally, it presents how Parasoft C++test can be used to fully or partially satisfy software development process requirements for particular ASILs.

Automotive Software Integrity Levels

Safety Integrity Level (SIL)—as defined by the IEC 61508 standard—or Automotive Safety Integrity Level (ASIL)—as defined by the ISO/DIS 26262 standard—is one of the four levels (1-4 in IEC 61508, A-D in ISO/DIS 26262) to specify the necessary safety measures for avoiding an unreasonable residual risk with 4 or D representing the most stringent and 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/DIS 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.

The ISO/DIS 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++test

Parasoft 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++test facilitates:

Static analysis – static code analysis, data flow static analysis, and metrics analysis

Safety in software development process is one of the key issues of future automobile development.

continued on page 2

- Peer code review process automation preparation, notification, and tracking
- Unit testing unit test creation, execution, optimization, and maintenance
- Runtime error detection memory access errors, leaks, corruptions, and more

This provides teams a practical way to prevent, expose, and correct errors in order 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++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++test statically analyzes code to check compliance with such a policy. To configure 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 MISRA, MISRA 2004, and the new MISRA C++ 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

BugDetective, the advanced interprocedural static analysis module of 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++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. The C++test Code Review module automates preparation, notification, and tracking of peer code reviews, enabling a very efficient team-oriented process. Status of all code reviews, including all comments by reviewers, is maintained and automatically distributed by the C++test infrastructure.

C++test supports two typical code review flows:

- Post-commit code review. This mode is based on automatic identification of code changes in a source repository via custom source control interfaces, and creating code review tasks based on pre-set mapping of changed code to reviewers.
- Pre-commit code review. Users can initiate a code review from the desktop by selecting a set of files to distribute for the review, or automatically identify all locally changed source code.

The effectiveness of team code reviews is further enhanced through C++test's static analysis capability. The need for line-by-line inspections is virtually eliminated because the team's coding policy is monitored automatically. By the time code is submitted for review, violations have already been identified and cleaned. Reviews can then focus on examining algorithms, reviewing design, and searching for subtle errors that automatic tools cannot detect.

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, allocation/deallocation stack trace etc.)

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).

With the trend of increasing complexity, software content, and mechatronic implementation, there are rising risks of systematic failures and random hardware failures.

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++test's automation greatly increases the efficiency of testing the correctness and reliability of newly-developed or legacy code. 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 hand-written 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++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 (e.g., different continuous integration phases, testing incomplete systems, or testing specific parts of complete systems).

A multi-metric test coverage analyzer, including statement, branch, path, and MC/DC coverage, helps users gauge the efficacy and completeness of the tests, as well as demonstrate compliance with test and validation requirements, such as DO-178B. 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++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++test sends summary reports to managers and team leads.

Efficient Team Deployment

C++test establishes an efficient process that ensures software verification tasks are ingrained into the team's existing workflow and automated—enabling the team to focus ontasks that truly require human intelligence. 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.

Team Workflow Module

Parasoft C++test integrates with Parasoft Concerto, which helps in configuring workflows for achieving compliance versus defined policies and process standardization, including CMMI and SPICE. Concerto provides full traceability of source code, automated tests, and manual tests to all project artifacts: requirements, defects/enhancements, and tasks. By tracking and analyzing software development metrics and progress, it determines which activities are costly and time-consuming and enables development teams to rapidly identify specific problems and problem areas.

ISO/DIS 26262 standard specifies the safety measures for achieving each automotive safety integrity level.

Achieving ASIL requirements with C++test

ISO/DIS 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++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++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/DIS 26262 standard. If you have any additional questions regarding how to use C++test in the ISO/DIS 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/DIS 26262-6
- (++) functionalities matching methods highly recommended by the ISO/DIS 26262-6

C++test functionality descriptions reference the appropriate requirement or method of the ISO/DIS 26262-6, for example (Table 10: 1f) reference ISO/DIS 26262-6, Method 1f from Table 10.

Coding standards compliance - static code analysis

| C++test functionality | ASIL | | | |
|---|------|------|------|------|
| | A | В | С | D |
| Coding standards compliance module – general | | | | |
| Verifying software implementation using static code analysis (Table 10: 1f) | (+) | (++) | (++) | (++) |
| Analysis types | | | | |
| Using code metrics (e.g. cyclomatic complexity, essential complexity, etc.) to enforce low complexity of the code (Table $1\colon 1a$) | (++) | (++) | (++) | (++) |
| Using coding standards to enforce using only a subset of the language, e.g. to avoid unsafe constructions (Table 1: 1b) | (++) | (++) | (++) | (++) |
| Enforcement of specific naming conventions (Table 1: 1h) | (++) | (++) | (++) | (++) |
| Enforcement of specific coding conventions (Table 1: 1g) | (+) | (++) | (++) | (++) |
| Enforcement of specific formatting conventions (Table 1: 1f) | (+) | (++) | (++) | (++) |
| Enforcement of industry-known coding standards rule sets, such as MISRA C/C++, JSF, HIS source code metrics, etc. (Table 1: 1e) | (+) | (+) | (+) | (++) |
| Enforcement of defensive programming using appropriate coding standards rules, for example, checking the return value of malloc, checking the error code value returned by called functions, etc. (Table 1: 1d) | | (+) | (++) | (++) |
| Specific coding standards guidelines | | | | |
| Finding implicit conversions to enforce strong typing (Table 1: 1c; Table 9: 1g) | (++) | (++) | (++) | (++) |
| Finding multiple exit points in functions (Table 9: 1a) | (++) | (++) | (++) | (++) |
| Reporting possible variable initialization problems (Table 9: 1c) | (++) | (++) | (++) | (++) |
| Finding unconditional jumps (Table 9: 1i) | (++) | (++) | (++) | (++) |
| Reporting unsafe usage of dynamic objects (Table 9: 1b) | (+) | (++) | (++) | (++) |
| Finding possible variable name ambiguity (Table 9: 1d) | (+) | (++) | (++) | (++) |
| Reporting of hidden data flow (Table 9: 1h) | (+) | (++) | (++) | (++) |
| Finding global variable usage (Table 9: 1e) | (+) | (+) | (++) | (++) |
| Reporting recursive functions (Table 9: 1j) | (+) | (+) | (++) | (++) |
| Reporting unsafe pointer usage (Table 9: 1f) | | (+) | (+) | (++) |

Parasoft C++test helps automotive software development teams meet requirements for particular ASIL levels.

| C++test functionality | ASIL | | | | |
|---|------|-----|-----|------|--|
| | A | В | С | D | |
| Enforcement of defensive implementation techniques – for example, checking the return value of malloc, checking the error code value returned by called functions, etc. (Table 1: 1d) | | (+) | (+) | (++) | |

Bug Detective - static data and execution flow analysis

| Contract Company Company | ASIL | | | | |
|--|------|------|------|------|--|
| C++test functionality | Α | В | С | D | |
| Bug Detective - general | | | | | |
| Verifying software implementation using data flow analysis (Table 10: 1e) | (+) | (+) | (++) | (++) | |
| Verifying software implementation using control flow analysis (Table 10: 1d) | (+) | (+) | (++) | (++) | |
| Analyzing source code using an abstract representation of possible values for the variables – semantic analysis (Table 10: 1g) | (+) | (+) | (+) | (+) | |
| Bug Detective – specific rule examples | | | | | |
| Reporting uninitialized variable usage (Table 9: 1c) | (++) | (++) | (++) | (++) | |
| Reporting erroneous pointer usage (Table 9: 1f) | | (+) | (+) | (++) | |

Automated peer code review

| C++test functionality | ASIL | | | | |
|--|------|------|------|------|--|
| | Α | В | С | D | |
| Automated peer code review module - general | | | | | |
| Inspection of the source code using automated peer code review module (Table $11:1e$) | (+) | (++) | (++) | (++) | |
| Walkthrough of the source code using automated peer code review module (Table 11: 1f) | (++) | (+) | | | |

Unit testing

| C++test functionality | ASIL | | | | |
|---|---|------|------|------|--|
| | A | В | C | D | |
| Unit testing module – general | | | | | |
| Unit tests execution and reporting the results of the executed unit tests (requirement 9.4.1) | No specific recommendations for particular ASILs | | | | |
| Ability to execute unit tests with and without instrumentation to verify that, for example, coverage instrumentation does not impact the test results (requirement 9.4.4) | No specific recommendations for particular ASILs | | | | |
| Execution of unit tests in the production environment on a target device or on a simulator (requirement 9.4.5) | No specific recommendations for particular ASILs | | | | |
| Automatic unit tests generation module | , | | | | |
| Automatic unit tests generation using boundary values (Table 13: 1c) | (+) | (++) | (++) | (++) | |
| Automatic unit tests generation using heuristic values (Table 12: 1c) | (+) | (+) | (+) | (++) | |
| Using factory functions to prepare sets of input parameter values for automatic unit test generation (Table 12: 1b) | (+) | (+) | (+) | (+) | |
| Test management module | | | | | |

Parasoft C++test helps automotive software development teams meet requirements for particular ASIL levels.

| C++test functionality | ASIL | | | | |
|--|------|------|------|------|--|
| | A | В | C | D | |
| Mapping test cases with requirements and/or defects – in cooperation with the Team Workflow module (Table 12: 1a) | (++) | (++) | (++) | (++) | |
| Using user-defined test cases – both manually-written and created using Test Case Wizard – to test specific atomic cases of the given requirement (Table 13: 1a) | (++) | (++) | (++) | (++) | |
| Using Data Sources to efficiently provide multiple inputs for functionally equivalent atomic cases of the requirement (Table 13: 1a) | (++) | (++) | (++) | (++) | |
| Using Data Sources to emulate behavior of external components for automatic unit test execution (Table 12: 1b) | (++) | (++) | (++) | (++) | |
| Function stubs | | | | | |
| Using stubs to control flow of the executed tests as specified in the given requirement (Table 13: 1a) | (++) | (++) | (++) | (++) | |
| Using function stubs to emulate behavior of external components for automatic unit test execution (Table 12: 1b) | (++) | (++) | (++) | (++) | |
| Using stubs to provide fault conditions in tests (Table 12: 1c) | (+) | (+) | (+) | (++) | |
| Coverage module | | | | | |
| Analyzing MC/DC code coverage (Table 14: 1a) | (+) | (+) | (+) | (++) | |
| Analyzing branch code coverage (Table 14: 1b) | (+) | (++) | (++) | (+) | |
| Analyzing statement code coverage (Table 14: 1c) | (++) | (++) | (+) | (+) | |

Parasoft C++test helps automotive software development teams meet requirements for particular ASIL levels.

Application monitoring

| Contract Company of the | | ASIL | | | | |
|--|-----|---|------|------|--|--|
| C++test functionality | A | В | C | D | | |
| Application monitoring module - general | | | | | | |
| Monitoring of the running application reporting runtime problems (requirement 10.4.2) | | No specific recommendations for particular ASILs | | | | |
| Application monitoring in the production environment on a target device or on a simulator (requirement 10.4.7) | | No specific recommendations for particular ASILs | | | | |
| Coverage module | • | | | | | |
| Analyzing function coverage (Table 17: 1a) | (+) | (+) | (++) | (++) | | |

Summary

Parasoft C++test helps automotive software development teams achieve ISO/DIS 26262 compliance and meet the SOP of the embedded software. A broad range of analysis types—including coding standards compliance analysis, data and control flow analysis, unit testing, application monitoring, workflow components, and automated peer code review process—together with the configurable test reports containing high level of details, significantly facilitates the work required for the software verification process.

About Parasoft

For over 20 years, Parasoft has investigated how and why software defects are introduced into applications. Our solutions leverage this research to dramatically improve SDLC productivity and application quality. Through an optimal combination of quality tools, configurable workflow, and automated infrastructure, Parasoft seamlessly integrates into your development environment to drive SDLC tasks to a predictable outcome. Whether you are delivering code, evolving and integrating business systems, or improving business processes—draw on our expertise and award—winning products to ensure that quality software can be delivered consistently and efficiently.

For more information, visit http://www.parasoft.com.

Contacting Parasoft

GLOBAL HEADQUARTERS

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

USA

Toll Free: (888) 305-0041 Tel: (626) 305-0041 Fax: (626) 305-3036

Email: info@parasoft-embedded.com URL: http://www.parasoft-embedded.com

URL: http://www.parasoft.com

Europe Headquarters

Parasoft SA

Kielkowskiego 9, Krakow 30-704

Poland

Phone +48 12 290 91 01 Fax +48 12 290 91 02

Email: info-pl@parasoft.com

ASIA HEADQUARTERS

11F, 508 Chung Hsiao E. Rd. Section 5

Taipei, Taiwan

Phone (02) 6636-8090

Email: info-psa@parasoft.com

Parasoft SA (France)

Chateau de Sainte Assise, 77240 Seine Port

France

Phone (33 1) 64 89 26 00 Fax (33 1) 64 89 26 10 Email: sales@parasoft-fr.com

Parasoft Deutchland GmbH

Mähringer Weg 45 89075 Ulm Germany

Phone +49 89 4613323-0 Fax +49 89 4613323-23 Email: info-de@parasoft.com

Parasoft UK Ltd

The Lansdowne Building 2 Lansdowne Road Croydon. CR9 2ER United Kingdom Phone +44 (0)208 263 6005

Fax +44 (0)208 263 6100 Email: sales@parasoft-uk.com

Other Locations

See http://www.parasoft.com/contacts

© 2010 Parasoft Corporation

All rights reserved. Parasoft and all Parasoft products and services listed within are trademarks or registered trademarks of Parasoft Corporation. All other products, services, and companies are trademarks, registered trademarks, or servicemarks of their respective holders in the US and/or other countries.

Parasoft C++test
helps automotive
software
development teams
meet requirements
for particular
ASIL levels.