Technical Whitepaper

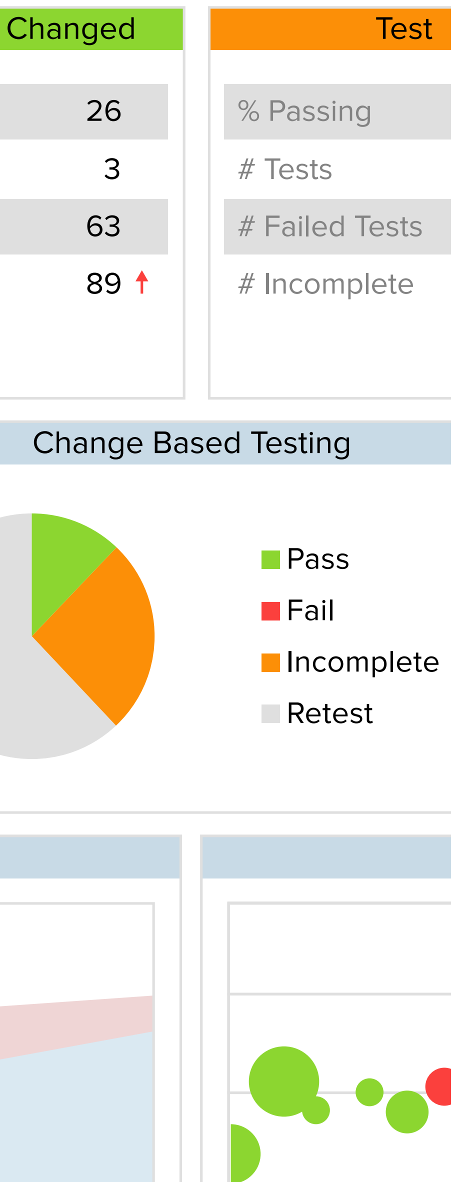
Streamlining Unit Testing for Embedded and Safety Critical Systems

INTRODUCTION

Unit testing is a fundamental software practice and, if done correctly, can increase the speed and quality of software development while simultaneously reducing risk and overall costs. In safety critical systems, unit testing is not only required to qualify software for intended use, but it's also a cornerstone of safety and security verification and validation.

Unfortunately, unit testing is expensive in terms of time and resources. Traditionally, someone with programming expertise is required to design, develop, and maintain unit tests. These unit tests, once written, are also fragile to changes in the software, and maintenance burdens the project as a whole.

This paper outlines the key elements needed to successfully address unit testing in a way that is scalable across the entire project and organization, enabling more of the software team as a whole to contribute to the unit tests – not only the engineers.



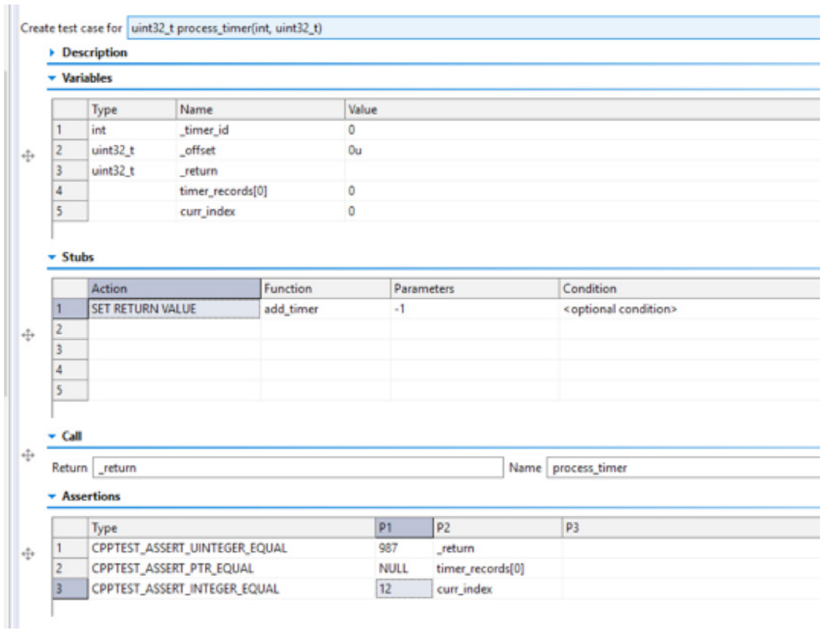
IT STARTS WITH TEST CREATION

When done without the assistance of automation tools, unit test creation is a tedious and time-consuming process. The unit tests must be coded by software engineers familiar with both the unit under test and the complete development and execution environment. Test automation tools, such as Parasoft C/C++test, reduce this complexity by integrating directly into the developers' IDE (such as Visual Studio or Eclipse-based IDEs) and providing a framework for unit testing that aids with creation and execution of automated unit tests.

One of the challenges when adopting an automation tool in support of unit testing is the mix of skills and preferences of the team members. Some team members will want to go deep and direct to the code (some without using an IDE), while others will want to stay at a higher level, focusing on test logic and parameterization. The solution is to use convenient graphical editors and views, along with source code based representations of the test artifacts, such as test cases and stubs.

This process enables experienced developers to quickly build their test cases using source code-based APIs, while others team members (such as test/QA engineers) can augment the process of test creation with graphical editors. Parasoft C/C++test provides a test editor that captures UI interactions and creates source tests that can be persisted at source or maintained as UI driven tests, allowing the team to determine the best way to create, augment, and maintain tests.

PARASOFT®
Automated Software Testing



Create test case for uint32_t process_timer(int, uint32_t)

Description

Variables

	Type	Name	Value
1	int	_timer_id	0
2	uint32_t	_offset	0u
3	uint32_t	_return	
4		timer_records[0]	0
5		curr_index	0

Stubs

	Action	Function	Parameters	Condition
1	SET RETURN VALUE	add_timer	-1	<optional condition>
2				
3				
4				
5				

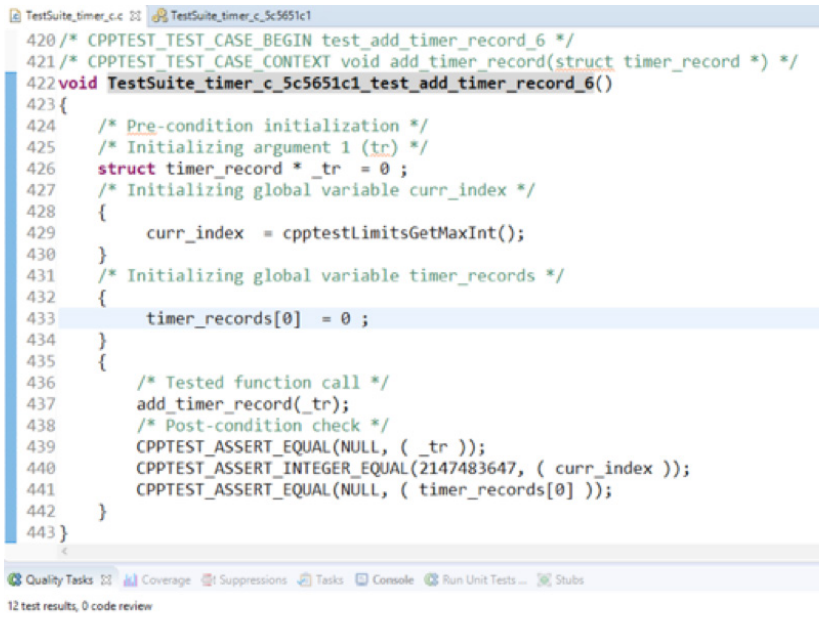
Call

Return _return Name process_timer

Assertions

	Type	P1	P2	P3
1	CPPTEST_ASSERT_UINTEGER_EQUAL	987	_return	
2	CPPTEST_ASSERT_PTR_EQUAL	NULL	timer_records[0]	
3	CPPTEST_ASSERT_INTEGER_EQUAL	12	curr_index	

There are advantages to having test cases stored as source code, and values for the tested code parameters initialized in the source code. This strategy is much closer to the process of parameter initialization in the production code, which makes it better suited for testing safety critical code, especially in embedded devices. In many embedded development projects, test artifacts need to be as close as possible to the code that is going to be run on the device and run on the intended target, rather than host-based testing alone.



TestSuite_timer_c.c TestSuite_timer_c_5c5651c1

```
/* CPPTEST_TEST_CASE_BEGIN test_add_timer_record_6 */
/* CPPTEST_TEST_CASE_CONTEXT void add_timer_record(struct timer_record *) */
void TestSuite_timer_c_5c5651c1_test_add_timer_record_6()
{
    /* Pre-condition initialization */
    /* Initializing argument 1 (tr) */
    struct timer_record * _tr  = 0 ;
    /* Initializing global variable curr_index */
    {
        curr_index  = cpptestLimitsGetMaxInt();
    }
    /* Initializing global variable timer_records */
    {
        timer_records[0]  = 0 ;
    }
    {
        /* Tested function call */
        add_timer_record(_tr);
        /* Post-condition check */
        CPPTEST_ASSERT_EQUAL(NULL, ( _tr ));
        CPPTEST_ASSERT_INTEGER_EQUAL(2147483647, ( curr_index ));
        CPPTEST_ASSERT_EQUAL(NULL, ( timer_records[0] ));
    }
}
```

12 test results, 0 code review

ISOLATING COMPONENTS TO IMPROVE AUTOMATION

Unit tests require much more than just a harness to execute and retrieve results from the unit under test. In fact, the trickiest part is isolating a unit from its external dependencies so that only the unit itself is being tested. This makes it easier to have stable automation, and, when a problem is found, we know the problem is in the unit and not somewhere hidden in one of its dependencies. Stubs or mocks (sometimes called test doubles) are used for unit isolation but programming and controlling the response from stubs can become complicated.

The difficulty of stubbing the tested system depends on how well the code was designed to be testable, and how complicated its interactions with the outside world are. For example, a stub or mock needed to emulate a hardware function call would need to respond with reasonable values during the test process and could simply use hard-coded values or may have to read test data dynamically from an external data source.

Flexible Stubbing Options

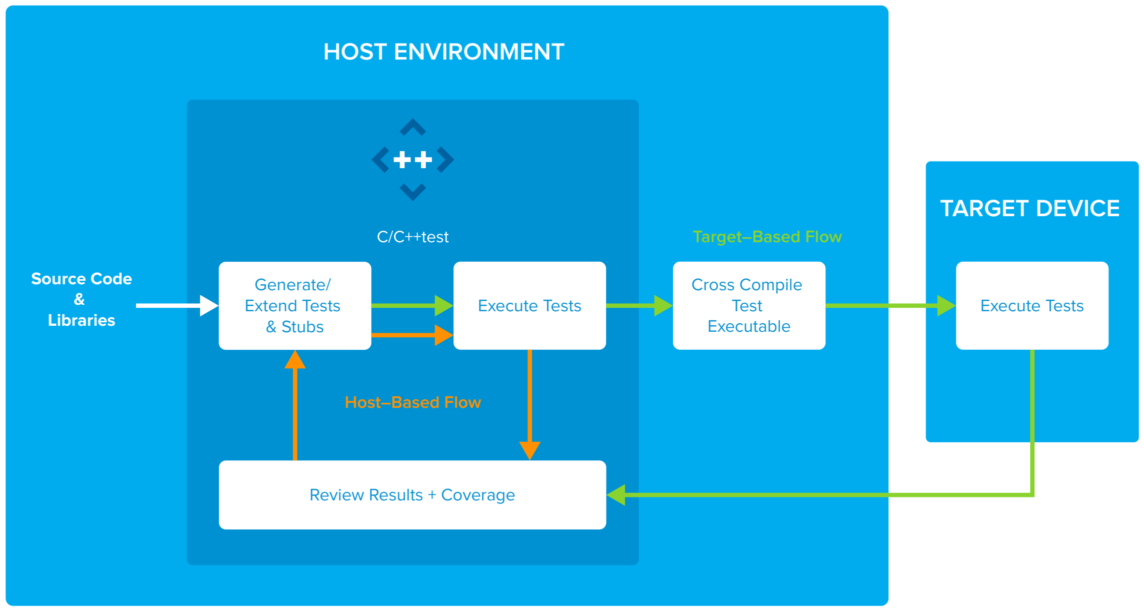
In the case of Parasoft C/C++test, stubbing of dependencies are the preferred approach for code isolation. Unlike mocking frameworks, stubs don't require you to re-architect your code and do not require any special code design to apply stubs (e.g. virtual functions).

Using stubs as call-backs enables easy association between tests and stubs, while a global stubbing framework can be reconfigured at execution time to enable flexibility between stubbed functionality or real code, such as done during integration tests. This means the same test cases can be verified in isolation and integration mode and results compared for any potential differences. Changing the mode of testing is a simple operation and requires only the specification of the new stubs suite location.

TEST EXECUTION

In some applications, test execution is a not a concern because tests can be executed and results collected on the development machine. For embedded systems, and in particular safety critical systems, tests must be run on the intended target system. The target system may have a completely different architecture than a typical desktop system, with a different operating system and performance characteristics.

Manual test execution on embedded targets is another time-consuming task that test automation alleviates. The automation of test execution, both local and remotely to a target system, is a great time saver. Moreover, in the case of Parasoft C/C++ test, remote execution includes runtime analysis, code coverage collection, and test results extraction. The following block diagram illustrates local and remote test execution in Parasoft C/C++test:



WHEN IS TESTING FINISHED?

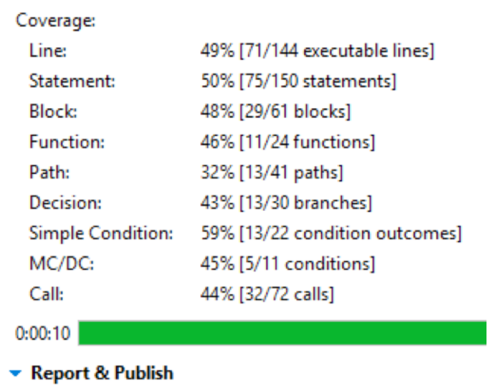
It seems like an innocuous question, but the answer of what it means to be finished testing is different to different people. In many projects, testing is finished when you run out of time to test anymore; although, officially, it's when the functionality of the application has been proven. So, one answer can be "when all of the requirements have been proven to be met."

Of course, the same answer isn't sufficient in some applications. What about performance, security, and other non-functional requirements? In safety critical systems, it's required to not only prove functionality, security, and safety but also to prove that there isn't some lingering problem in the code untouched by tests. Testing completion depends on many factors, but in each case, tool automation is key to keeping track of the various metrics to show project status, whether that's code or requirements coverage.

Code coverage analysis is often used in conjunction with the execution of unit tests. The code is instrumented and, as the tests execute, data about the parts of the code that are executed is captured. Additionally, some manual functional testing of the application may be required to exercise code that's difficult to run with unit tests (i.e. requiring integration or higher system-level tests). Therefore, to obtain accurate and complete code coverage information, data from both unit and functional tests needs to be merged, taking into account overlapping coverage. This is very important if the system is a safety-critical application where accurate coverage is necessary product certification or qualification.

With a code coverage tool in place, it is much easier to analyze the quality of test cases. Experience shows that even after executing all test scenarios that seem to cover all requirements, some portions of code may not have been executed. Analyzing uncovered sections of the code allows you to make a decision about whether additional testing is required.

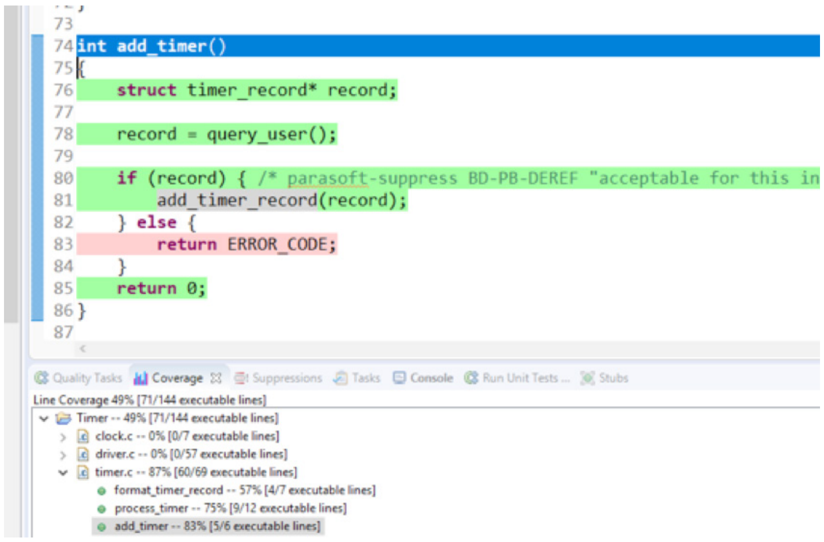
It is common that those uncovered sections of the code require special attention — it may be error-handling code or code for processing unexpected inputs. It may appear that the easiest way for verifying if such code works correctly is by applying unit tests together with stubbing/mocking frameworks to override standard behavior of software components and to force code execution to go through rare paths.



```
Coverage:
  Line:               49% [71/144 executable lines]
  Statement:          50% [75/150 statements]
  Block:              48% [29/61 blocks]
  Function:           46% [11/24 functions]
  Path:               32% [13/41 paths]
  Decision:           43% [13/30 branches]
  Simple Condition:   59% [13/22 condition outcomes]
  MC/DC:              45% [5/11 conditions]
  Call:               44% [32/72 calls]
0:00:10
```

The figure above shows different coverage metrics that can be used to measure test coverage, each with different levels of sophistication, ranging from the most basic metrics (i.e. *Line* and *Statement*) to more comprehensive metrics such as *Modified Condition/Decision Coverage (MC/DC)*. Which metric to use is typically determined by the level of risk associated with the code being tested. For example, with ISO 26262, the Automotive Safety Integrity Level (ASIL) defines which coverage types should be used: Level A can use just *Statement*, while Level D requires *Branch* and *MC/DC*.

Here's an example of a code coverage report directly in the IDE from Parasoft C/C++test, showing lines covered and not covered during test execution:



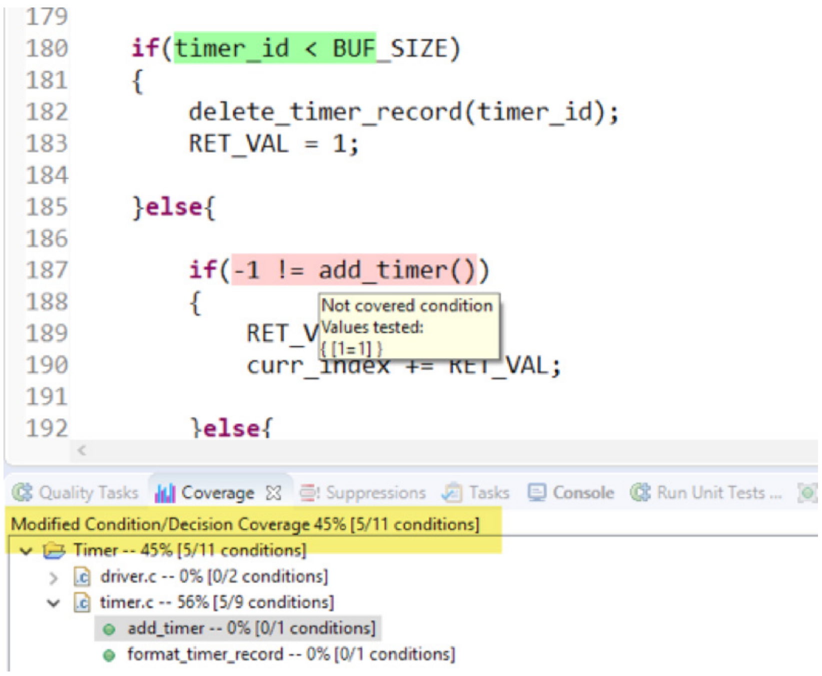
```
int add_timer()
{
    struct timer_record* record;

    record = query_user();

    if (record) { /* parasoft-suppress BD-PB-DEREF "acceptable for this in
        add_timer_record(record);
    } else {
        return ERROR_CODE;
    }
    return 0;
}
```

```
Line Coverage 49% [71/144 executable lines]
  Timer -- 49% [71/144 executable lines]
    clock.c -- 0% [0/7 executable lines]
    driver.c -- 0% [0/57 executable lines]
    timer.c -- 87% [60/69 executable lines]
      format_timer_record -- 57% [4/7 executable lines]
      process_timer -- 75% [9/12 executable lines]
      add_timer -- 83% [5/6 executable lines]
```

Looking at another example, we can see a particular condition not covered as part of MC/DC:



```
    if(timer_id < BUF_SIZE)
    {
        delete_timer_record(timer_id);
        RET_VAL = 1;

    }else{

        if(-1 != add_timer())
        {
            RET_V
            curr_index += RET_VAL;

        }else{
```

```
Not covered condition
Values tested:
{ [1=1] }
```

```
Modified Condition/Decision Coverage 45% [5/11 conditions]
  Timer -- 45% [5/11 conditions]
    driver.c -- 0% [0/2 conditions]
    timer.c -- 56% [5/9 conditions]
      add_timer -- 0% [0/1 conditions]
      format_timer_record -- 0% [0/1 conditions]
```

FINDING THE ONE THAT GOT AWAY

Despite a significant investment in unit testing, there are still going to be defects, particularly security vulnerabilities that make it through unit testing and out into later stages of development. In the worst case, these bugs make into production code, where they are expensive to fix and come with other costs such as recertification or actual safety or security incidents. There are two important techniques available that have both low cost and low overhead of implementation: static analysis and runtime analysis.

Static analysis is the inspection of source code for deviations from best practices that can lead to bugs, security vulnerabilities, and reliability issues in production environments. Runtime analysis, on the other hand, is the instrumentation and monitoring of runtime execution of applications to detect 'real' defects at runtime.

Static Analysis

The static analysis capability in Parasoft C/C++test verifies C and C++ code quality and checks compliance with security, functional-safety, and industry standards (such as CERT, MISRA C, or JSF AV C++) by applying a wide range of checkers (rules) and using a comprehensive set of analysis techniques such as pattern-based analysis, data/control flow analysis, and software metrics.

Static analysis performs the following functions:

- **Analyzing code compliance with industry standards.** Static analysis provides an easy way to check for verifying compliance with standards like MISRA C 2012, MISRA C++ 2008, JSF AV C++, SEI CERT C/C++, AUTOSAR C++14, HIC++, and more. Such analysis is recommended/required for regulated industries (automotive, medical, aviation, etc.) and functional safety development (with standards like ISO 26262, IEC-61508 or DO-178-B/C).

- **Detecting runtime problems without executing code.** With the data flow static analysis, detection of complex runtime-like problems is possible early in the development stage – without the need to execute costly runtime tests. Using flow analysis to analyze execution paths through the code, it's possible to find issues like Null Pointer Dereferencing, Buffer Overflows, Division by Zero, Memory Leaks, and more.

- **Creating checkers for custom coding standards.** Static analysis can verify company- or team-specific guidelines and coding standards. For example, Parasoft C/C++test includes an editor for creating custom rules that can extend upon the built-in rules provided.

As with Unit Testing, Static Analysis can be performed either in the IDE or using the command-line interface (for automation / continuous integration scenarios). The results of the analysis can then be accessed immediately (in the IDE, or with HTML/XML reports) or aggregated into a centralized dashboard in Parasoft DTP for further post-processing, reporting, and analytics.

Runtime Analysis

Runtime analysis, as the name implies, finds runtime defects, stability issues, and security vulnerabilities such as Memory Leaks, Null Pointers, Uninitialized Memory, and Buffer Overflows, by monitoring the executing application both while executing unit tests and during manual execution of the application.

Runtime analysis includes the following:

- **Detecting runtime problems when running an application.** Runtime analysis monitors a running application, detecting runtime-related problems such as memory leaks, memory corruption, reading uninitialized memory, or buffer overflows that can lead to stability issues, functional misbehaviors, or security vulnerabilities.

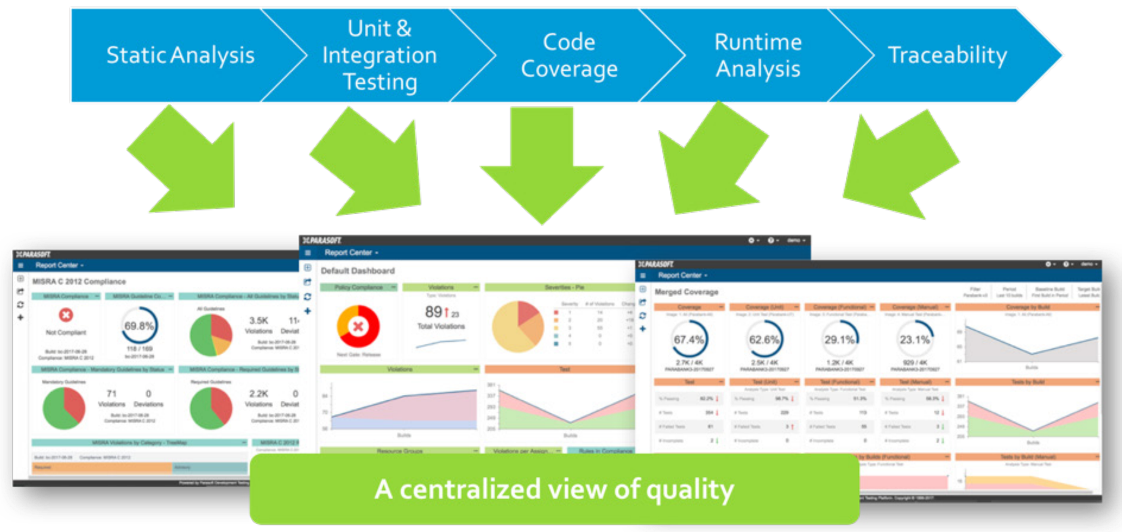
- **Detecting runtime problems when executing unit tests.** Runtime analysis can monitor a test-binary when running unit tests, providing additional insight into the code under test, that can help with understanding unit testing failures or instabilities.

- **Performing runtime checks for cross-platform environments.** Runtime Analysis can be performed not only for native applications but also for cross-platform and embedded environments, so the analysis is performed in the original "production" environment, which in many cases has limited testing and debugging capabilities.

- **Combining runtime analysis results with other test data.** Runtime defects are presented in the IDE in a unified way with other test data such as code coverage, unit testing failures, or static analysis findings, allowing for easier analysis and understanding the root cause of a runtime defect.

TYING IT ALL TOGETHER: ADVANCED REPORTING AND ANALYTICS

Reporting on the results of unit test automation alone doesn't provide enough information to correctly quantify the quality and risk of a project. The real power of test automation comes from aggregating data from across all test practices to provide a comprehensive view of quality.



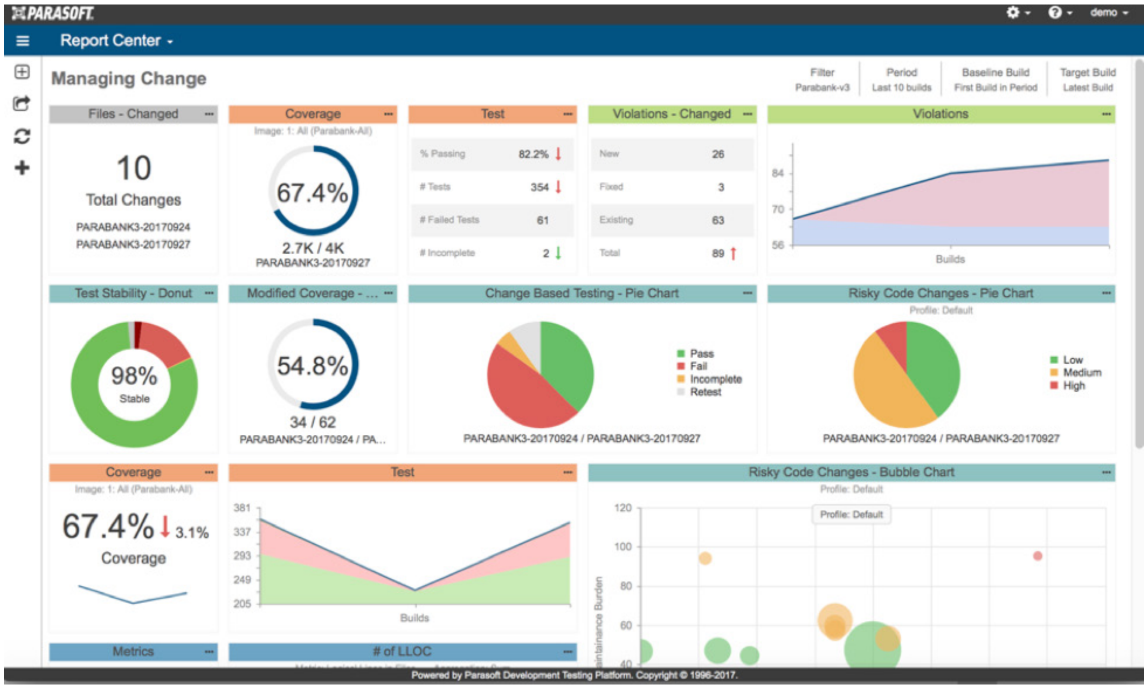
Parasoft DTP is the reporting and analytics dashboard for Parasoft C/C++ that provides an interactive web-based reporting capability for navigating metrics and test results, demonstrating compliance, investigating areas of concern, and implementing and enforcing project policies.

The data aggregated by Parasoft DTP powers advanced analytics to enable process driven improvements, gain greater insights into project status, and to streamline the team's testing activities.

The advanced analytics calculated by Parasoft DTP include:

- **Change-Based Testing:** The detailed code coverage provided by Parasoft C/C++test is correlated with the intersection of code changes with tests to identify the subset of tests that need to be re-executed to validate the changes. Change-Based Testing enables the team to discover regressions sooner and focus testing efforts on just the changes and their implications.

- **Modified Code Coverage:** Parasoft DTP analyzes the changes between builds to determine where the risky gaps in code coverage exist, helping to focus on creating the correct tests to cover these modifications.



- **Risky Code Changes:** Parasoft DTP allows for a customized, multi-dimensional, multi-variable definition of "risk" as it applies to the current project. The analysis engine can overlay this risk factor on the project codebase to indicate areas of highest risk. These areas might be suitable for additional testing and scrutiny.

- **Test Stability:** Because DTP tracks tests individually across builds, it analyzes patterns in the test execution to identify unstable tests. This enables focus on true regressions rather than wasting time chasing instabilities. Alternatively, if the focus is to improve test stability, these are quickly pinpointed, again independently of test failures which might be red herrings.

- **KPI Calculations:** Simply counting the number of static analysis violations is not sufficient to understand the amount of work or risk that exists within a codebase. Different organizations associate different levels of risk and different costs to different static analysis rules. To track this, Parasoft DTP provides customizable KPI calculations that build a consistent score card across various teams to quantify these values and monitor improvements.

- **Custom Analytics and Derived Metrics:** In addition to existing calculations, software teams can create and customize Parasoft DTP to measure whatever they wish. DTP can process imported data from external analysis tools, painting a complete picture of code quality. The data aggregated in DTP is also available via REST APIs, allowing for a wide range of external applications.

TOOL CERTIFICATION AND QUALIFICATION FOR SAFETY CRITICAL SYSTEMS

For organizations developing software in safety-critical industries, tool certification and qualification is mandated by safety standards, in order to validate that a particular software development tool is fit for its purpose. Unit testing is an important part of a safe software testing practice, so it's critical that results reported by unit test automation tools are correct.

Parasoft C/C++test is certified by TÜV SÜD for use in functional safety applications, and Parasoft provides Qualification Kits for Parasoft C/C++test to automate the process of creating the supporting documentation required for tool qualification of static analysis, unit testing, and coverage requirements, reducing both the potential for human error and time taken to perform tool qualification. These qualification kits help alleviate the extra burden placed on the development team by:

- **Automating the process of creating documentation required for tool qualification:** Automation greatly reduces the manual effort in the creation of documentation required for qualifying Parasoft C/C++test for use in safety-critical industries.

- **Reducing the scope of what's needed to qualify the tool:** Qualifying only the features of the software being used greatly reduces the scope of qualification and saves valuable time.

- **Executing automated tests:** The process of tool qualification cannot be fully automated, but the workflow of the qualification kit makes it as painless as possible by both managing manual testing efforts and performing execution of automated tests for the selected use-cases.

SUMMARY

Unit testing is a critical software testing technique, yet it introduces challenges for software teams, especially those in safety and security critical industries. Test automation is a clear solution to reduce the tedium, cost, risk, and complexity of unit testing, but the biggest benefits come from automating all aspects of the testing cycle – test creation, execution, results, and coverage data, along with project-wide metrics and static analysis. Parasoft C/C++test is a unified C and C++ development testing solution for enterprise and embedded applications that provides end-to-end test automation with a powerful analytics engine to help software teams understand the true status of their testing and to help them focus limited resources in the right direction.

www.parasoft.com

Parasoft Headquarters:
+1-626-256-3680

Parasoft EMEA:
+31-70-3922000

Parasoft APAC:
+65-6338-3628

PARASOFT®
Automated Software Testing