

Transform C/C++ Verification with GoogleTest Automation

Discover practical ways to improve quality, compliance, and confidence while reducing manual effort and leveraging existing tools.

A green test result is reassuring when developing high-quality software. The code compiled, the tests executed, and the assertions passed. For C and C++ development teams, these results provide an important measure of progress and confidence in software behavior.

For teams developing safety-critical and highly regulated software, however, passing tests are only part of the verification challenge. A successful test can demonstrate that a software component behaved as expected under a specific set of conditions, but it doesn't necessarily show how thoroughly the implementation was exercised, whether critical failure scenarios were evaluated, which requirements were verified, or whether sufficient evidence exists to support a safety assessment or compliance audit.

That distinction is becoming increasingly important as embedded software grows more complex. Modern systems incorporate more software-defined functionality, connected technologies, sophisticated user experiences, and AI-enabled capabilities. At the same time, development cycles continue to accelerate, placing greater pressure on engineering teams to deliver increasingly complex software without compromising quality, safety, security, or compliance.

The challenge is no longer simply how to run more tests. It's how to transform testing into continuous, objective evidence that software is ready to move forward.

Why Engineering Teams Choose GoogleTest

There are good reasons engineering teams have embraced [GoogleTest](#). The open-source framework gives C and C++ developers a flexible and familiar way to create tests using the language, tools, and development practices they already know.

GoogleTest supports assertions, test fixtures, parame-

trized testing, and other capabilities needed to develop effective C and C++ tests. [GoogleMock](#) (now part of GoogleTest) extends these capabilities by allowing developers to isolate software components, control dependencies, and evaluate how the software responds under conditions that may be difficult to reproduce with actual hardware.

These capabilities are particularly valuable in embedded development. A software component may depend on a sensor, communications interface, operating system service, or hardware device that's unavailable during development or difficult to manipulate during testing. Mocking allows developers to replace these dependencies with controlled behavior and simulate invalid values, unexpected responses, communication failures, and other abnormal conditions.

This provides a strong foundation for fault injection, robustness testing, and other verification activities that are important when developing critical software.

GoogleTest also fits naturally into modern development environments. Teams can incorporate it into existing build systems, execute tests locally, automate testing through CI/CD pipelines, and adapt their testing workflows to complex C and C++ applications.

As software moves through the development lifecycle, though, the questions engineering teams need to answer become progressively more difficult. This is where test execution alone reaches its limits.

Passing Tests Don't Tell the Whole Story

Consider a development team that executes thousands of GoogleTest tests and achieves a 100% pass rate. Does that mean the software has been adequately verified?

The answer can't be determined from the number of tests or their pass/fail status alone. Thousands of tests could repeatedly exercise the same portions of the implementation



1. Verification activities can be connected across testing, coverage, traceability and compliance to provide a clearer view of software readiness.

while leaving critical decisions, error-handling routines, and failure conditions untouched.

Effective software verification therefore requires multiple forms of evidence. Test results reveal whether the software behaved as expected. Structural code coverage reveals which parts of the implementation were exercised and identifies areas that remain untested. Requirements traceability connects verification activities to the intended behavior of the system. Fault injection and robustness testing help demonstrate how software responds when something goes wrong. Compliance reporting organizes this information into evidence that engineering teams, quality organizations, assessors, and auditors can evaluate.

Each capability answers a different question about software quality and verification. When connected across the development lifecycle, they provide something much more valuable: confidence in the completeness and effectiveness of the verification process.

The Real Challenge is Connecting the Verification Process

Most embedded development organizations don't suffer from a lack of tools. They may use GoogleTest for testing, another technology to collect code coverage, a requirements management system, CI/CD infrastructure for automation, IDEs for development, and additional technologies for reporting and compliance.

The challenge is that the information produced by these tools often remains fragmented. Test results live in one environment, coverage data in another, and requirements in a separate system. Compliance reports may need to be assembled from information collected across multiple tools

and development activities.

As a result, engineers spend valuable time moving between systems, reconciling results, maintaining custom integrations, and manually connecting information that should already be part of the same verification process.

The problem becomes more difficult as projects grow. A collection of scripts and custom integrations may work for a small development team, but they become increasingly difficult to maintain across multiple projects, development environments, target platforms, and CI/CD pipelines. Every integration creates another dependency, every change to the development environment can require updates, and every manual step creates another opportunity for verification evidence to become incomplete, inconsistent, or outdated.

Organizations can automate thousands of tests while the verification process surrounding those tests remains highly manual. Solving this problem requires looking beyond test automation and considering how verification activities connect across the entire software development lifecycle.

From Test Automation to Continuous Verification

A more scalable approach begins by treating testing as part of a broader verification process. Tests shouldn't simply execute and disappear into a log file. The results should contribute to an evolving body of engineering evidence that helps teams understand software quality, verification progress, and release readiness (Fig. 1).

When developers execute GoogleTest tests, the same workflow can collect structural code coverage information. Coverage results may reveal portions of the implementation that require additional attention. Tests could be associated

with requirements so teams are able to determine whether intended system behaviors have been verified. Results from local development, CI/CD pipelines, and target environments can contribute to a broader understanding of verification progress.

Reporting and analytics subsequently transforms this information into actionable insights for developers, engineering managers, quality teams, and compliance stakeholders.

This represents an important evolution from test automation to continuous verification. Test automation focuses primarily on executing tests more efficiently. Continuous verification connects testing with coverage, traceability, reporting, and other engineering activities to provide ongoing evidence about software quality and readiness throughout development.

Preserve the Developer Workflow While Expanding Verification

The most effective verification strategy should not require developers to abandon the frameworks and development environments that already work.

This is particularly important for organizations that have invested heavily in GoogleTest. Existing tests represent years of engineering knowledge. Development teams understand the framework, build systems and CI/CD pipelines already

execute the tests, and engineers have established workflows for creating, reviewing, and maintaining test code.

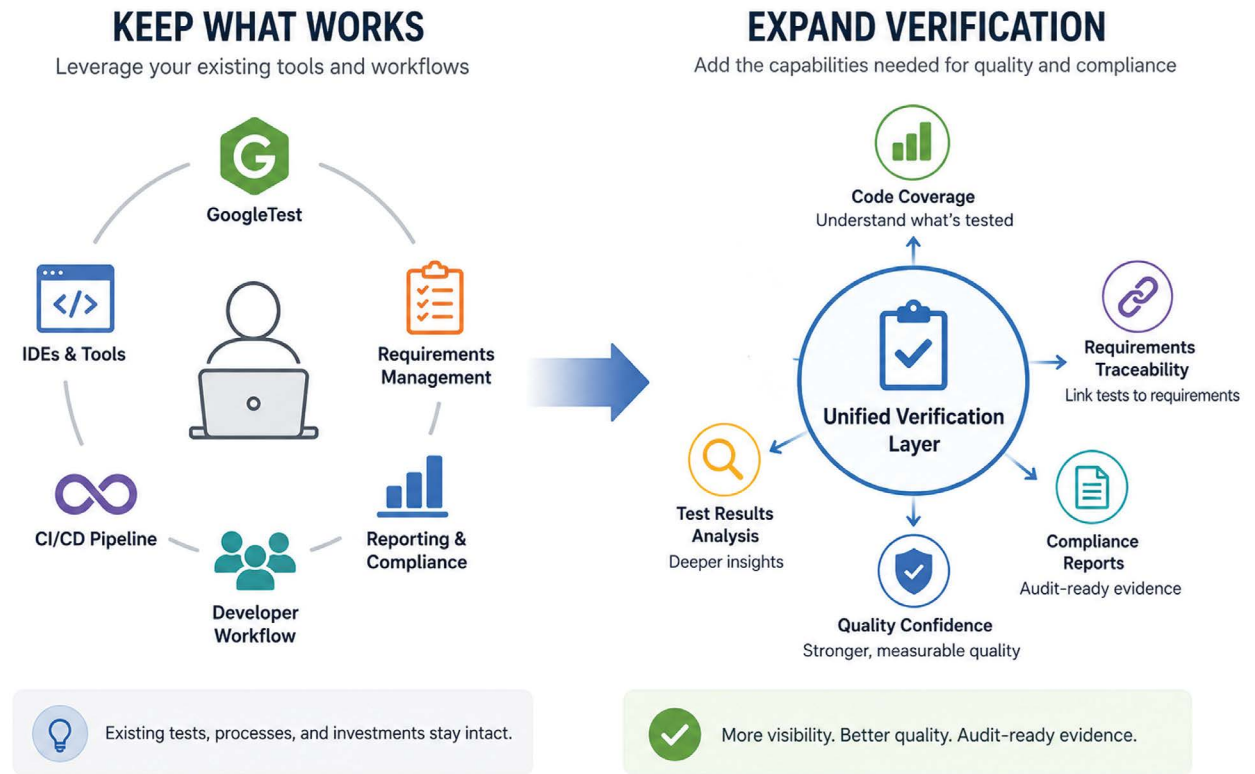
Replacing that infrastructure simply to gain additional verification capabilities can introduce unnecessary cost, disruption, and risk. A more practical approach is to extend existing workflows with the capabilities needed to support broader software quality and compliance objectives (Fig. 2).

[Parasoft C/C++test CT](#) extends existing C and C++ testing workflows by adding capabilities such as structural code coverage, requirements traceability, test results analysis, and compliance-ready reporting. Developers can continue using GoogleTest and other supported testing frameworks while incorporating test execution into a more comprehensive verification process.

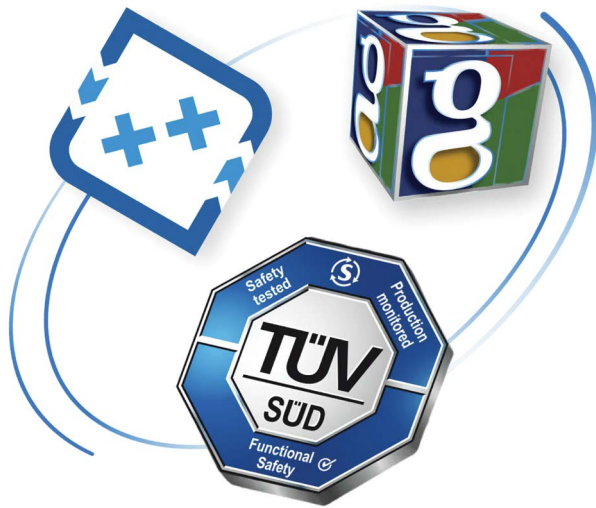
This approach allows organizations to preserve their investments in existing development tools, build environments, CI/CD pipelines, and testing frameworks while gaining greater visibility into what has been verified and what work remains.

Turn Test Execution into Actionable Verification Evidence

When GoogleTest becomes part of a connected verification workflow, a single test execution can provide significantly more information than a pass or fail result.



2. Existing GoogleTest workflows can be extended with coverage, traceability, and reporting capabilities to support broader verification goals.



3. Modern software verification combines automated testing with the evidence needed to support quality, safety and compliance objectives.

In addition to determining whether the software behaved as expected, engineering teams are able to identify which statements, branches, decisions, and conditions were exercised. They can determine whether the execution contributed to project coverage objectives, associate tests with requirements, identify requirements without corresponding tests, and locate portions of the implementation that remain untested.

Coverage information collected from different stages of development could also contribute to a more complete view of verification progress. This is particularly valuable for embedded systems, where testing may occur on development hosts, simulators, virtual environments, target hardware, or hardware-in-the-loop systems.

Instead of treating each environment as an isolated source of test results, teams can combine verification evidence to better understand how thoroughly the software has been exercised across the development lifecycle.

This changes the value of testing. A test no longer exists solely to produce a pass or fail result. It becomes part of a continuously growing body of evidence that helps engineering teams evaluate software quality and make better decisions.

Safety-Critical Development Raises the Stakes

A connected verification process becomes particularly important for organizations developing software subject to functional-safety standards.

Standards such as ISO 26262, IEC 61508, and IEC 62304 establish rigorous expectations for systematic software development and verification. Depending on the applicable standard, criticality level, intended tool use, and project

objectives, engineering teams may need to demonstrate requirements-based testing, structural code coverage, traceability, documented verification activities, and confidence in the tools used during development.

Open-source technologies such as GoogleTest provide significant advantages, including flexibility, transparency, and familiarity. However, their use within safety-critical development could introduce additional considerations. Organizations must understand how testing technologies are being used, what risks may result from tool failures, and what evidence is necessary within the applicable safety process.

Parasoft addresses part of this challenge by providing TÜV SÜD-certified GoogleTest technologies integrated with C/C++test CT. This gives organizations a documented foundation for incorporating the familiar open-source testing framework into verification workflows designed to support functional safety development (*Fig. 3*).

The value extends beyond the certification itself. Engineering teams can continue using familiar GoogleTest workflows while connecting testing with coverage, traceability, reporting, and other capabilities needed to support software verification and compliance.

Certification doesn't eliminate an organization's responsibility to satisfy the requirements of the applicable safety standard or address project-specific tool qualification considerations. But it can reduce the effort required to independently develop and maintain supporting evidence for the testing technologies covered by the certification.

Connected Verification Creates a Stronger Foundation for AI

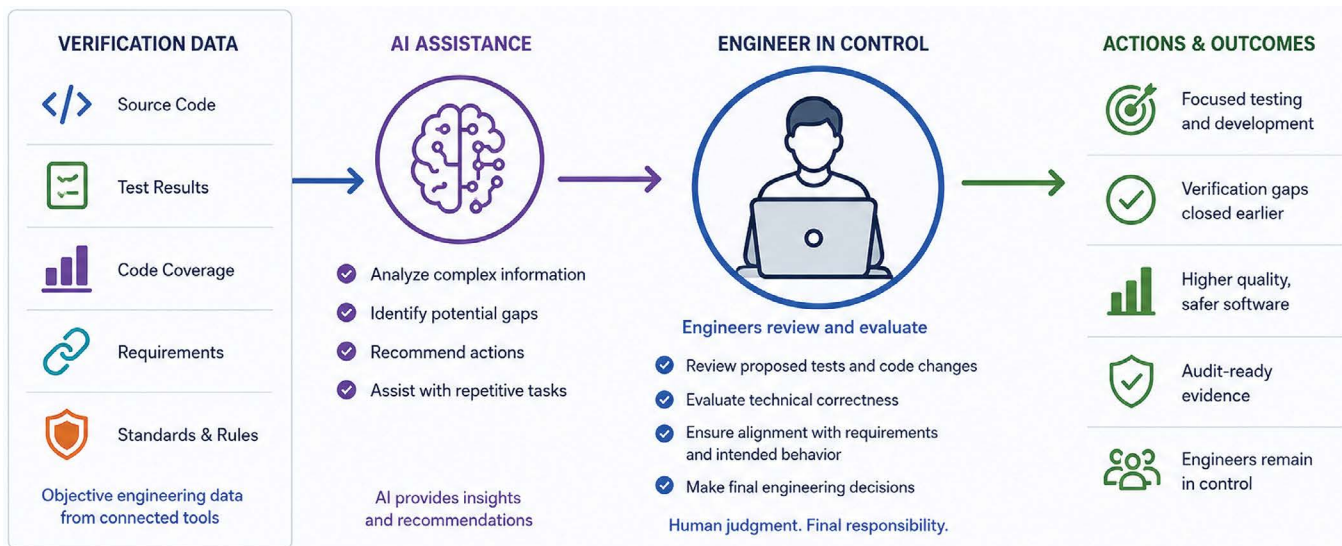
The emergence of generative AI provides another reason to connect development and verification information.

AI can assist developers with generating code, creating tests, analyzing results, and recommending changes. Without sufficient engineering context, however, an AI agent may have little understanding of what the development team actually needs.

Generating additional tests provides limited value if those tests exercise code that's already been thoroughly tested. Recommending a code change may create additional work if the proposed modification violates a required coding standard or fails to address the underlying engineering problem.

AI becomes more useful when it has access to objective information from the verification process. Technologies such as Model Context Protocol (MCP) can provide AI agents with structured information from development and verification tools, giving them greater context about the software and its current verification status (*Fig. 4*).

Instead of simply asking AI to generate a test, engineers can use verification data to focus AI assistance on specific engineering objectives. An AI agent helps analyze coverage



4. AI can help interpret verification data and identify opportunities for improvement while engineers retain final decision-making authority.

gaps, identifies portions of the implementation that may require additional testing, provides information about static-analysis findings, and assists engineers in determining appropriate next steps.

This represents a more practical approach to using AI in critical software development. Rather than generating more code and tests simply because the technology makes it possible, AI helps engineering teams act on objective information produced throughout the verification process.

Keep Engineers in Control of AI-Assisted Verification

For safety- and security-critical software development, the goal of AI should be to accelerate engineering work rather than replace engineering judgment.

AI can help analyze complex information, identify potential verification gaps, recommend actions, and assist with repetitive engineering tasks. Developers remain responsible for reviewing proposed tests and code changes, evaluating their technical correctness, and determining whether they accurately reflect intended software behavior and project requirements.

A connected verification process supports this human-in-the-loop approach by giving both engineers and AI systems access to more meaningful information. Verification technologies provide objective engineering data, AI helps engineers interpret and act on that information, and human experts remain responsible for final engineering decisions.

The result isn't autonomous software verification. It's a more efficient and informed verification process that allows engineering teams to use AI while maintaining the oversight required for critical software development.

The Future of C and C++ Verification is Connected

The embedded software industry doesn't suffer from a shortage of development and testing tools. The greater challenge is turning the information produced by those tools

into a coherent understanding of software quality, safety, compliance, and release readiness.

GoogleTest can remain an important part of that process. Its flexibility, familiarity, and integration with modern C and C++ development workflows make it a valuable testing framework for embedded development.

The greater opportunity is to connect GoogleTest execution with structural code coverage, requirements traceability, compliance reporting, CI/CD automation, and AI-assisted engineering workflows.

By transforming isolated testing activities into a continuous verification process, organizations can identify gaps earlier, make better engineering decisions, reduce manual effort, and build greater confidence in increasingly complex software.

Passing tests will always be important. For teams developing safety-critical and highly regulated software, though, the more important question is whether the verification process can provide the objective evidence needed to demonstrate that the software is ready.

Miroslaw Zielinski has been developing and supporting embedded systems testing frameworks since he graduated from AGH University of Science and Technology in Krakow, where his studies of automation systems and applied robotics gave him key insight into quality challenges in the embedded software industry. He directs the development of Parasoft's flagship testing solution, C/C++test, from gathering user requirements to defining product development strategies to building successful go-to-market plans.

Ricardo Camacho guides the strategy and growth of Parasoft's software test automation solutions for the embedded safety- and security-critical market. With 30+ years of experience in systems and software engineering of real-time systems, Ricardo is a thought leader for standards like ISO 26262, DO-178C, IEC 62304, IEC 61508, IEC 62443, DO-326A, and more.