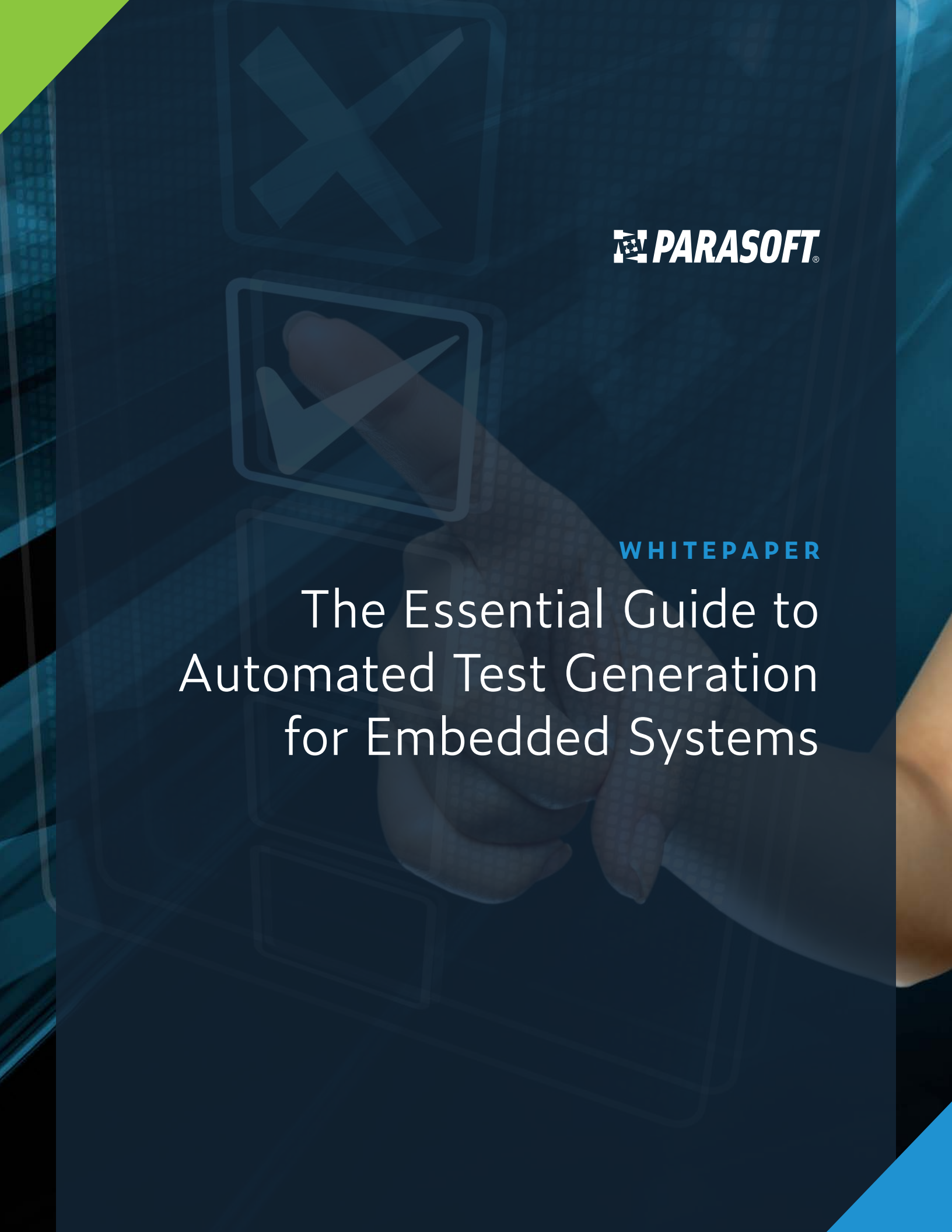
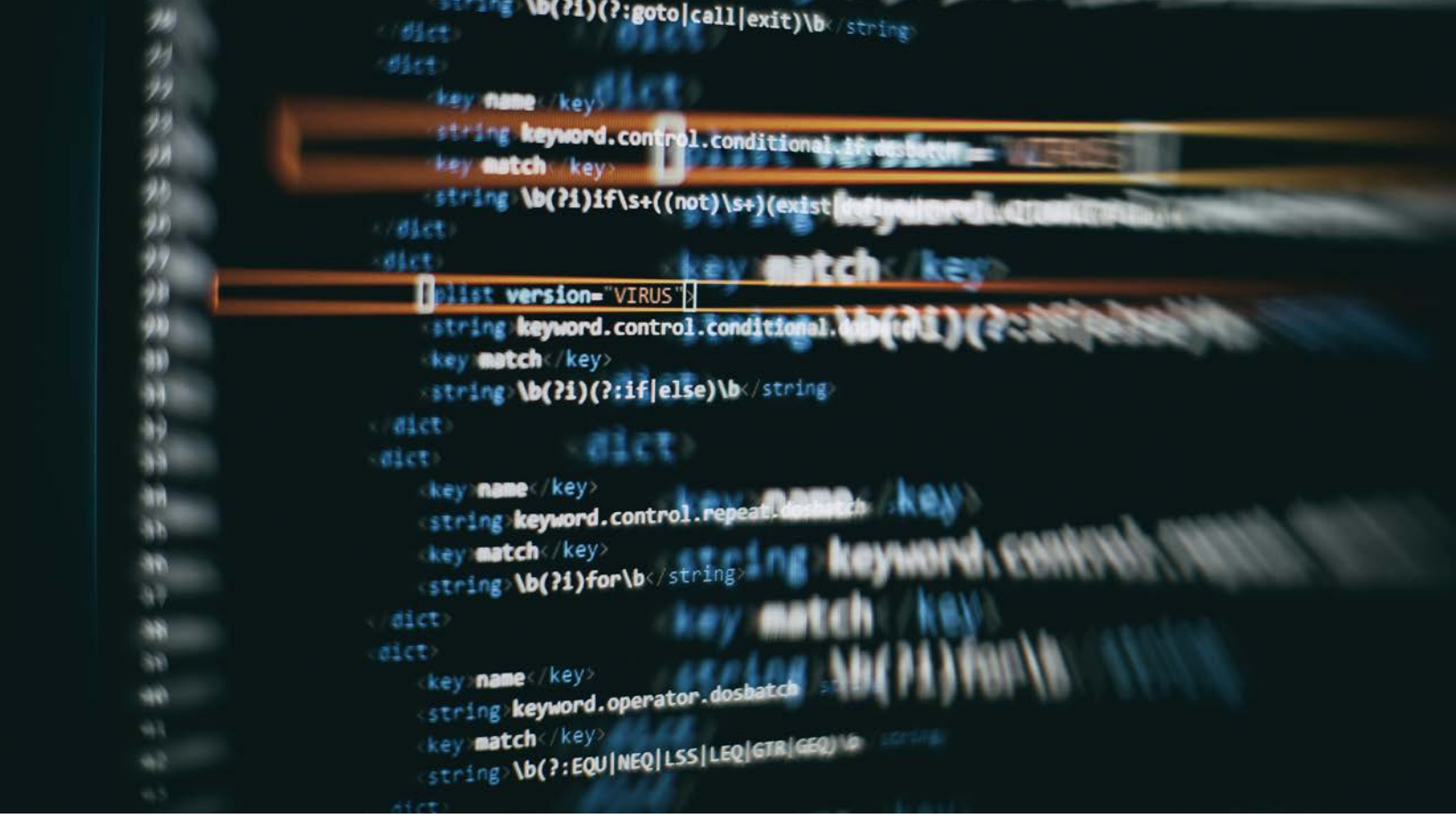


PARASOFT®

WHITEPAPER

# The Essential Guide to Automated Test Generation for Embedded Systems




Most development teams will agree that despite the effort and costs of unit testing, it's essential to embedded software development. Unit testing helps developers truly understand the code they're developing and provides a solid foundation to a verification and validation regimen needed to satisfy safety and security goals for a product. Building on this foundation of unit tests enables teams to accelerate agile development while mitigating risk of defects slipping into later stages of the pipeline.


*Figure 1: Continuous integration pipeline for a solid modern development foundation.*

## WHY AUTOMATED TEST GENERATION?

Typically, development teams do an inadequate amount of unit testing. Alternatively, if they're required to achieve high levels of code coverage, for example, they spend a large amount of money and time to achieve it.

The constraints on the amount of testing are due to multiple factors such as the pressure and time it takes to deliver increased functionality, and the complexity and time-consuming nature of creating valuable unit tests.

Common reasons developers cite that limit the efficiency of unit testing as a core development practice include the following.

- It's difficult to understand, initialize, and/or isolate the dependencies of the unit under test.
- Determining what to validate and defining appropriate assertions is time consuming and often requires intelligent guess work.
- There's a lot of manual coding involved, often even more than was required to implement a specific feature or enhancement.
- It’s just not that interesting. Developers don’t want to feel like testers. They want to spend time delivering more functionality.

Unit test automation tools universally support some sort of test framework, which provides the harness infrastructure to execute units in isolation while satisfying dependencies via stubs. This includes the automated generation of test harnesses and the executable components needed for host and target-based testing.

Test data generation and management, however, is the biggest challenge in unit testing and test generation. Test cases need to cover a gamut of validation roles such as ensuring functional requirements, detecting unpredictable behavior, and assuring security, and safety requirements. All while satisfying test coverage criteria.

Automated test generation decreases the inefficiencies of unit testing by removing the difficulties with initialization, isolation, and managing dependencies. It also removes much of the manual coding required while helping to manage the test data needed to drive verification and validation.

## UNIT TESTING IN EMBEDDED SYSTEMS

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

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

In terms of the classic V model of development, unit test execution is a validation practice to ensure module design is correct. Many safety-specific development standards have guidelines for what needs to be tested for unit testing. For example, ISO 61502 and related standards, have specific guidelines for testing in accordance with safety integrity level where requirements-based testing and interface testing are highly recommended for all levels. Fault injection and resource usage tests are recommended at lower integrity levels and highly recommended at the highest SIL (Safety Integrity Levels) levels. Similarly, the method of driving test cases is also specified with recommended practices.

### TEST CASE DRIVERS

#### Analysis of Requirements

Every requirement drives—at minimum—a single unit test case. Although test automation tools don't generate tests directly from requirements, they must support two-way traceability from requirements to code and requirements to test. And maintain requirements, tests, and code coverage information.

#### Generation & Analysis of Equivalence Classes

Test cases must ensure that units behave in the same manner for a range of inputs not just cherry picked inputs for each unit. Test automation tools must support test case generation using data sources to efficiently use a wide range of input values.

#### Analysis of Boundary Values

Automatically generated test cases, such as heuristic values, boundary values, employ data sources to use a wide range of input values in tests.

#### Error Guessing

This method uses the function stubs mechanism to inject fault conditions into tested code flow analysis results and can be used to write additional tests.

## AUTOMATED TEST EXECUTION

Test automation provides large benefits to embedded software. Moving away from test suites that require a lot of manual intervention means that testing can be done quicker, easier, and more often.

Offloading this manual testing effort frees up time for better test coverage and other safety and quality objectives. An important requirement for automated test suite execution is being able to run these tests on both host and target environments.

### TARGET-BASED TESTING FOR EMBEDDED SYSTEMS

Automating testing for embedded software is more challenging due to the complexity of initiating and observing tests on target hardware. Not to mention the limited access to target hardware that software teams have.

Software test automation is essential to make embedded testing workable on a continuous basis from host development system to target system. Testing embedded software is particularly time consuming. Automating the regression test suite provides considerable time and cost savings. In addition, test results and code coverage data collection from the target system are essential for validation and standards compliance.

Traceability between test cases, test results, source code, and requirements must be recorded and maintained. So, data collection is critical in test execution.

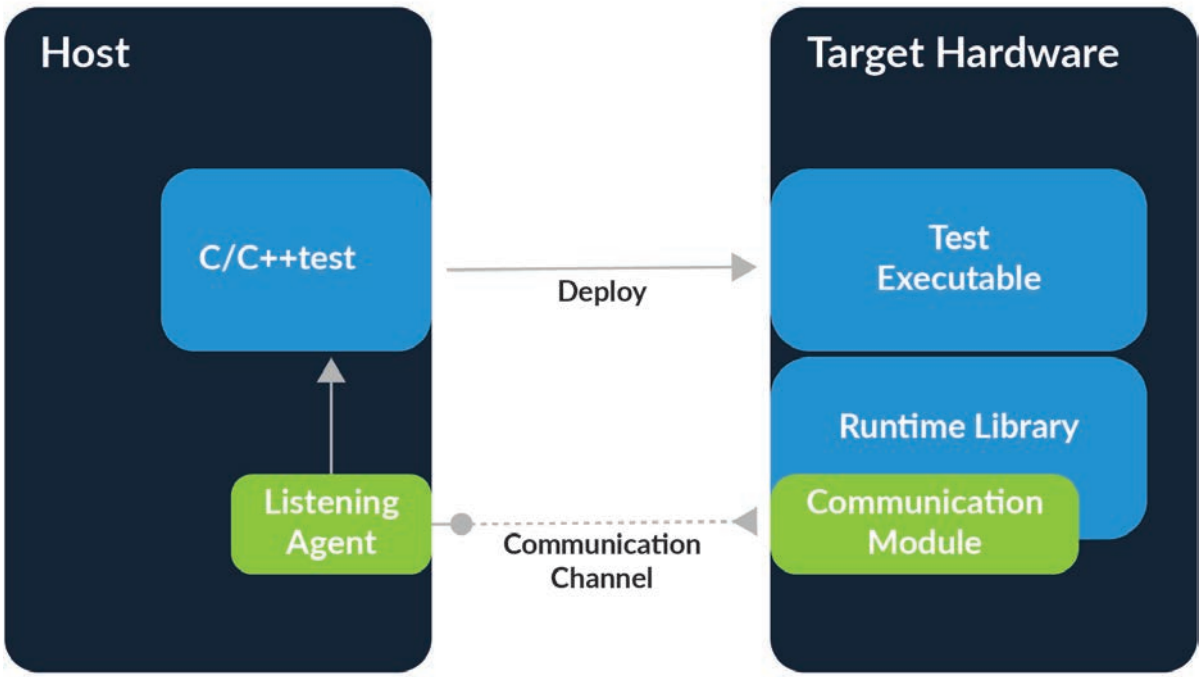



*Figure 2: A high-level view of deploying, executing, and observing tests from host to embedded target.*

## STRUCTURAL CODE COVERAGE

Collecting and analyzing code coverage metrics is an important aspect of safety-critical software development. Code coverage measures the completion of test cases and executed tests. It provides evidence that validation is complete, at least as specified by the software design. It also identifies dead code. This is code that can logically never be reached. It demonstrates the absence of unintended behavior. Code that isn't covered by any test is a liability because its behavior and functionality are unknown.

The amount and extent of code coverage depends on the safety integrity level. The higher the integrity level, the higher the rigor used, and inevitably the number and complexity of test cases. Regardless of the level of coverage required, automated test case generation can increase test coverage over time.

Advanced unit test automation tools should measure these code coverage metrics. In addition, it's necessary that this data collection works on host and target testing and accumulates test coverage history over time. This code coverage history can span unit, integration, and system testing to ensure coverage is complete and traceable at all levels of testing.

## TYPES OF AUTOMATED TEST CASE GENERATION

For practical purposes, automated tools should generate test cases in existing well-known formats like CppUnit. By default, one test suite per source/header file makes sense, but tools should support one test suite per function or one test suite per source file if needed.

Another important consideration is the automatic stub definitions to replace "dangerous" functions, which includes system I/O routines such as rmdir(), remove(), rename(), and so on. In addition, stubs can be automatically generated for missing function and variable definitions. User-defined stubs can be added as needed.

### REQUIREMENTS-BASED TEST CASE GENERATION

Although test automation tools can't derive requirements tests from documentation, they can help make the creation of test cases, stubs, and mocks easier and more efficient. In addition, automation greatly improves test case data management and tool support for parameterized tests also reduces manual effort.

Particularly important is traceability from requirements to code to tests and test results. Manually managing traceability is nearly impossible and automation makes two traceability a reality.

While requirements are being decomposed, traceability must be maintained throughout the phases of development as customer requirements decompose into system, high-level, and low-level requirements. The coding or implementation phase realizes the low-level requirements. Consider the typical V diagram of software.

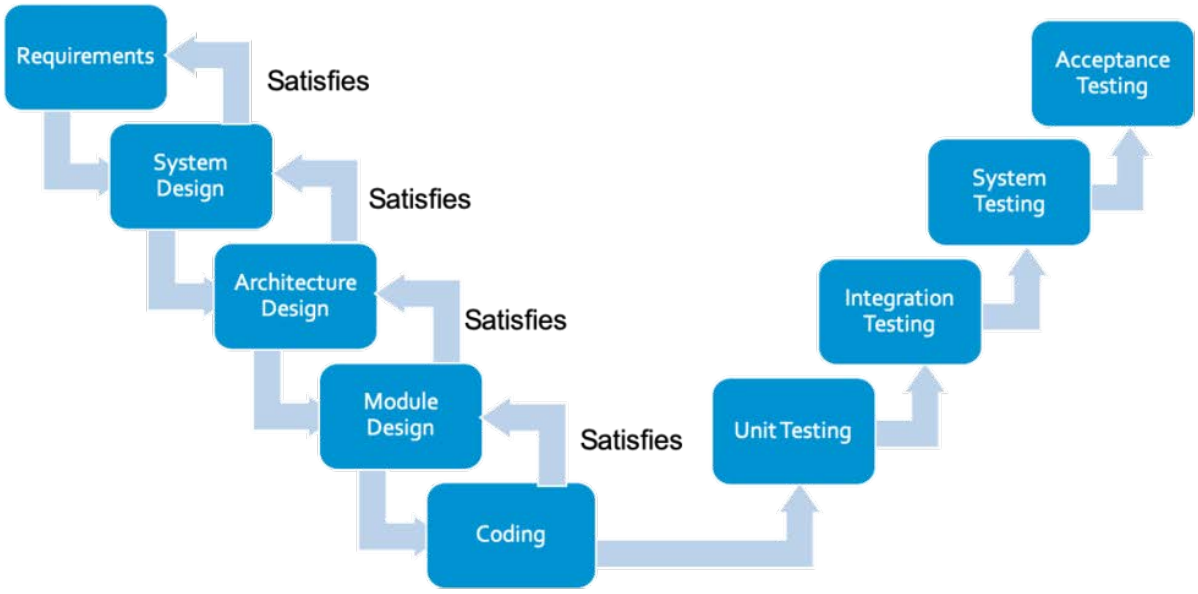



*Figure 3:*
*The V-model of system development with traceability overlay.*

Each phase drives the subsequent phase. In turn, the work items or refined requirements in each phase must satisfy the requirements from the previous phase. Architectural requirements that have been created or decomposed from system design must satisfy the system design/requirements, and so on.

Traceability proves that each phase is satisfying the requirements of each subsequent phase. Developers write code that implements or realizes each requirement and for safety-critical applications, links for traceability to test cases and down to the code are established. Therefore, if a customer requirement changes or is removed, the team knows what it impacts down the line, all the way to the code and tests that validate the requirements.

Industry standards like DO-178B/C, ISO 26262, IEC 62304, IEC 61508, EN 50128, and others require the construction of a traceability matrix for identification of any gaps in the design and verification of requirements. This helps achieve the ultimate goal of building the right product. More than that, it's to ensure the product has the quality, safety, and security to ensure it remains the right product.

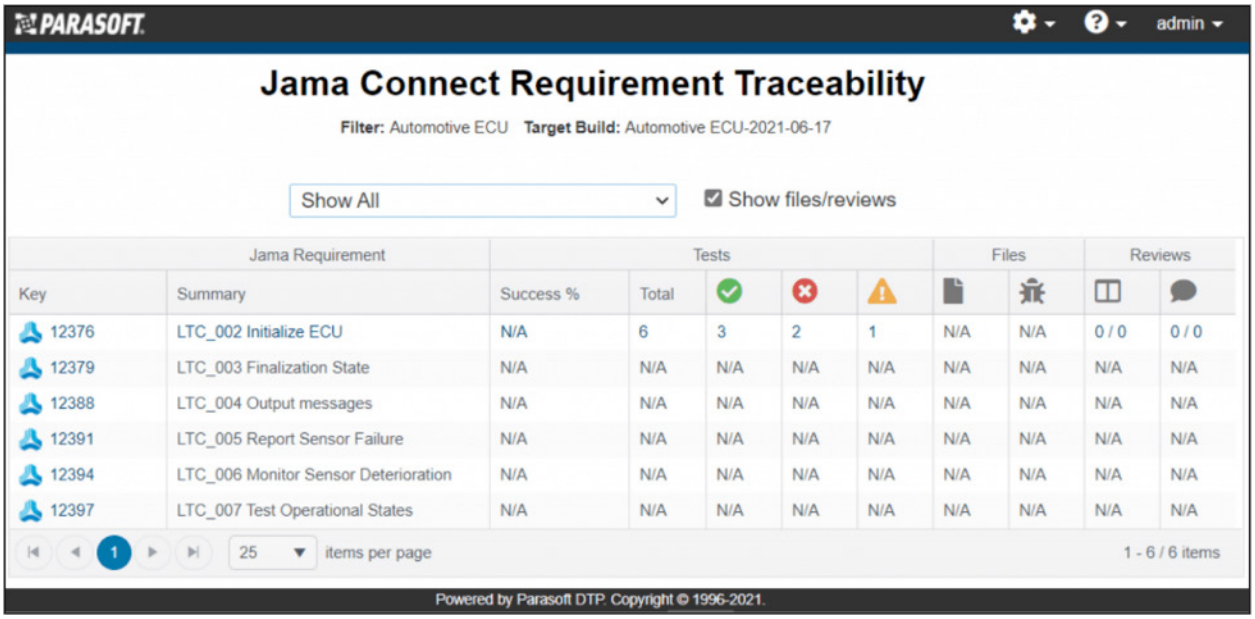



*Figure 4: Parasoft traceability matrix of Jama requirements to tests and code.*

## CODE COVERAGE-BASED TEST CASE GENERATION

The creation of productive unit tests has always been a challenge. Functional safety standards compliance demands high-quality software, which drives a need for test suites that affect and produce high code coverage statistics. Teams require unit test cases that help them achieve 100% code coverage. This is easier said than done. Analyzing branches in the code and trying to find reasons why certain code sections are not covered continues to steal cycles from development teams.

Unit test automation tools can be used to fill in the coverage gaps in test suites. For example, advanced static code analysis (data and control flow analysis) is used to find values for the input parameters required to execute specific lines of uncovered code.

It's also valuable if you have automated tools that not only measure code coverage but also keep track of how much modified code is being covered by tests, because this can provide visibility into whether enough tests are being written along with changes in production code. See the following example code coverage report.

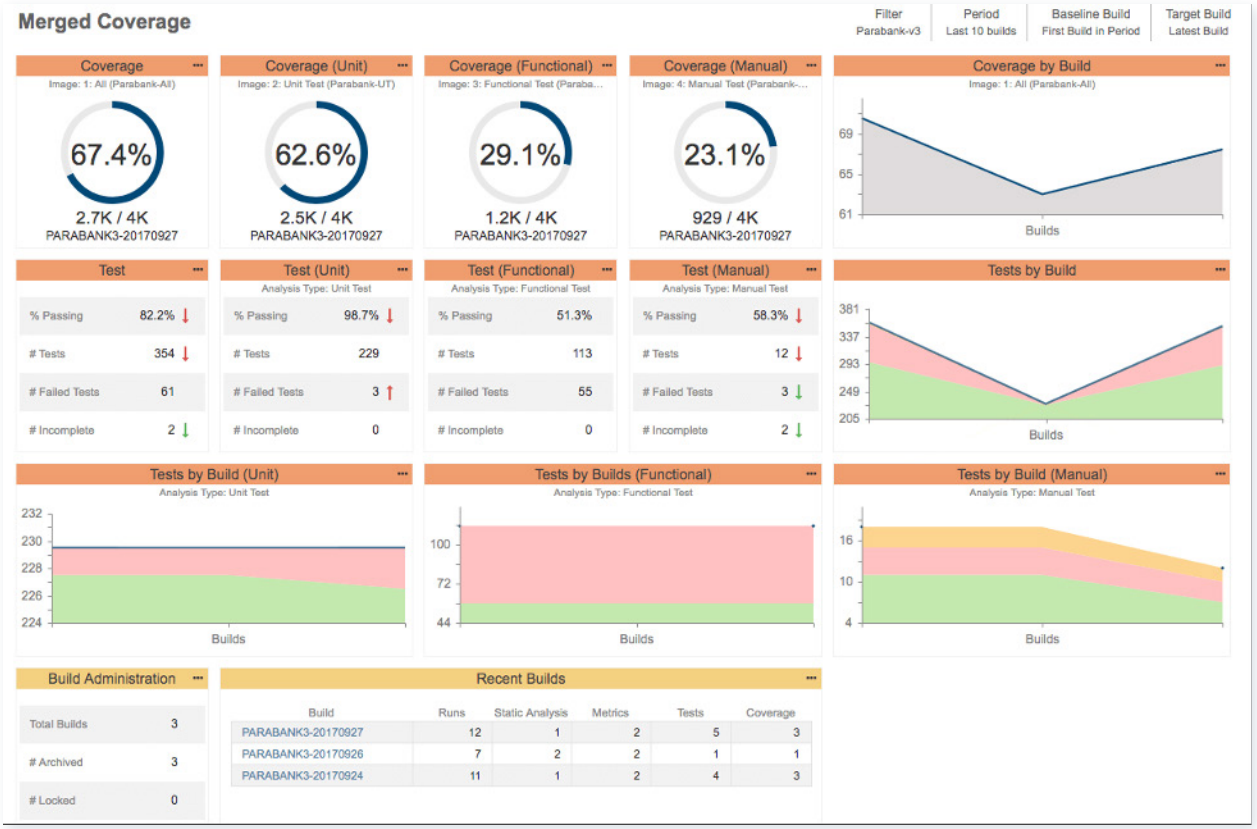



*Figure 5: Aggregation of code coverage from various testing methods in Parasoft DTP.*

## Using Static Analysis to Drive Coverage-Based Test Cases

In complex code, there are always those elusive code statements of which it's exceedingly difficult to obtain coverage. It's likely there are multiple input values with various permutations and possible paths that make it mind twisting and time consuming to decipher. But only one combination can get you the coverage you need. Combining test automation and static analysis makes it easy to obtain coverage of those difficult to reach lines of code. An example of test preconditions calculated with static analysis is shown in the Coverage Advisor.

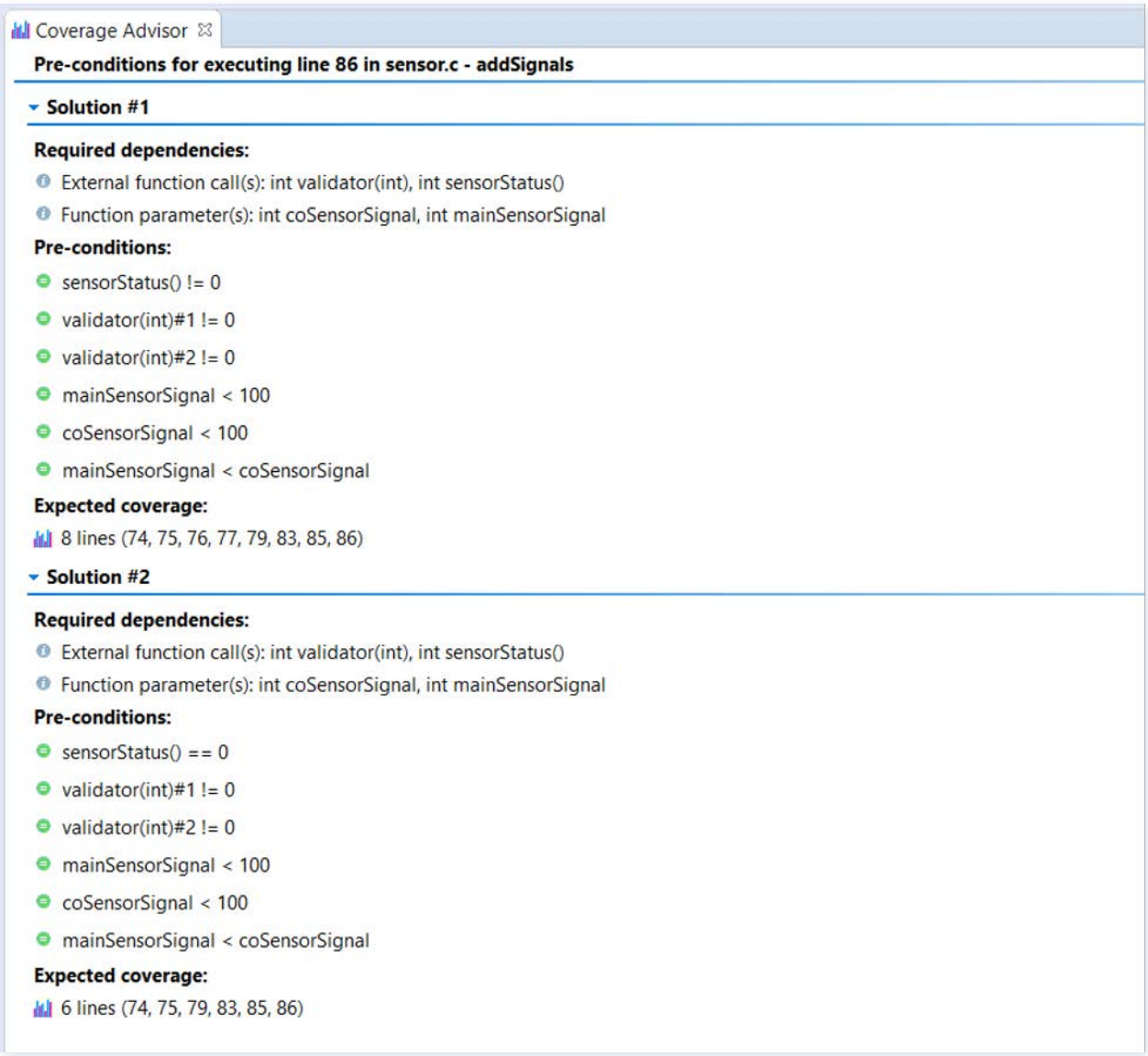



*Figure 6: Code coverage analysis feedback from Parasoft C/C++test.*

### *Defect Test Case Generation*

Another class of test are those created to induce an error condition in the unit under test. The input parameters in these cases are often out of bounds and are just at the boundary conditions for data types, such as using the highest 32-bit positive and negative integers for test data. Other examples are fuzz testing where these boundary conditions are mixed with random data designed to create an error condition or trigger a security vulnerability.

These test cases validate nonfunctional requirements since they fall outside the scope of product requirements, but are essential for determining performance, security, safety, reliability, and other product qualities. Automation is essential since these tests can be numerous (fuzz testing) and rely on repeated execution (performance testing) to help discover quality issues. Test case generation helps reduce the manual effort needed to create these test suites.

# REGRESSION TESTING

As part of most software development processes, regression testing is done after changes are made to software. These tests determine if the new changes had an impact on the existing operation of the software. Managing and executing regression tests are a large part of the effort and cost in testing. Even with automated test generation, test execution, gathering results, and re-running tests is very time consuming. Regression testing encompasses test case maintenance, code coverage improvements and traceability.

Regression tests are necessary, but they only indicate that recent code changes have not caused tests to fail. There's no assurance that these changes will work. In addition, the nature of the changes that motivate the need to do regression testing can go beyond the current application and include changes in hardware, operating system, and operating environment.

In fact, all previously created test cases may need to be executed to ensure that no regressions exist and that a new dependable software version release is constructed. This is critical because each new software system or subsystem release is built or developed upon. If you don't have a solid foundation the whole thing can collapse.

To prevent this, it's important to create regression testing baselines that are an organized collection of tests and will automatically verify all outcomes. These tests are run automatically on a regular basis to verify whether code modifications change or break the functionality captured in the regression tests. If any changes are introduced, these test cases will fail to alert the team to the problem. During subsequent tests, Parasoft C++test will report tasks if it detects changes to the behavior captured in the initial test.

## HOW TO DECIDE WHAT TO TEST

The key challenge with regression testing is determining what parts of an application to test. It's common to default to executing all regression tests when there's doubt on what impacts recent code changes have had—the all or nothing approach.

For large software projects, this becomes a huge undertaking and drags down the productivity of the team. This inability to focus testing hinders much of the benefits of iterative and continuous processes, potentially exacerbated in embedded software where test targets are a limited resource.

A couple of tasks are required here.

1. Identify which tests need to be re-executed.
2. Focus the testing efforts (unit testing, automated functional testing, and manual testing) on validating the features and related code impacted by the most recent changes.

### Test Impact Analysis

Test Impact Analysis (TIA) uses data collected during test runs and changes in code between builds to determine which files have changed and which specific tests touched those files. Parasoft's analysis engine can analyze the delta between two builds and identify the subset of regression tests that need to be executed. It also understands the dependencies on the units modified to determine what ripple effect the changes have made on other units.

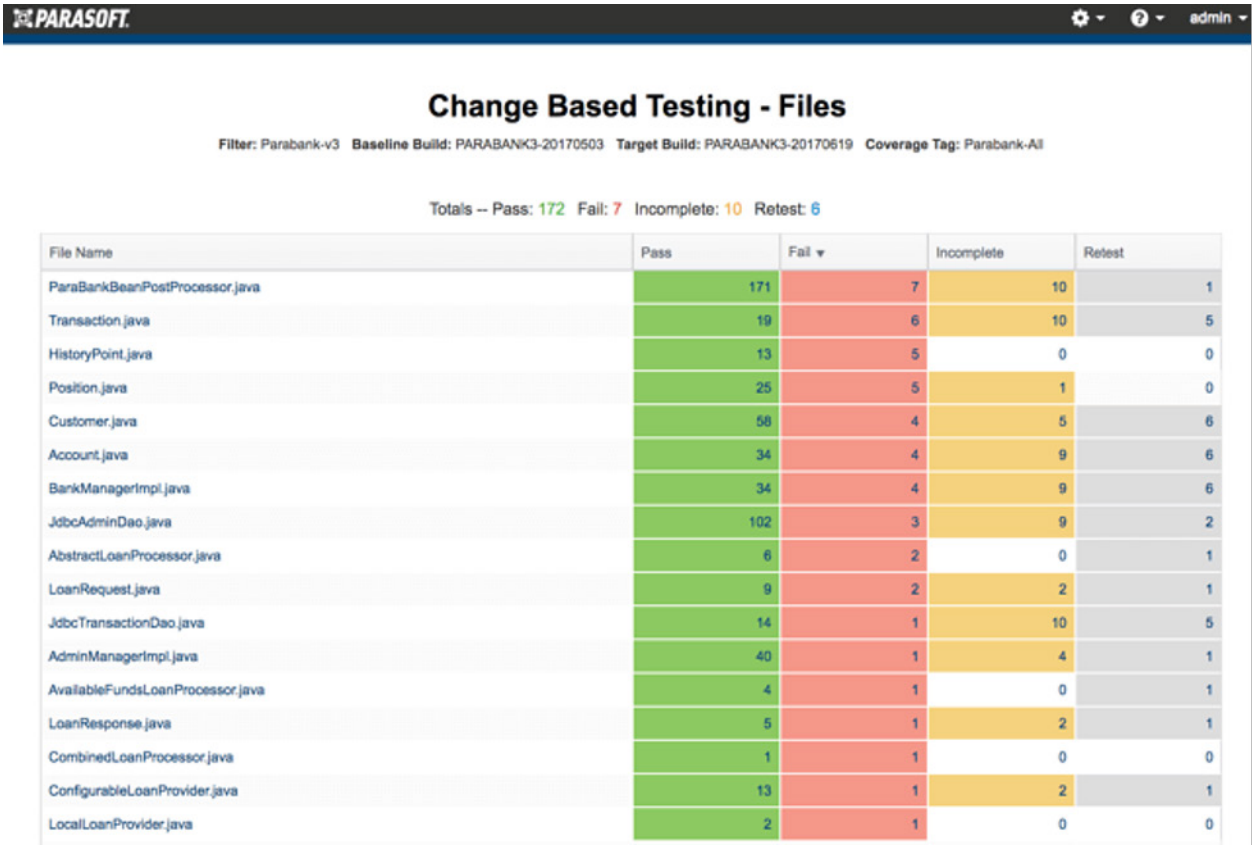
#### *Focus on the Risk*

Due to the complexity of today's codebases, every code change, however innocuous, can subtly impact application stability and ultimately "break the system." These unintended consequences are impossible to discover through manual inspection, so testing is critical

to mitigate the risk they represent. Unless it's understood what needs to be re-rested, efficient testing practice can't be achieved. If there is too much testing in each sprint or iteration, the efficiency brought by test automation is reduced. Testing too little is not an option.

The best approach is to identify which tests need to be re-executed and focus the testing efforts (unit testing, automated functional testing, and manual testing) on validating the features and related code that are impacted by the most recent changes. This is discovered with TIA and planning testing based on a data-driven approach called change-based testing.

TIA needs a repository of already-completed tests that are already executed against each build, either as part of a fully automated test process (such as a CI-driven build step) or while testing the new functionality. This analysis provides insight into where in the code the changes occurred, how the existing tests correlate to those changes, and where testing resources need to focus. Following is an example of a TIA.




| File Name | Pass | Fail | Incomplete | Retest |
|---|---|---|---|---|
| ParaBankBeanPostProcessor.java | 171 | 7 | 10 | 1 |
| Transaction.java | 19 | 6 | 10 | 5 |
| HistoryPoint.java | 13 | 5 | 0 | 0 |
| Position.java | 25 | 5 | 1 | 0 |
| Customer.java | 58 | 4 | 5 | 6 |
| Account.java | 34 | 4 | 9 | 6 |
| BankManagerImpl.java | 34 | 4 | 9 | 6 |
| JdbcAdminDao.java | 102 | 3 | 9 | 2 |
| AbstractLoanProcessor.java | 6 | 2 | 0 | 1 |
| LoanRequest.java | 9 | 2 | 2 | 1 |
| JdbcTransactionDao.java | 14 | 1 | 10 | 5 |
| AdminManagerImpl.java | 40 | 1 | 4 | 1 |
| AvailableFundsLoanProcessor.java | 4 | 1 | 0 | 1 |
| LoanResponse.java | 5 | 1 | 2 | 1 |
| CombinedLoanProcessor.java | 1 | 1 | 0 | 0 |
| ConfigurableLoanProvider.java | 13 | 1 | 2 | 1 |
| LocalLoanProvider.java | 2 | 1 | 0 | 0 |

*Figure 7: Parasoft DTP report on Test Impact Analysis.*

From here, the regression test plan is augmented to address failed and incomplete test cases with the highest priority and using the re-test recommendations to focus scheduling of additional automated runs and prioritizing manual testing efforts.

Testing is a major bottleneck for embedded software development with too many defects being identified at the end of the release cycle due to not enough or misdirected testing. To yield the best results, focus testing efforts on the impact of the changes being made to unlock the efficiency that test automation delivers.

# BENEFITS OF AUTOMATED TEST GENERATION

Automated test case generation removes the human effort, errors, and tedium from unit testing and benefits embedded software development in several ways.

- Reducing labor costs.
- Shortening time to market.
- Satisfying compliance to standards.
- Increasing quality, security, and safety.

## REDUCING LABOR COSTS

Unit test automation by itself is a productivity booster for embedded software development because manual testing on target hardware is time consuming. It's also difficult to ascertain code coverage and requirements traceability. Automated test case generation further increases developer and test productivity and eliminates manual effort of creating and maintaining unit test.

In conjunction with smart test execution, there's a high ROI for the automation investment. In addition, as teams and products mature, these benefits grow over time as the foundation of test assets grows, team expertise increases and product quality, security, and safety improve.

## SHORTENING TIME TO MARKET

The productivity improvements from test automation save money and decrease the time for a software product to converge on the final shipping product. Increased coverage, more frequent and thorough testing, and completed requirements traceability arrive sooner than more ad hoc techniques. Time to market further improves when combining these practices with Agile development, CI/CD, and DevSecOps pipelines. Moreover, customers report higher quality products and discover fewer bugs.

## SATISFYING COMPLIANCE TO STANDARDS

Compliance to industry standards for safety and security requires use of automation to be feasible. Whether it's coding, development, or testing, automation is required for documenting the process, capturing traceability, and proving adequate verification and validation.

For safety-critical devices, validation is crucial as the burden is on the developer to prove they've met more than just product requirements but also ensured the level of safety and security expected by the industry. Unit test generation is a valuable tool for increasing coverage, expanding testing to discover possible error conditions, and fulfilling requirements validation.

## INCREASING QUALITY, SECURITY, & SAFETY

Improved code coverage and better testing for security, performance, and reliability are all achieved with more test cases aided by automatic test case generation. This improved test regimen is made possible through automation with higher productivity and more testing in the same development schedule. The end result is improved safety, security, and quality. Software organizations generally mature their process over time. These improvements span more than just one product and the benefits continue throughout product life cycles, which are significantly longer for embedded devices.

# HOW A MEDICAL DEVICES LEADER USES AUTOMATED TEST GENERATION FOR SAFE, HIGH-QUALITY DELIVERY

Smiths Medical is a leading global manufacturer of specialty medical devices that provides innovative and lifesaving solutions for the world's healthcare markets. The company specializes in infusion therapy, vascular access, and vital care. Its products are found in hospital, emergency, home, and specialty care environments and are used during critical and intensive care, surgery, post-operative care, and for support in managing chronic illness.

Delivering safe, high-quality software for their medical devices is imperative. For that reason, Smiths Medical builds its safety-critical medical devices with a rigorous engineering process where software testing plays a critical role for verification and validation.

Developing medical device software is difficult due the safety and security requirements. Software verification and validation plays an important role in proving the intended functionality has been implemented and safety and security have been incorporated into the products.

Test automation is an important foundation of Smiths Medical's testing approach. Previous attempts at adopting tools weren't fully successful. The development team was looking for a solution to support their entire testing effort with a new approach and mindset of test-driven development (TDD).

## THE SOLUTION: EVOLVING TO TEST DRIVEN DEVELOPMENT

Smiths Medical recognized that they needed to go further than just adopting unit test automation. Their plan was to move the team to test-driven development where design/refactoring and testing are tightly interwoven and rely heavily on automated test generation. Tests are written as a description of the expected unit functionality and code is written and factored to make sure tests pass.

Although the move to TDD can incur some upfront costs, there are significant benefits downstream in terms of lower defect rates, including:

- A fast feedback loop for developers.
- Less time spent debugging.
- Building solid code with clean interfaces.

An important part of making the move to TDD was test automation and tools that support this process. Test automation, including test generation made tests more valuable in terms of their relationship and traceability to requirements, code coverage, work items, builds, and other artifacts.

The benefits of moving to automated test generation meant reduced test maintenance costs and lower costs for medical device pre-market approval. Flexible support for target and host-based testing with comprehensive code coverage was essential for their product development.

## THE RESULTS: INCREASED CODE COVERAGE, BETTER TEST STABILITY, DECREASED TEST FAILURES

Smiths Medical has evolved their testing to test-driven development and seen numerous positive results from their adoption of automated test generation for their safety-critical software development, including:

**Improved test stability.** Unit tests are code. Just like any code, they're prone to mistakes and bugs and require maintenance. Smiths Medical was struggling with test failures that required too much debugging time to figure out if the unit under test was broken or if it was the test itself.

Once they moved to TDD and automation with Parasoft C/C++test, their test stability increased dramatically. Test maintenance was easier and test failures decreased in general.

**Better code coverage and decreased code complexity.** Increasing code coverage was critical for Smiths Medical due to the safety aspect of their products. They needed to show due diligence in testing their software and demonstrating appropriate code coverage is part of that. To this end, they used Parasoft C/C++test to instrument the code and capture their code coverage, and Parasoft DTP to track the code coverage and code complexity metrics. In both cases, the trends have been improving over time. Code coverage is now over 70%. Code complexity decreased below 15 based on McCabe's cyclomatic complexity measurements. In fact, it was now easier than ever to increase coverage because of automated test generation, execution, and results collection.

**Open to closed defect ratio trending to zero.** Smiths Medical observed that the number of tests was increasing due to efforts in obtaining better code coverage, which was directly attributed to their new processes and automation. However, instead of test failures going up in tandem with the increased tests, they were dropping. Also, the ratio of open to closed defects was trending towards zero. This meant that test case quality was improving in terms of clarity and properly set expected test results. There were more tests and more tests passed. There was also a reduction in manual work needed to fix defects or the tests themselves.

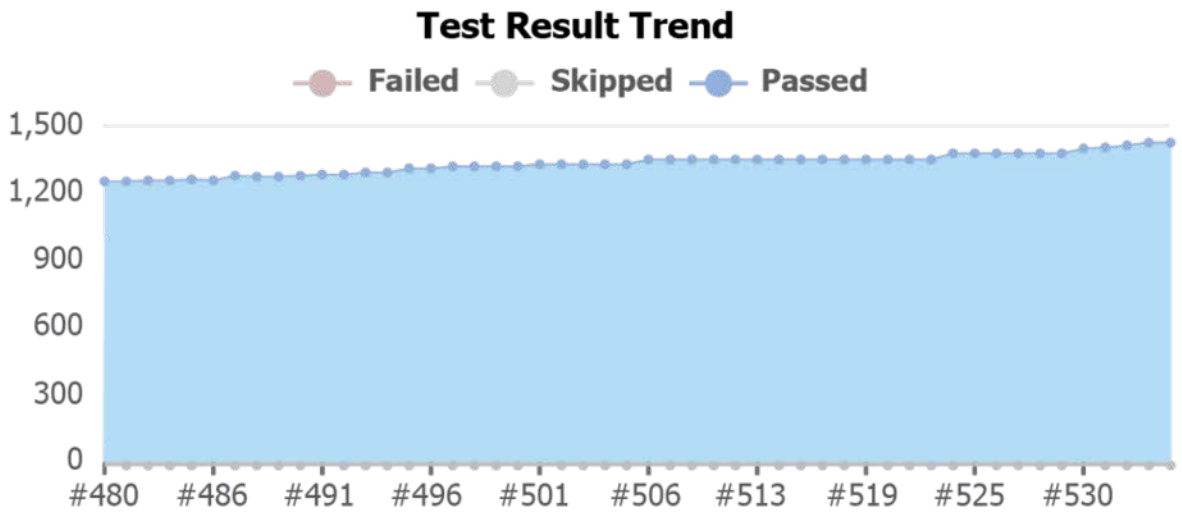



*Figure 8: Smiths Medical test results trend over time. Total tests are increasing but the ratio of failures is decreasing.*

## SUMMARY

Testing is essential to embedded software development. It fosters true understanding of the code being developed and provides a solid foundation to a verification and validation regimen needed to satisfy safety and security goals for a product.

The constraints on testing productivity are due to multiple factors such as the pressure and time it takes to deliver increased functionality, and the complexity and time-consuming nature of creating valuable tests.

Test data generation and management is by far the biggest challenge in unit testing and test generation. Test cases are particularly important in safety-critical software development because they must ensure functional requirements and test for unpredictable behavior, security, and safety requirements. All while satisfying test coverage criteria.

Automated test generation decreases the inefficiencies of unit testing by removing the difficulties with initialization and isolation and managing dependencies. It also removes much of the manual coding required while helping to manage the test data needed to drive verification and validation. This improves quality, safety, and security. It also reduces test time, costs, and time to market.

## TAKE THE NEXT STEP

Learn more about automating unit test generation for your embedded software development team. Talk to one of our experts today.

### ABOUT PARASOFT

Parasoft helps organizations continuously deliver quality software with its market-proven, integrated suite of automated software testing tools. Supporting the embedded, enterprise, and IoT markets, Parasoft's technologies reduce the time, effort, and cost of delivering secure, reliable, and compliant software by integrating everything from deep code analysis and unit testing to web UI and API testing, plus service virtualization and complete code coverage, into the delivery pipeline. Bringing all this together, Parasoft's award winning reporting and analytics dashboard delivers a centralized view of quality enabling organizations to deliver with confidence and succeed in today's most strategic ecosystems and development initiatives — security, safety-critical, Agile, DevOps, and continuous testing.