PARASOFT®

WHITEPAPER

GoogleTest Adoption Challenges for Safety-Critical Code

INTRODUCTION

If your team works on safety-critical C/C++ software, you apply unit testing to ensure you're implementing the system correctly and adhering to software requirements. Unit testing is typically implemented with dedicated commercial or open-source frameworks.

Open-source frameworks are leaner than commercial solutions. They're based entirely on the source code. Commercial solutions, on the other hand, are feature-rich, tightly integrated with IDEs, and designed to satisfy safety-critical standards requirements, which include support for tool qualification.

Feature-rich and IDE-based tools became problematic to integrate with the growing complexity of projects, the prevalence of modern C++, and complex distributed build systems. These difficulties have led the industry to focus on open-source unit testing frameworks, which align with the general trend of increasing open-source usage in safety-critical software development. GoogleTest is an example of an open-source framework that's lightweight and easily integrates with complex builds.

Adopting GoogleTest for compliance with safety-critical standards is not trivial. There are two main challenges that we'll cover in this paper.

1. We'll analyze options to extend GoogleTest with functionalities that are critical for standards compliance.

2. We'll explore various aspects related to the tool qualification, which is a formal process of approving tool for use in safety-critical software development.

In our considerations, we will refer to ISO 26262 and DO-178C as examples of industry standards. However, the intent of this paper isn't to provide a complete qualification process overview for any of the standards. It includes standard references only to help visualize certain aspects.

Understanding Standards Requirements for Unit Level Testing

Functional safety and process standards differ in detail when it comes to recommended methods and techniques for software testing at the unit level. The testable unit is often debatable. Standards intentionally abstract from providing a precise definition of “unit” since it may depend on selected technologies and architectural design.

The engineering understanding of unit and unit testing is frequently focused on functions and methods, but from the standards compliance point of view, teams may define units at a higher level. It can be a class, an entire module or even a service. However, the spirit of the standards is that a unit shall be the smallest architectural block with a clearly defined interface that can be tested using, for example, a unit testing framework.

The ISO 26262 standard, used in the automotive industry, leaves little wiggle room for organizations that prefer to skip unit testing. The standard clearly requires testing at the unit level and recommends several techniques that are expected to be deployed together:

- Software unit verification
- Structural coverage metrics
- Fault injection testing
- Requirements-based testing

This isn’t a full list, but the absolute minimum. The standard recommends deploying more techniques and metrics with unit testing, such as back-to-back testing and resource usage evaluation.

The DO-178C standard, used in aviation, is less precise and speaks only about low-level software testing. Assuming that low-level testing is equivalent to unit testing—which is typical but not the only interpretation of the standard—the review of the standard brings a similar core set of testing techniques as in the case of ISO 26262.

- Low-level tests
- Structural coverage metrics
- Fault injection, robustness, and fault tolerance testing
- Requirements-based testing/ requirements traceability reporting

There are some other techniques and metrics that teams can deploy for low-level testing to meet DO-178C objectives. For example, for software Level A, the standard requires monitoring object/assembly coverage to ensure the complete inspection of the machine code added by the compiler. In typical scenarios, however, the four testing techniques listed above are the foundation of testing.

Other standards like IEC 62304 or IEC 61508 do not provide any additional insights regarding low level or unit testing. The common practice across regulated, safety-critical software development is that the software testing toolchain includes:

- A framework to test correctness of "units" implementation against low-level requirements
- Tools for monitoring various structural code coverage metrics
- Tools that enable tested software isolation and fault injection/fault tolerance testing
- A reporting framework that enables corelating unit tests with requirements and presenting test execution results in the context of requirements

Advantages & Disadvantages of Applying GoogleTest for Safety-Critical Software Development

GoogleTest provides rich functionality for defining test cases, supporting various test source strategies required by standards. ISO 26262 recommends the following four methods for deriving test cases for software unit testing.

1. Requirements analysis
2. Generation and analysis of equivalence classes
3. Boundary values analysis
4. Error guessing based on knowledge and experience

Commercial unit testing frameworks typically support all four of these methods, providing some level of automation for methods 2, 3, and 4. The main challenge, however, is that some of these frameworks force proprietary APIs for test case development.

These APIs abstract from the source code, which in theory should enable test portability, but, in practice, they significantly hamper test creation for modern C++. Initializing STL containers with complex data objects, using lambdas as inputs for tests, and many other aspects of modern C++ are exceedingly difficult to express clearly with frameworks that utilize data-driven, proprietary APIs.

The advantage of commercial frameworks is in the test generation capabilities. These tools can typically analyze the tested source code, produce tests that exercise boundary values, and try to increase code penetration by guessing errors. This generation works relatively well for C and simple C++ code but is less effective for complex C++ interfaces of industry platforms such as Adaptive AUTOSAR.

Advanced generation capabilities also require strong integration of the unit testing tool with the build process to obtain the code parsing data needed for test generation. The complexity of integration is often a disqualifying factor.

GoogleTest, on the other hand, does not provide any automation for deriving test cases following methods 1 through 4. It wins, however, with its flexibility and modern C++-based API for test creation. Test scenarios with complex data structures are simpler and cleaner to create with GoogleTest and more manageable to review. These advantages scale with the size of the development team and the number of created test cases.

Tool qualification is an aspect of safety-related GoogleTest applications that requires special attention. Commercial frameworks are precertified and provide qualification packages that either eliminate or highly simplify the qualification effort. The open-source community does not provide this kind of support for GoogleTest. We discuss this problem later in "Qualifying GoogleTest for Use in Regulated Software Development."

Parasoft's experience interacting with leading automotive and aerospace companies shows that teams strongly prefer GoogleTest's ease of use and clarity of tests over basic test generation capabilities. The ease of build system integration is often raised as a key advantage, especially by teams using the Bazel build system.

Extending GoogleTest to Meet Compliance Requirements

Assuming the requirements previously discussed, several options exist for composing the GoogleTest-based testing tool chain that satisfies regulated industries' criteria. The options alternate between open-source and commercial products to augment GoogleTest with missing capabilities.

Structural Code Coverage Tool

Code coverage reports document the completeness of testing and provide evidence that there is no source code that has not been tested. Code coverage reports are essential for standards compliance. Organizations planning to utilize GoogleTest for standards compliance need to extend the framework with a code coverage component.

Important criteria to consider when selecting the code coverage tool:

- Support for specific compilers and required coverage metrics
- Ease of build integration
- Reporting with support for merging results, coverage gaps annotations, and IDE-based presentation
- Support for the tool qualification

The above criteria should be analyzed when deciding on a tool for GoogleTest code coverage. Open-source tools like GCOV or LLVM-COV are becoming an interesting alternative to commercial solutions, especially with the recent addition of MC/DC coverage support in LLVM-based tools and the prospects of adding similar support to GCC-based tools.

Another advantage of open-source tools is that they are integrated with compilers, simplifying the build integration effort. However, open-source coverage utilities still lack maturity and some capabilities for merging coverage sessions, advanced graphical results presentation in the IDEs, and a system of annotations to indicate accepted coverage gaps. The most significant disadvantage is the lack of support for the qualification of the tool.

Parasoft's experience interacting with leading organizations developing safety-critical software shows that coverage tool qualification is considered highly problematic. Teams prefer precertified commercial solutions or to qualify using commercially provided qualification kits.

Fault Injection/Fault Tolerance Testing

Fault injection testing or fault tolerance testing is one of the recommended methods for testing at the unit level. The intention of the fault injection testing is to exercise mechanisms that are designed to bring the system to a safe state in the presence of unexpected errors. DO-178C speaks about it in the following way:

> "The goal of fault tolerance testing methods is to include safety features in the software design or Source Code to ensure that the software will respond correctly to input data errors and prevent output and control errors."

The fault injection for unit testing is typically implemented with stubbing/mocking frameworks. These frameworks enable substituting a function or method called from the tested code with an alternative implementation that can simulate an error. For example, a function reading data from a hardware sensor can be stubbed for specific test scenario execution to isolate the test from the hardware and to enable simulating sensor error. These capabilities are critical for the unit testing toolchain.

The important criteria for the stubbing/mocking frameworks include:

- Ability to replace all types of calls including global functions, static functions and class methods
- Ability to define test case specific behavior for mocked/stubbed calls
- Ease of build integration
- Support for the tool qualification

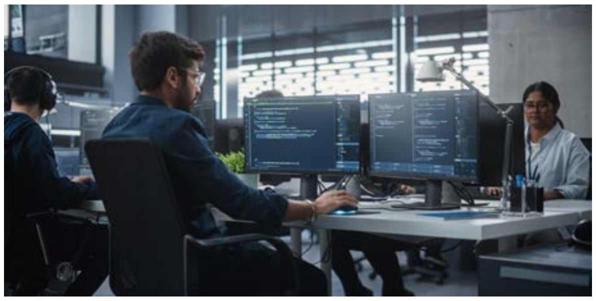
There are a few commercial tools for stubbing, and several options exist in open source. Some commercial tools enable test doubles injections through code instrumentation, and some operate by modifying symbol tables in binary object files. Open-source tools typically enable stubbing/mocking using C/C++ language-provided mechanisms such as virtual methods, inheritance, and function pointers substitution. Some open-source examples are CppUMock, FFF (Fake Function Framework), CMock, and GoogleMock.

In the case of a GoogleTest-based testing toolchain the obvious choice is the GoogleMock framework that is a part of the GoogleTest repository. The Gmock framework provides a broad mocking capabilities that are in most cases sufficient for the fault injection testing, assuming the code base follows mocking friendly patterns.

GoogleMock is designed for mocking virtual class methods. It's expected to work very well for code bases that follow the virtual interfaces paradigm. It's possible to use GoogleMock for free functions (C-style), but it may require some additional scaffolding and changes in the code base, which some teams may see as a significant disadvantage. In these situations, it may be justified to apply one of the commercial products. Another advantage of commercial products over open-source stubbing/mocking frameworks, as in the case of coverage tools, is support for the tool qualification.

Requirements-Based Traceability Reporting

Requirements-based traceability reporting is fundamental for safety-critical software development. To comply with popular standards, teams must document that all requirements were implemented, tested, and that no code is unrelated to any of the requirements. The reporting framework must support traceability links between requirements, source code implementing requirements, and test cases.



These correlations enable the generation of bidirectional traceability reports critical to the compliance processes. Some functionalities, like correlating low-level requirements with source code implementing these requirements, are typically implemented only for testing at a unit level. However, the general capability to trace requirements to test cases and results is relevant for any type of testing, including integration and system-level testing.

There are commercial reporting frameworks provided by software testing tool vendors that satisfy all of the above requirements. These solutions combine traceability reporting with advanced test results analytics. Some organizations, however, decide to fulfill these reporting requirements with home-grown solutions. There are two challenges with home grown reporting solutions:

1. Correlating requirement with source code implementation.
2. Correlating requirements with tests.

To satisfy both traceability links, a system of annotations must be developed. Based on Parasoft's interactions with organizations that decided to build this kind of reporting system, the correlation between requirements and the source code is typically solved by annotating SCM commits with appropriate comments that include requirement IDs.

The requirement to test case correlation, in case of GoogleTest, is frequently solved with help of GoogleTest API "Test::RecordProperty" method. The RecordProperty method is utilized to propagate test specific requirement ID information to the test's execution log. The homegrown reporting frameworks provide much flexibility in designing reports and structuring the information but require costly maintenance, which for many teams is disqualifying.

The main challenge, however, is the tool qualification. These frameworks generate critical compliance reports that provide evidence for completeness of testing and must be qualified. Commercial solutions come precertified or have qualification kits that reduce the effort.

The summary of the GoogleTest extensions overview is that each organization planning to deploy GoogleTest for safety-critical software development needs to assess the cost and effort of deploying a commercial, open-source, or mixed solution. The qualification cost of open-source toolchain components must be carefully considered. The following section discusses the main challenges of certifying open-source tools on example of GoogleTest framework.

Qualifying GoogleTest for Use in Regulated Software Development

Tool qualification is a formal process mandated by safety standards to ensure that the risk of tool-related errors impacting system safety is acceptably low. The open-source community does not provide any support for the GoogleTest qualification. This section discusses the key challenges of GoogleTest unit testing framework qualification. The qualification process considerations from this section also apply to other open-source tools that may potentially be used as an extension of the GoogleTest, as discussed above in "Extending GoogleTest to Meet Compliance Requirements."

The qualification process must include tool classification and validation and conclude with appropriate documentation. The documentation package is expected to contain guidelines for deployments in critical development, frequently called the Tool Safety Manual (TSM). Below, we discuss the main aspects of tool classification and validation, and the content of the documentation package.

Tool Classification

Tool classification must be performed in the context of a specific project. It determines the qualification rigor and the content of qualification documents. The classification depends on the tool use cases and additional measures like process or tool redundancy. These properties may differ between projects; hence, it's not possible to pre-classify a tool without understanding its deployment details.

The complete discussion of the classification process for any of the safety standards is beyond the scope of this paper. However, the common part is the tool impact analysis. With some simplification, standards define it as the probability of the tool introducing an error to the system or failing to detect an error. Unit testing frameworks like GoogleTest do not generate code that goes to the target system, so there is no possibility of introducing an error. However, they can fail to detect errors. A malfunctioning assertion, for example, may report a failing test as passed, resulting in faulty code deployed to the product.

ISO 26262 defines three confidence levels for software tools, TCL one to three. Level three imposes the highest qualification rigor. The conservative unit testing framework classification for ISO 26262 is TCL3, but TCL2 can also be achieved by extending the process with additional measures for error detection. The DO-178C standard uses five levels (TQL) to define qualification rigor. TQL1 represents the highest risk, and TQL5 represents the lowest risk. Based on Parasoft's experience, unit testing frameworks are typically classified as TQL5 for DO-178C compliance.

The tool classification determines appropriate qualification method and rigor. The canonical approach to qualification is to validate the tool against its operational requirements, which is expected to be performed in the project's environment.

The ISO 26262, IEC 62304, and IEC 61508 standards allow tool precertification to be a qualification method in most cases. Tool vendors acquire tool certificates by evaluating the tool development process with organizations such as TÜV and offering them to the end users. Certificates greatly simplify the qualification process. However, the aviation standard, DO-178C, does not honor precertification and mandates the qualification through validation. Also, in the case of ISO 26262, IEC 61508, or IEC 62304 for the top criticality systems, tool certification may not be accepted by the certification body and tools may need to be qualified through validation.

Tool Qualification Through Validation

The essence of tool qualification through validation is proving that the tool behaves as declared in the operational requirements. Hence, the probability of tool errors that could affect the developed system is acceptably low. The two most important aspects of this process are tool operational requirements and tests that validate these requirements.

For the GoogleTest qualification, both aspects require significant investment. GoogleTest comes with good quality documentation, but no formal requirements are available that could be the basis for the validation process. Available tests are difficult to assess for consistency and completeness. Organizations planning to qualify GoogleTest must develop requirements documentation relevant to their use cases and validation tests. Below, we discuss the key challenges for deriving consistent requirements and test cases for GoogleTest.

Defining Tool Operational Requirements

Requirements play a critical role in safety processes. It's essential to ensure their appropriate quality. In the context of tool qualification, requirements precisely specify what has been validated and can be safely used. Tool documentation, like the GoogleTest user guide, cannot replace requirements because they play different roles. Writing formal requirements for preexisting open-source code is costly and challenging.

Parasoft’s experience with requirements specification for GoogleTest resulted in a three-phased process:

1. Tool architectural analysis and API elements identification
2. Grouping of API elements and defining requirements seeds
3. Actual requirements specification

The architectural analysis helped identify the framework's functional blocks and user-facing API elements. The API elements were then grouped into functionality-based classes and smaller subgroups focused on specific user goals. Each subgroup was used as a seed for the requirement definition. The architectural analysis and API elements groupings helped develop a requirement document structure and a consistent strategy for assigning requirement IDs.

Requirements were developed based on GoogleTest public documentation. For situations where the documentation was not exhaustive enough, dedicated experiments and code reviews were conducted. Each requirement was linked to an API element, source code, and user documentation to facilitate maintenance. Traceability links to API elements (source code) and user documentation help identify changes between GoogleTest releases that may invalidate requirements. Additionally, traceability to test cases was also established to document completeness of qualification.

A significant challenge during the process was capturing the enormous flexibility and extensibility of the GoogleTest framework in the requirements specification. The framework uses modern C++ with templates, allowing for difficult-to-predict code constructs. Parasoft's C++ experts were critical in the requirements refining process.

The complete analysis of the GoogleTest framework resulted in 1500+ requirements.   The general recommendations for organizations planning to conduct similar processes are:

- Ensure the team is highly skilled in modern C++.
- Pay attention to requirements structure and proper granularity. Requirements that are too coarse are difficult to validate and those that are too fine-grained result in significant overhead.
- If qualifications for future GoogleTest versions are planned, invest in correlating defined requirements to GoogleTest source code and potentially also to user documentation to reduce the effort of identifying changes that may affect requirements.

Developing Test Cases for Requirements Validation

The qualification test cases must be derived from the requirements since they are used for validation. The preexisting test cases shipped with the GoogleTest framework are generally of high quality, but they are difficult to use directly in the qualification process.

The granularity of existing tests may not reflect the granularity of developed requirements. Mapping existing tests to requirements is time-consuming. In many situations, there is not enough gravity on negative tests—proving that the framework correctly handles incorrect inputs.

Parasoft's experience from the process is that most test cases for requirements validation were developed from scratch, and very few preexisting ones were reused and mapped to requirements.

Developing requirements-based test cases posed several challenges. One of the challenges for Parasoft's team was developing a testing strategy for dealing with API call-chains. GoogleTest is a template-based framework offering enormous flexibility and extensibility. Exhaustive testing of all possible API combinations is impossible. Simplifying testing by imposing limitations on API use would highly undermine the qualification value. The EXPECT_CALL GoogleTest (Mock) macro visualizes the challenge:

```
EXPECT _ CALL(mock _ object, method _ name(matchers...))
   .With(multi _ argument _ matcher)  // Can be used at most once
   .Times(cardinality)                // Can be used at most once
   .InSequence(sequences...)          // Can be used any number of times
   .After(expectations...)            // Can be used any number of times
   .WillOnce(action)                  // Can be used any number of times
   .WillRepeatedly(action)            // Can be used at most once
   .RetiresOnSaturation();            // Can be used at most once
```

Figure 1: EXPECT_CALL macro syntax

The EXPECT_CALL macro takes parameters and can be fine-tuned with a collection of modifiers like "With," "InSequence," or "WillOnce." Most modifiers can take parameters, so the total number of combinations, even if equivalence class partitioning is used for inputs, results in an astronomical number of test cases.

The testing effort could be simplified in several situations by providing additional argumentation derived from GoogleTest implementation. This approach was intentionally avoided because of the lack of control over the open-source project development and implementation decisions. Whatever implementation-based reasoning would be provided to reduce the testing effort, future changes in the framework implementation could invalidate the argumentation.

To cope with the testing complexity and the large number of possible combinations in GoogleTest API call-chains, the Parasoft team applied the following strategies.

All-Pairs Approach

In the case of long call chains, the All-Pairs methodology was applied to ensure that each combination of two different modifiers from the whole set is tested at least once instead of testing all possible combinations.

MC/DC Coverage-Inspired Strategy

The MC/DC coverage-inspired strategy was applied to Assertions/Expectations long call chains. In this approach, the entire call chain was treated as a decision and each modifier in the call chain was treated as a condition in a decision. The testing effort focused on proving that each individual modifier in the call chain can be demonstrated to affect the Assertion/Expectation outcome with other modifiers being unchanged.

Both the all-pairs testing approach and MC/DC strategy enabled a significant reduction in the number of test cases for complex API invocations.

Qualification Documents

The qualification process requires specific documents be provided. There are differences in standards regarding the expected documents. However, certain information is shared across all standards, and the documentation package shall include:

- Tool operational requirements
- Declaration of the tool confidence/qualification level (result of the tool classification process)
- Qualification plan that specifies tool use cases and outlines the validation strategy
- Requirements coverage matrix with qualification testing results (results of the tool validation)
- Guidelines regarding tool application in the safety-critical software development process

The above list is not exhaustive. As mentioned, standards have specific naming conventions and expectations regarding documentation, for example, DO-330. The information listed in all but the last bullet focuses on documenting the qualification process. The last item is typically the most challenging and is meant to guide tool users regarding any potential limitations and restrictions for using the tool in a safe way. The ISO 26262 and IEC 61508 standards family requires this information in the form of the Tool Safety Manual (TSM).

In the case of GoogleTest, there are TSM-relevant guidelines that can be extracted from the GoogleTest documentation, for example:

- When comparing a pointer to NULL, use EXPECT_EQ(ptr, nullptr) instead of EXPECT_EQ(ptr, NULL)
- In binary comparison assertions, argument evaluation order is undefined, tests should not depend on argument evaluation order

The GoogleTest documentation contains more similar guidelines, and Parasoft's tool qualification efforts resulted in additional restrictions. Furthermore, all known bugs reported for GoogleTest must be reviewed and analyzed for the potential implications for safety-software testing. If there are known bugs that may affect safety software testing, appropriate entries, either in the form of workarounds or restrictions, must be added to the TSM.

All these guidelines are critical for the end users since not following them may compromise safety-critical software testing. Any organization planning to deploy GoogleTest for safety-critical software development needs to go through this kind of analysis, including periodic reviews of all reported bugs, and develop a TSM.

One of the main challenges related to TSM is enforcing the restrictions and recommendations in those documents. Enforcing TSM guidelines is typically a manual effort imposed on the tool end users. The process is time and resource consuming and may require code reviews. With GoogleTest qualification, Parasoft applied an innovative strategy and provided static analysis rules for many of the TSM recommendations. These rules automate enforcing restrictions and tool use recommendations, significantly reducing costs.

Conclusion

The GoogleTest unit testing framework is probably the best option now for testing applications based on modern C++. Projects utilizing Adaptive AUTOSAR or similar platforms and those based on AI/ML modern C++ libraries select GoogleTest by default. Increasing use of the GoogleTest framework is also an emanation of a strong industry trend to increase open-source software use for safety systems development.

Parsoft's experience shows that the GoogleTest framework is utilized not only for classic unit testing but frequently deployed in broader contexts, for example, for automated integration and API testing at the component and system level.

Deploying GoogleTest for safety-critical software development is possible. However, a significant effort and a skilled team are required to qualify the tool successfully. Organizations deciding to deploy the tool for large teams must also know that the GoogleTest qualification may not be a one-time effort. With advancements in modern C++ and new functionalities added to the subsequent framework releases, development teams may demand adopting the newer framework versions, forcing tool re-qualification. However, the overall gains in software testing efficiency, tool integration effort, ease of testing assets development, and maintenance outweigh the additional compliance-related costs.

TAKE THE NEXT STEP

Watch this demo video to see how your C/C++ development team can unify GoogleTest and Parasoft to meet your full range of testing and verification needs.

About Parasoft