WHITEPAPER

Continuous Integration & Continuous Delivery for Embedded Systems

Continuous integration and continuous delivery (CI/CD) combine to form the software development practice of frequent integration (merging smaller build units into apps, services, libraries, or components) and continuous delivery or deployment. The intention is to automate testing software for early detection of build and integration issues and errors.

These integration and delivery steps are done frequently, often multiple times a day. To be frequent, automation is a key factor in continuous development practices.

> "Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily — leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.
>
> "You achieve continuous delivery by continuously integrating the software done by the development team, building executables, and running automated tests on those executables to detect problems. Furthermore you push the executables into increasingly production-like environments to ensure the software will work in production."
>
> —Martin Fowler

AUTOMATION IN CONTINUOUS INTEGRATION AND DELIVERY

Without automation of the build process and, by extension, processes connected to creating deployment, artifacts and build verification would be tedious and time consuming—the antithesis of continuous.

Continuous integration relies on a single source repository and automated software build tools. It's important that integration issues and broken builds are sorted out immediately. Following that, the most critical step is to build a deployable product and test the integrated product.

This is the point where most CI/CD and iterative approaches grind to a halt. Testing takes time and effort. It's difficult to decide what to test.

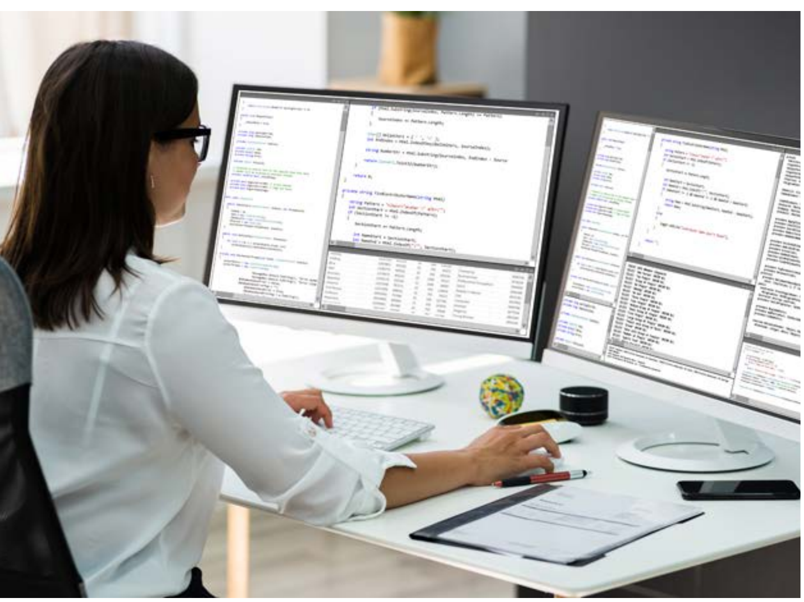
Manual and UI tests can exacerbate this problem. Test automation is important, but more is needed than just running tests. Further discussion of this follows in the paper.

Figure 1: Continuous integration as part of a continuous development cycle.

CONTINUOUS INTEGRATION AND DELIVERY FOR EMBEDDED SYSTEMS

CI/CD is becoming increasingly popular in embedded software development. However projects are often constrained in ways that application development is not.

Besides the physical and computational constraints of the target hardware platform, there are constraints in the marketplace. The marketplace for embedded software has unique requirements for safety, security, privacy, and extremely long life cycles. Products can remain in the market for decades.



At the development level, embedded software isn't much different than typical application development, requiring IDEs, compilers, static, and dynamic analysis and build tools. However, tools often target different architectures than they work on (host versus target environment). Versions of tools are important to ensure homogenous development environment across the team.

Automation at the build level uses the same techniques. But when code needs to be executed, the host/target barrier becomes significant. Automation that requires code execution needs special support in embedded software development.

Automating testing for embedded software is more challenging due to the complexity of initiating and observing tests on embedded targets, 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 the host development system to the target system.

Testing embedded software is particularly time consuming. Automating the regression test suite provides significant time and cost savings. In addition, test results and code coverage data collection from the target system are essential for validation and standards compliance, as needed.

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

Parasoft C/C++test is offered with its test harness optimized to take minimal additional overhead for the binary footprint and provides it in the form of source code, where it can be customized if platform-specific modifications are required.

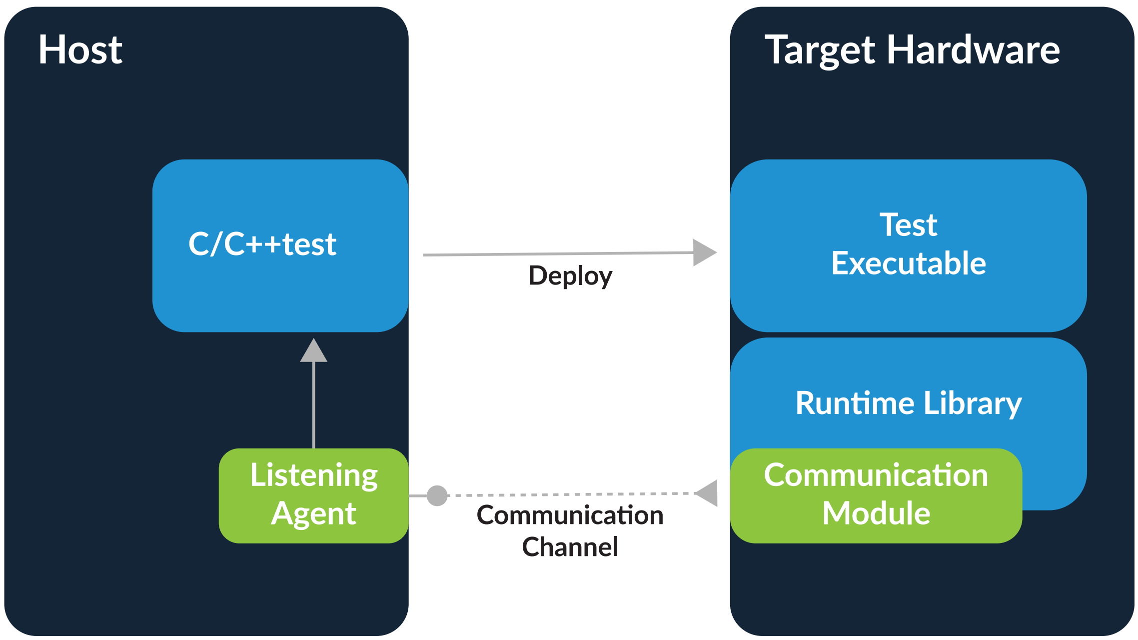


Figure 2: A high level view of deploying, executing, and observing tests from host to target.

One huge benefit that the Parasoft C/C++test solution offers, is its dedicated integrations with embedded IDEs and debuggers that make the process of executing test cases smooth and automated. Supported IDE environments include Eclipse, VS Code, Green Hills Multi, Wind River Workbench, IAR EW, ARM MDK, ARM DS-5, TI CCS, Visual Studio, and many others. See all technical specifications.

The Parasoft solution supports the creation of regression testing baselines as an organized collection of tests and will automatically verify all outcomes. These tests 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, C++test will report tasks if it detects changes to the behavior captured in the initial test.

The parity of capabilities of remote target execution with host-based testing means that embedded software teams can reap the same benefits of automation as any other type of application development.

CONTAINERIZED DEVELOPMENT PLATFORM AT EVERY DEVELOPER'S DESKTOP

Containerized deployments of development tools are becoming the bread and butter of embedded development teams. Even though containers were initially developed to solve problems with the deployment of microservices and web-based applications, they recently gained popularity among embedded teams—especially big teams that use containers to manage complex toolchains.

When it comes to managing complex development environments, specifically in safety-critical software development, teams usually struggle with the following challenges:

- Synchronizing upgrades for the entire team to a new version of a tool like a compiler, build toolchain, and so on.
- Dynamically reacting to a new security patch for the library or software development kit (SDK), and the like.
- Assuring consistency of the toolchain for all team members and the automated infrastructure (CI/CD).
- Ability to version the development environment and restoring it to service the older version of the product that was certified with the specific toolchain.
- Onboarding and setting up new developers.

All these problems are easy to solve with containers.

USING A COMMAND-LINE BASED TOOL WITH A CONTAINERIZED COMPILATION TOOLCHAIN

It's easy to configure the command-line based tool, Parasoft C/C++test Professional, to work with a compilation toolchain and the execution environment deployed in containers. The tool supports deployments that are based on Linux and Docker containers.

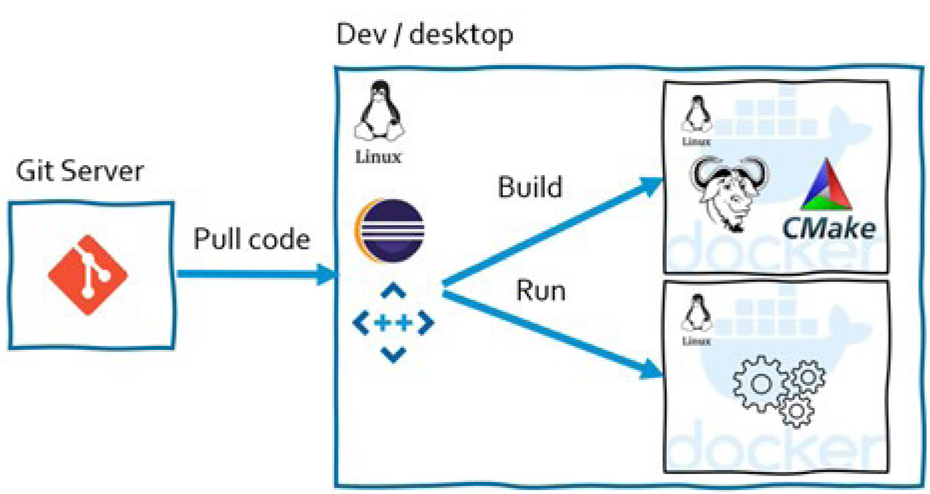
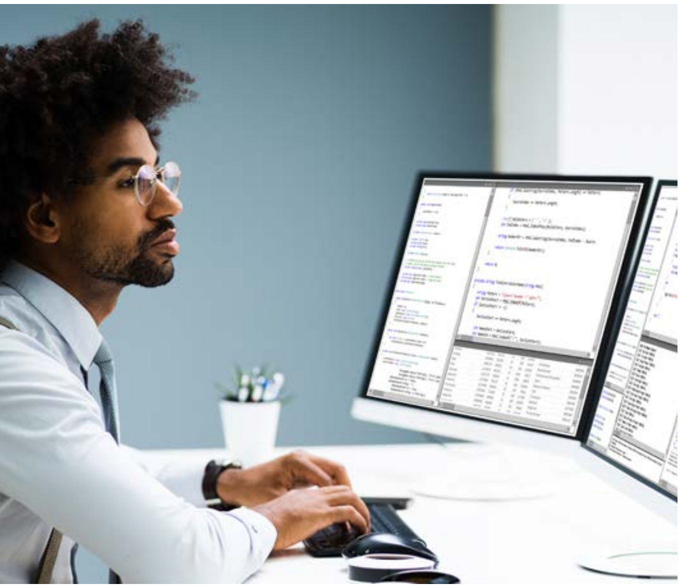


Figure 3:
An example d
of Parasoft C/
with Docker b
containers.



As a command line based tool, Parasoft C/C++test standard is suitable for in-container deployments. It can be packaged with the compiler and build system into one container image and used for CI/CD and deployed to developers' desktops for the local command line scans.

The tool accesses the containerized compiler (GNU GCC) and the runtime environments. In this specific setup, there are two separate Docker containers:

- One for the compiler and build tools
- Another for the execution environment (such as a stripped down version of embedded Linux)

The diagram also shows Jenkins using containerized C/C++test to run static analysis. In this specific setup, the tool, the compilation toolchain, and the build tools are deployed in the container shown in the top right corner of the diagram.

The container below it in the bottom right, provides the execution environment, which may be required to execute runtime tests, like unit tests or automated system level tests. If you on want to implement static analysis, then the setup will most likely include only one type of container with the compilation toolchain and Parasoft C/C++test.

BENEFITS OF CONTINUOUS INTEGRATION AND DELIVERY

The biggest benefit of CI/CD is reducing project risk. In the past, too many projects rely on "big bang" software integration efforts where software teams attempt to integrate their software too near the end of the product development. These teams encounter huge issues with integration and are often under massive "crunch" schedules to get the project finished.

Testing is pushed even further to the end of the project where it becomes too little too late. By using continuous integration, software teams always have a full build of the product ready for testing, delivery, and release. Rather than throw things together in one big bang, the team has gone through smaller integration steps, continuously, to find issues early and reduce the risk from late-cycle integration.

Here are more benefits to continuous integration. Think of this list as incremental and contributing to reduced risk and better quality.

- **Integration testing is early and often,** which means bugs are exposed earlier where they can be fixed easier and more cheaply.
- **Regression testing starts earlier** so that new features can be tested to see how they impact existing code. New tests are added to the regression test suite after each iteration.
- **Incremental improvement** of the product in terms of new features added and tested and bugs removed. It's easier to build in quality and security in an incremental fashion.
- **Enables continuous testing and delivery** which are equal parts of the continuous development process. Continuous integration alone isn't effective without continuous testing and delivery.

CONTINUOUS INTEGRATION AND DELIVERY NEEDS CONTINUOUS TESTING

Continuous integration is just part of a continuous development process that needs testing and delivery to reap the benefits of the approach.

Continuous testing provides an automated, unobtrusive way to obtain immediate feedback on a software release candidate. Continuous testing isn't simply more test automation. The purpose is to build quality and security into the product as part of a continuous integration/release/delivery process. Some of the activities include:

- **Static analysis for early detection of bugs and security vulnerabilities.** Early detection, usually at the developer's desktop, prevents bugs from wasting unit testing time and entering the software build.
- **Coding standard enforcement** helps conform to required industry standards (such as MISRA C/C++ or SEI CERT C) and prevents whole classes of defects and poor coding practices from entering the build to become larger issues later on.
- **Automated test execution** is needed as soon as the application is built. The required tests that need to verify units also include nonfunctional, load, security, and performance testing. These tests are executed directly from the CI orchestration system. The results from these tests get pulled back into the same build and gathered. Code coverage (statement, branch, & MC/DC) information is cross referenced by unit, file, test, and build number.
- **Requirements traceability** correlates code, tests, and other assets with business requirements. This provides an objective assessment of the requirements that are working as expected, which ones require validation, and the ones at risk.
- **Test impact analysis** provides direction to the team on where testing efforts need to go. From a risk perspective, changed code impacts more than the software itself, it impacts relevant tests and assets. As code changes are made, are new tests needed or existing tests modified? What are the impacts on dependencies? Automation helps teams focus just on the tests that are impacted.
- **Test data management** significantly increases the effectiveness of a continuous testing strategy. Good test data and test data management practices increase coverage and drive more accurate results. However, developing or accessing test data can be a considerable challenge, in terms of time, effort, and compliance.

  Data generation underpins continuous testing because you can continuously generate data appropriate for the type of scenario you're trying to execute instead of trying to rely on production data sources and hoping that all the right data is in the right place. Combining data generation with simulation will allow you to inject the right data in the right place at the right time.

CI/CD & CONTINUOUS TESTING: THE FOUNDATION FOR DEVSECOPS

It's worth mentioning that DevOps and DevSecOps methodologies share the use of automation and continuous processes for establishing collaborative cycles of development. While DevOps prioritizes delivery speed, DevSecOps shifts security to the left, which is more important in software that's classified as embedded safety- and security-critical.

DevSecOps represents a shift in software development processes that stresses collaboration between end users, developers, and IT professionals. Software test automation can enhance these connections and help organizations accelerate secure software development.

Software test automation plays an important role but it's just one piece of the DevSecOps puzzle. Testing is often one of the greatest constraints in the software development life cycle (SDLC) so optimizing security processes that allow testing to begin earlier—and shrink the amount of testing required—has a significant impact on the security of the software and development efficiency.

Adopting a continuous testing process (more than just automated tests running regularly) helps promote the 6 Pillars of DevSecOps:

- Collective responsibility
- Collaboration and integration
- Pragmatic implementation
- Compliance and development
- Automation and measurement
- Monitoring and reporting

SHIFT-LEFT SECURITY IN DEVSECOPS

The drive to shift-left security in the SDLC comes from the desire to find and fix bugs and security vulnerabilities as early as possible. Issues are much easier, cheaper, and less risky to fix earlier, not later. Common sense, but the software industry is full of examples where critical defects caused catastrophic results.

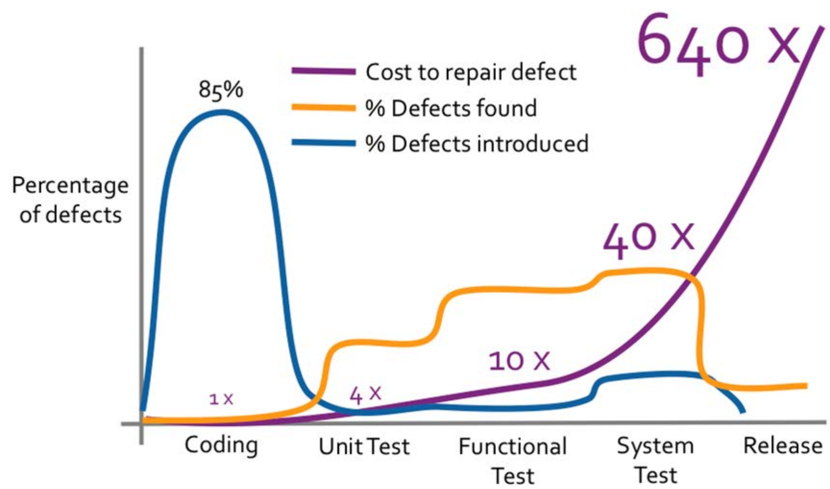


Figure 4:
Finding and fixing security vulnerabilities early is cheaper and less risky.

The essential requirements to shift-left security center around the need to incorporate security into any and all applications at the very beginning. Security can't be added on. It must be built in. Here are some recommendations to shift-left security in the DevSecOps pipeline that help create the necessary platform for continuous testing.

» Accelerate security

» Improve test automation

» Increase code coverage

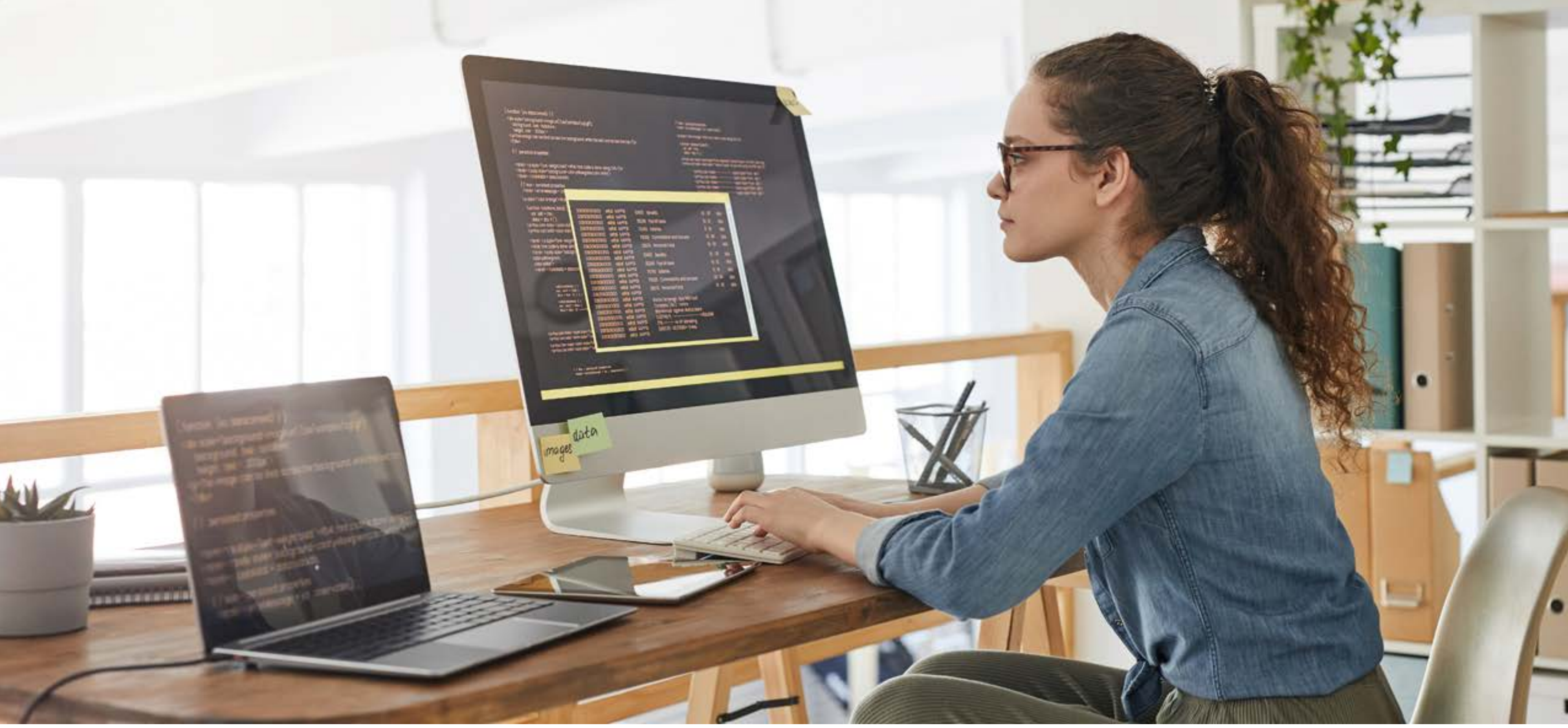
» Optimize testing with smart test execution

» Automate bidirectional traceability

ACCELERATE SECURITY IN DEVSECOPS

Modern DevSecOps initiatives require the ability to assess the risks associated with a release candidate—instantly and continuously. Continuous testing provides an automated, unobtrusive way to obtain immediate feedback on the security risks associated with a software release candidate. It guides development teams to meet security requirements and helps managers make informed trade-off decisions to optimize the release candidate.

Continuous testing delivers a quantitative assessment of risk as well as actionable tasks that mitigate risks before they progress to the next stage of the SDLC. The goal is to eliminate meaningless activities while improving quality and security and driving development towards a successful release.



IMPROVE TEST AUTOMATION TO OPTIMIZE THE CI/CD

It should be clear at this point that test automation is a key aspect to achieve both quality and security in a CI/CD pipeline. In turn, it becomes clear that test automation needs to be a focus for improvement and optimization.

The largest struggle teams face is what to test. Since full system testing with each new candidate release is too time consuming and expensive, teams inevitably compromise testing by picking parts of the test suite to execute.

Focusing testing on exactly what is needed to increase code coverage and determine which regression tests are needed after each code change is critical to accelerating testing, enabling continuous testing, and accelerating the DevSecOps pipeline.

INCREASE CODE COVERAGE

In general, code coverage is a measurement of how much of the production code is executed while your automated tests are running. By running a suite of tests and looking at code coverage data, there is a general sense of how much of the application is being tested.

There are multiple kinds of code coverage. For embedded systems, you need to be familiar with statement, branch, and MC/DC. For the strictest requirements, like in avionics software, object code verification or assembly language code coverage may be required.

Structural Code Coverage

Collecting and analyzing code coverage metrics is an important aspect of safety critical embedded 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 demonstrates the absence of unintended behavior—code that isn't covered by any test is a liability since 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. Below are examples of types of recommended code coverage.

- **Statement coverage** requires that each program statement be executed at least once (branch and MC/DC coverage encompasses statement coverage.)
- **Branch coverage** ensures that each possible decision branch (if-then-else constructs) is executed.
- **Modified condition/decision coverage (MC/DC)** requires the most complete code coverage to ensure test cases execute each decision branch and all the possible combinations of inputs that affect the outcome of decision logic. For complex logic, the number of test cases can explode so the modified condition restrictions are used to limit test cases to those that result in stand alone logical expressions changing. See this tutorial from NASA.

Advanced unit test automation tools like Parasoft C/++test provide all of these code coverage metrics and more. C/C++test automates this data collection 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.

Code Coverage With Automated Unit Test Case Creation

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 their coverage goals which is still important even outside the realm of safety critical software. Any code not covered by at least one test is shipping untested!

Increasing code coverage can be challenging. Analyzing branches in the code and trying to find reasons why certain code sections aren't covered, continues to steal cycles from development teams.

Resolve Coverage Gaps

Teams can resolve coverage gaps in test suites using a coverage advisor. Parasoft discovered how to use advanced static code analysis (data and control flow analysis) to find values for the input parameters required to execute specific lines of uncovered code.

This analysis computes preconditions for function parameters, global variables, and external function calls required to execute a specific line of code. The Coverage Advisor view presents a collection of solutions for the user-selected lines of code. Presented values are used for creating new unit test cases. The functionality boosts the productivity of developers working on unit test cases to improve code coverage.

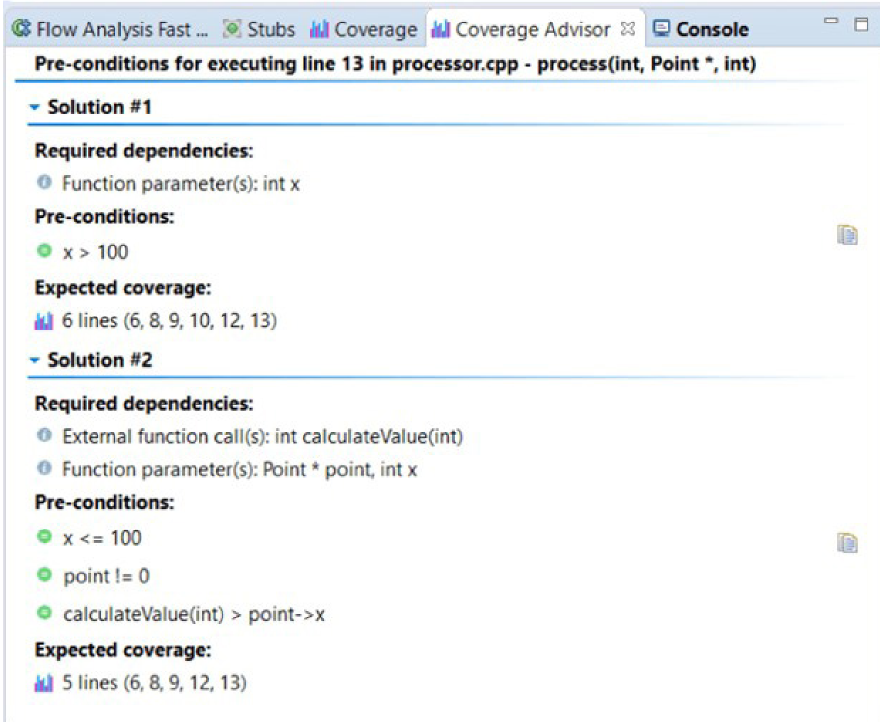


Figure 5:
Coverage Advisor displays what input values, global variables, and external calls are needed for a test case to obtain the needed code coverage.

Each coverage solution includes:

» **Required dependencies.** Dependencies that need to be customized to cover the selected line. These may include function parameters, external function calls, global variables, local variables, and class members.

» **Preconditions.** Conditions that must be satisfied by the required dependencies to cover the selected line. Clicking a precondition navigates to the related code line.

» **Expected coverage.** Code lines that will be covered if all of the preconditions are satisfied.

OPTIMIZE TESTING WITH SMART TEST EXECUTION

To accelerate testing in a continuous pipeline, smart test execution on a per-build basis is needed to reduce the set of tests required to be executed to address the risk that each new iteration has introduced. The analytics provided by test impact analysis are key to making testing focused on only what absolutely needs to be tested rather than the shotgun approach used otherwise.

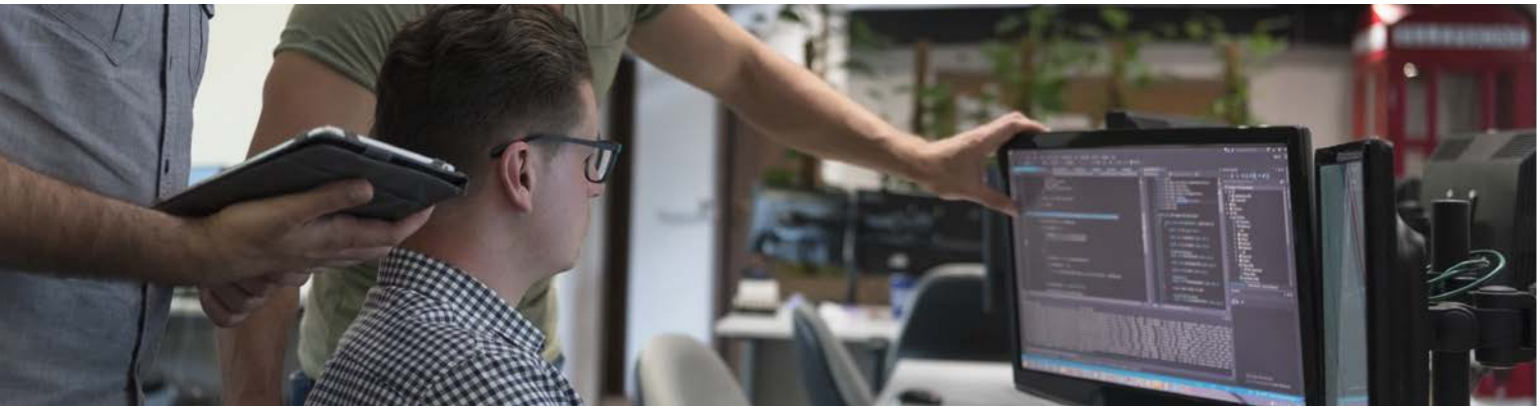
Only through smart, can data-based decision making enable continuous testing. Focusing the development team on the minimum set of tests to ensure proper coverage at each iteration is the key to bring the agility back to Agile development methods.

Smart test execution in Parasoft C/C++test is extended with plugins for CI systems (Jenkins, TeamCity, Bamboo, and so on) for advanced functionality to help software development organizations reduce bottlenecks associated with running continuous builds. The same capabilities are available inside IDE environments with dedicated plugins that access a centralized coverage image through a REST API, and determine which tests need to be executed locally inside the IDE to verify all changed code.

On the development side, experienced developers might apply the proper structure in organizing their tests and run only a subset of them manually, but they still might not know which tests need execution to verify ALL changes.

Teams using CI might rely on nightly builds to execute all tests automatically overnight and get feedback the next day, but only if it's possible to execute the total number of tests in under 12 hours.

Unfortunately, most software development teams are running their day-to-day operations accepting those unscalable testing practices. The situation becomes even more difficult when manual testing is involved. Traditional software development organizations are still following testing practices represented by an inverted pyramid, which emphasizes running manual tests over automated tests for one reason or another.



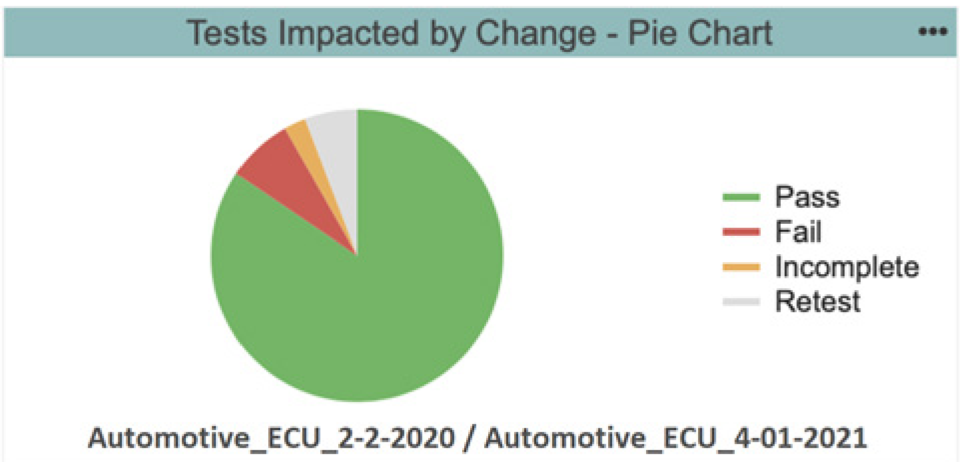


Figure 6:
An example high-level chart provided showing the distribution of tests that passed, failed, are incomplete, and need retesting. Users can click the pie chart to view more details.

Smart test execution uses test impact analysis to trace the execution of manual tests against applications and associated, captured code coverage information with those tests. Similar technology is used for automated testing. This analysis figures out which manual tests need execution to access changed functionality delivered with every new build. Thus, smart test execution is critical at the developer and tester levels in their local IDEs. It enables them to focus the testing where it's needed, removing guesswork and extra "just in case" work.

PARASOFT admin

Change Based Testing - Files

Filter: Parabank-v3 **Baseline Build:** PARABANK3-20170503 **Target Build:** PARABANK3-21170619 **Coverage Tag:** Parabank-All

Totals -- Pass: 172 Fail: 7 Incomplete: 10 Retest: 6

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:
An example change-based testing report showing areas of the code which are and are not tested.

AUTOMATE BIDIRECTIONAL TRACEABILITY

Requirements traceability is defined as “the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e. from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases).”

In the simplest sense, requirements traceability is needed to keep track of exactly what you’re building when writing software. This means making sure the software does what it’s supposed to and that you’re only building what's needed.

Traceability works both to prove you satisfied the requirements and to identify what doesn’t. If there are architectural elements or source code that can’t be traced to a requirement, then it’s a risk and shouldn’t be there. The benefits go beyond providing proof of the implementation. Disciplined traceability is an important visibility into development progress.

It’s important to realize that many requirements in embedded software are derived from safety analysis and risk management. The system must perform it’s intended functions, of course, but it must also mitigate risks to greatly reduce the possibility of injury. Moreover, in order to document and prove that these safety functions are implemented and tested fully and correctly, traceability is critical.

Maintaining traceability records on any sort of scale requires automation. This is particularly important in a CI/CD pipeline since manual maintained traceability would slow down each iteration. Application life cycle management tools include requirements management capabilities that are mature and tend to be the hub for traceability.

Integrated software testing tools can complete the verification and validation of requirements by providing an automated bidirectional traceability to the executable test case, which includes the pass or fail result and traces down to the source code that implements the requirement.

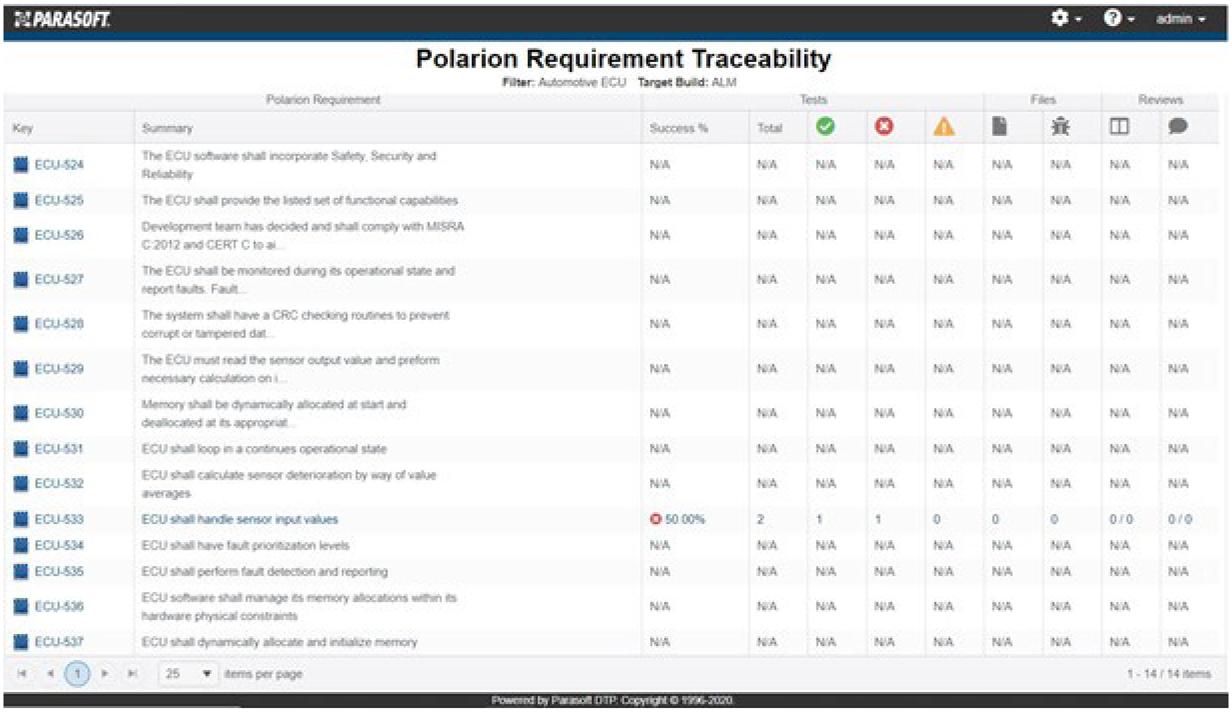
Parasoft integrates with market-leading requirements management and Agile planning systems like Intland, codebeamer, Polarion from Siemens, Jama Connect, Atlassian Jira, CollabNet, VersionOne, and TeamForge.

As shown in the image below, each of Parasoft's test automation tools (C/C++test, Jtest, dotTEST, SOAtest, and Selenic) support the association of tests with work items defined in these systems, such as requirements, stories in IT, defects, test runs, and test cases). Traceability is managed through a central reporting and analytics dashboard, Parasoft DTP.

Figure 8: Bidirectional traceability from work items to test cases and test results. Traceability reports are displayed, and results are sent back to the requirements management system.

Parasoft DTP correlates the unique identifiers from the management system with static analysis findings, code coverage, and test results from unit, integration, and functional tests. Results are displayed within traceability reports and sent back to the requirements management system. They provide full bidirectional traceability and reporting as part of the system's traceability matrix.

The traceability reporting is highly customizable. The following image shows a requirements traceability matrix template that traces to the test cases, static analysis findings, source code files, and manual code reviews.



Polarion Requirement		Tests					Files		Reviews	
Key	Summary	Success %	Total							
ECU-524	The ECU software shall incorporate Safety, Security and Reliability	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-525	The ECU shall provide the listed set of functional capabilities	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-526	Development team has decided and shall comply with MISRA C:2012 and CERT C to ai...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-527	The ECU shall be monitored during its operational state and report faults. Fault...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-528	The system shall have a CRC checking routines to prevent corrupt or tampered dat...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-529	The ECU must read the sensor output value and preform necessary calculation on i...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-530	Memory shall be dynamically allocated at start and deallocated at its appropriat...	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-531	ECU shall loop in a continues operational state	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-532	ECU shall calculate sensor deterioration by way of value averages	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-533	ECU shall handle sensor input values	50.00%	2	1	1	0	0	0	0 / 0	0 / 0
ECU-534	ECU shall have fault prioritization levels	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-535	ECU shall perform fault detection and reporting	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-536	ECU software shall manage its memory allocations within its hardware physical constraints	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ECU-537	ECU shall dynamically allocate and initialize memory	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Figure 9: Requirements traceability matrix template

The bidirectional correlation between test results and work items provides the basis of requirements traceability. Parasoft DTP adds test and code coverage analysis to evaluate test completeness. Maintaining this bidirectional correlation between requirements, tests, and the artifacts that implement them is an essential component of traceability.

Bidirectional traceability is important so that requirement management tools and other life cycle tools can correlate results and align them with requirements and associated work items.

SUMMARY

Continuous integration and delivery is common place in embedded development. Migrating a waterfall process to CI/CD and Agile development pays off in risk reduction and quality and security improvements. Security is top of mind for embedded developers and CI/CD is an enabler for DevSecOps, which introduces security requirements and controls into all aspects of the pipeline.

Containers are a perfect fit with CI/CD as they support rapid deployment and portability across different host environments with support for versioning and centralized control. Containerized development environments are important for secure development in a DevSecOps pipeline, making it possible to provide a reproducible application environment with built-in security controls.

Continuous testing is a necessary component of a well oiled CI/CD pipeline because testing is by far the most time and resource consuming activity. Continuous testing provides a framework to shift testing earlier in the life cycle.

With the right application of automation and focus on the highest risk areas of the application, it's possible to streamline testing to be less of an inhibitor in continuous processes. Continuous testing requires tools support for automation and optimization. Continuous testing is further improved with tools that drive larger code coverage, smart test execution and bidirectional traceability.

TAKE THE NEXT STEP

Learn how your embedded software development team can streamline testing with continuous integration and continuous delivery. Contact one of our experts today to request a demo.

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.