PARASOFT®

TECHNICAL WHITEPAPER

# The Value of Using a Unified C/C++ Testing Solution

## INTRODUCTION

Software verification and validation is an inherent part of software development. The effort and budget put into verification and validation projects depends on many factors, such as:

- Functional safety objectives of a project
- Level of business risk
- Quality culture of an organization

Regardless of what drives an organization to implement quality initiatives and processes, it takes more than determination to produce safe, secure, and high-quality software products. Implementing efficient and effective practices that lead to successful product releases and effortless software maintenance requires solid knowledge and experience, including the ability to choose the right combination of software testing techniques and tools that are appropriate for the intended level of quality.

Choosing appropriate testing methodologies is a challenging task for many reasons. Technology is evolving at a rapid pace. There's is a wide range of technologies available, but none of them offers a complete solution to all the problems. Choosing between open source and commercial products may also be difficult. In addition, it's crucial to consider how developers will be using the tool or technique in their daily work. Will it become a handy tool, or a boring duty they will try to avoid?

This paper explains how development teams can improve quality assurance processes by combining automated testing techniques including:

- Advanced static analysis
- Runtime memory monitoring
- Automated unit testing
- Flow analysis

Next, the paper explores the benefits of implementing a unified testing tool. The concepts discussed are generic and can be applied to any programming language. However, the examples were created with C/C++ programming languages and Parasoft C/C++test in mind. This paper doesn't dive into the details of testing techniques, however, it is focused on facets related to optimizing process that are built on the pillars of multiple testing technologies.

## THERE'S NO SILVER BULLET

When thinking about possible high-level software malfunctions, several classes of software errors can be distinguished:

- Errors resulting from missing requirements.
- Errors caused by incorrectly specified requirements.
- Errors that occur because requirements have been coded incorrectly.

The first two classes of software defects fall into the category of requirements engineering, and will not be discussed in this paper. The paper focuses on the third one, which embraces a wide range of potential software problems that many teams struggle with.

What does "a requirement has been coded incorrectly" mean?

It can be several things.

Let's examine an example of an implementation that doesn't satisfy one of the functional criteria stated in the requirements: A software module is expected to process two thousand samples per second.

The output computed by the implementation is perfectly correct in terms of values, but the result is invalid due to the fact that the response was not provided within the required time frame.

Also, this implemented algorithm may incorrectly compute the number of samples in the buffer by reading random values from outside the buffer. Or the initial state of variables may not be correctly set, resulting in transient errors.

Finally, the logic in the branching point may be coded incorrectly, resulting in dead code that will never be executed.

Although the above list is not exhaustive, it is obvious that there are different categories of errors, and the current state of the art does not effectively eradicate all of them with a single technology. Uninitialized memory, such as class members, can be detected by pattern-based static analysis, while detecting samples that are read from outside of the buffer requires advanced, flow-based static analysis or a runtime memory monitoring tool. Code that is never executed can be identified by analyzing code coverage reports or reported by a flow analyzer. Last but not least, is implementation of low-level requirements. Low-level requirements must have a set of corresponding unit tests, including performance checks and traceability links to the requirements they verify and validate.

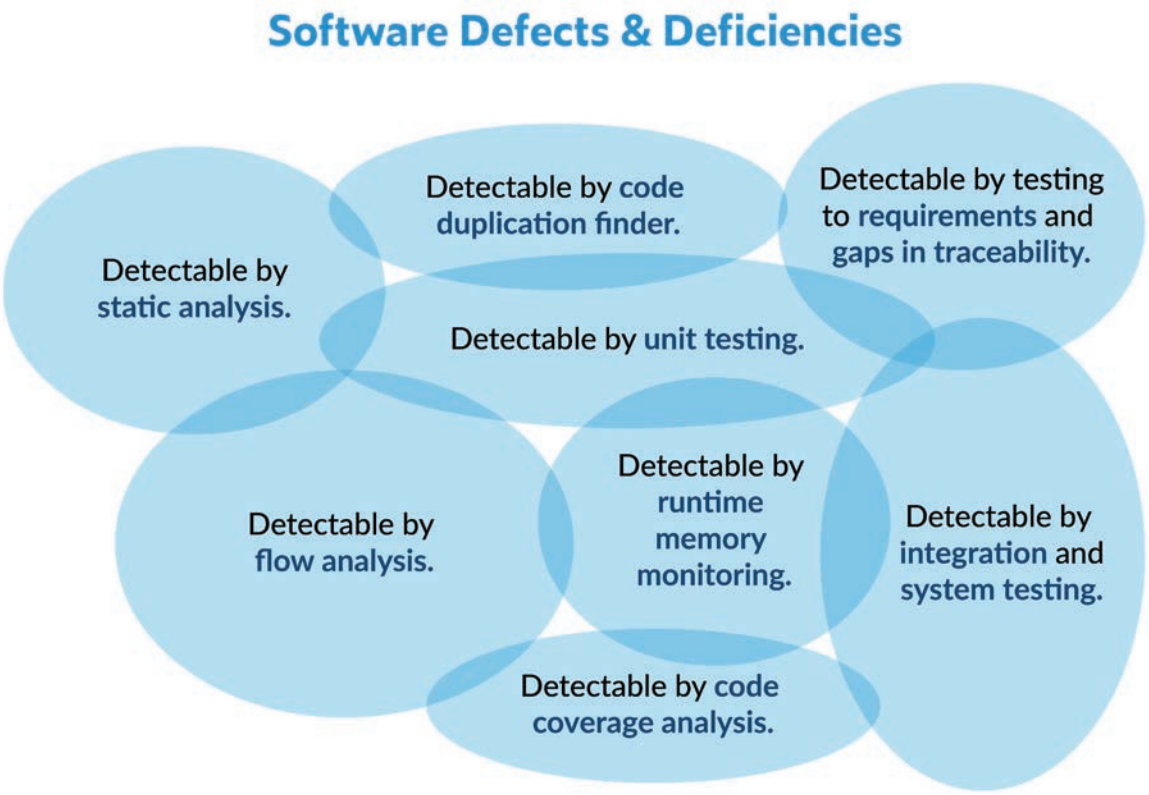



*Figure 1: Software defects and detection strategy landscape.*

Obviously, not all software projects need to use all available technologies for testing and improving software quality, and organizations face the dilemma of how to balance budget and quality. Projects that are related to functional safety will most likely select all available technologies to ensure uncompromised quality, as well as compliance with software safety standards, such as ISO 26262, DO-178C, or IEC 62304.

Other teams may decide to choose only static analysis and unit testing, as they seem to cover a significant portion of software defects, while some teams may find an open source unit testing framework sufficient. In any case, teams that design quality assurance processes must be aware of the landscape pictured in Figure 1, and understand the consequences of selecting a particular subset of available testing technologies.

Unwillingness to implement a wide range of testing technologies often stems from a concern that using multiple techniques imposes significant overhead on the pace of development and impact the budget. This is more likely if teams decide to select unintegrated tools. The cost of multiple separate tools, the learning curve, and the necessity to switch between different use models and interfaces can be problematic. As a result, developers will try to avoid using tools and automation, as it redirects their attention from writing code to tool usage, decreasing their productivity.

## THE VALUE OF HAVING A UNIFIED TESTING TOOL

It's important to specify the expectations of a unified testing tool before discussing its value. The tool should:

- Support multiple testing technologies.
- Be easy to use.
- Detect functional problems and regressions.
- Provide traceability from requirements to tests.
- Measure the complexity, portability, and maintainability of code.
- Educate developers by providing instant feedback as they are writing the code.
- Provide information about development progress.
- Combine results from different techniques for advanced analytics.

Unified testing helps avoid a number of problems, including:

- Multiple learning curves and usability issues with using detached tools with different interfaces.
- Distracting developers from writing code.
- Preventing the exchange of information between different elements of a toolchain.

## DETECTING AS MANY DEFECTS AS POSSIBLE

As it was mentioned in the introduction, software defects fall into different categories, so it cannot be expected that all of them are identified by one testing technique, for example, manual system level testing. To provide a more concrete example, observe the following example code snippet.

source1.c

```
1 int samples[5];
2 int limit = 10;
3
4 static int calculateIdx()
5 {
6     int sum = 0;
7     for (int i = 0; i < limit; i++) {
8         sum = sum + 1;
9     }
10    return sum;
11 }
12
13 void initialize(void)
14 {
15    int index = calculateIdx();
16    samples[index] = 0;
17 }
```

*Figure 2: Example source code.*

These few lines of code contain several issues. Specifically, in line 16 the developer is trying to initialize global buffer using the index value computed in the calculateIdx() function, but they fail to validate whether it is within the allowed range. Even if dozens of manual testing sessions are run, they may not reveal this issue, as writing an integer value to a random memory location rarely produces spectacular effect immediately.

But one day, most likely after the release, the memory layout may change, and the operation from line 16 will crash the application. Interestingly, static analysis may also fail to flag this problem because of the loop that is used to compute the index.

For performance reasons, flow analyzers used in static analysis tools have to apply heuristics and simplifications to finish code analysis within a reasonable time, and it is common to apply them in loops. For example, a flow analyzer may evaluate only the first few iterations, or try to guess which iterations from the range are more interesting than others. Whatever approach is used, it may work for one case but not for another. For this reason, flow analysis may not find the problem in this particular example.

Using the previous example, consider the use of a memory monitoring tool, which instruments source code by injecting special checks, and reports improper memory operations. This tool detects software problems only on paths that were executed, without making any guesses, a big advantage over static analysis, as the accuracy of reported problems is very high. In this example, Parasoft memory error detection easily identified the problem.

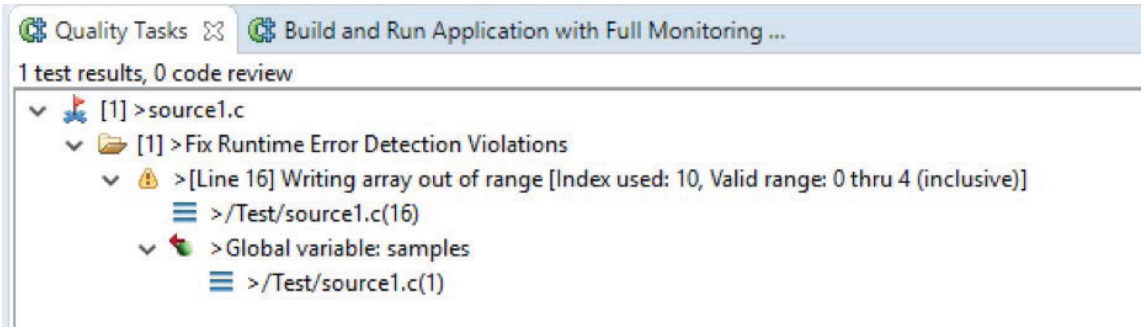



*Figure 3: Runtime error detection tool - violation report.*

A reverse situation is equally possible. Static analysis may detect issues that runtime memory monitoring cannot identify. For example, there is possibility of a null pointer dereference at the line 27/28 in the code snippet below. Calling the storePersonToFile function with the person pointer argument being null causes an error, but this occurs only if the retrivePersonFromDB function returns null. Such a condition is unlikely during system testing sessions, because the data base connection is most likely functioning as expected, so runtime memory monitoring tools remain silent. However, flow analysis within the static analysis tool easily detects the null pointer to be returned from this function, and reports a potential null pointer dereferencing problem.

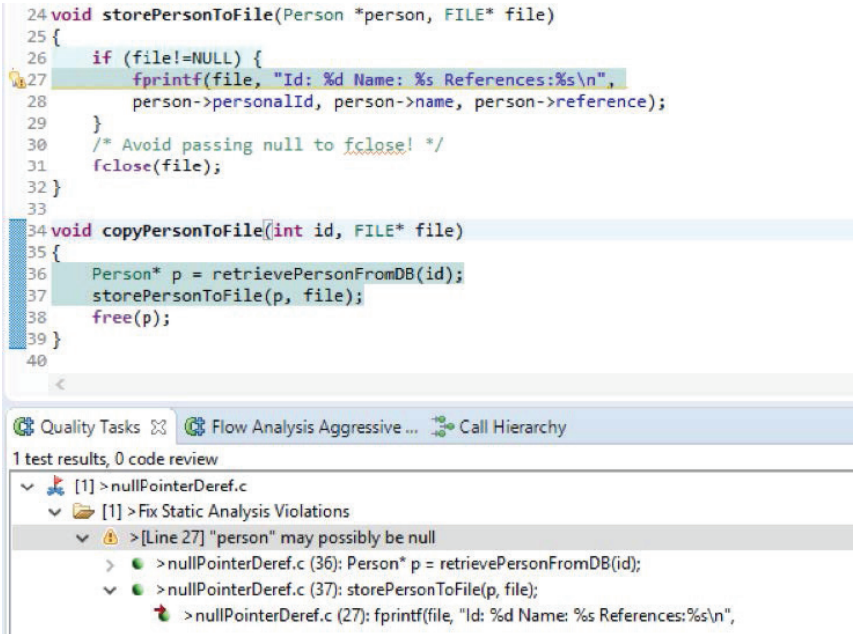



*Figure 4: Identified null pointer deferencing problem.*

## DETECTING FUNCTIONAL PROBLEMS & REGRESSIONS

The previous section discussed finding code defects that have the potential to crash an application under certain conditions. But what about code that always executes flawlessly, but does not comply with requirements?

Static and dynamic analysis aren't useful in identifying these kinds of issues. Detecting discrepancies between the expected result and the actual result requires comparison of computation results with predefined values. There are many popular ways of implementing such checks, including manual system-level testing, integration testing, and unit testing.

Unit testing requires developer effort in creating test cases, which typically include some kind of verification to ensure that results are correct, usually a set of automatic assertions. The technical concepts behind unit testing aren't covered in this paper. However, it is more interesting to focus on what a unified testing tool can bring into the software development process, and how it can increase productivity and efficiency without hindering developers.

There are many areas where a unified testing tool can facilitate the unit testing process. The list of benefits includes:

- Increasing developer productivity while creating test cases, for example, by providing graphical wizards for creating, editing, and configuring test cases.
- Integrating with stubbing/mocking frameworks, allowing easy mocking and stubbing to simulate complex test scenarios, without involving large code bases.
- Correlating test cases with code coverage reports to assess the completeness of testing.
- Optimizing testing sessions by automatically computing the minimal set of test cases required to verify the code delta.
- Merging code coverage reports from other testing techniques, such as manual/system level testing, or integration testing to identify untested code.
- Tracing results back to requirements for better understanding of the impact of failing tests.

Test creation with graphical wizards or editors can increase the productivity of an entire organization by engaging the QA teams in the process of writing unit tests, thus making them actively contribute to the development process. It's easier for QA team members to use graphic wizards than write appropriate test code in code editors, since configuring values using entry forms (as shown below) does not require advanced coding skills and is less time consuming, especially if team members lack experience. For example, Figure 5 below shows an example of Parasoft's test unit assistant for C/C++, a wizard that helps with unit test creation.

Another benefit of using a unified testing tool is the feedback it provides about the completeness of testing and the health status of critical business requirements. Low numbers in the MC/DC coverage report may indicate insufficient branch coverage, even if the statement coverage report shows high values. This kind of analysis requires a toolchain that supports multiple coverage metrics, which allows teams to start with something simple, such as line or statement coverage, and proceed towards more thorough code coverage as they are progressing with improving their test cases.

## REQUIREMENTS TO TEST TRACEABILITY

Unit testing and system testing coverage reports are a great source of information about the testing process, especially when combined. However, if test results are not correlated with requirements, teams lack several pieces of critical information.

- Are all the requirements implemented?
- Are all the low-level requirements satisfied?
- What about the 2% of tests that failed?
- What is their impact on the overall quality of the system?
- Are they blocking the release?
- Maybe they affect only deprecated functionalities, having no impact on important use cases?

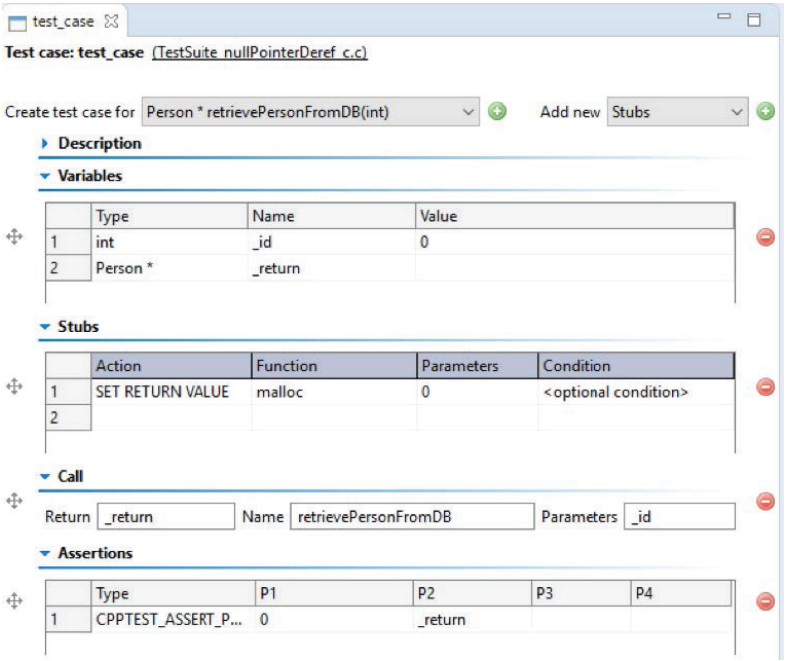
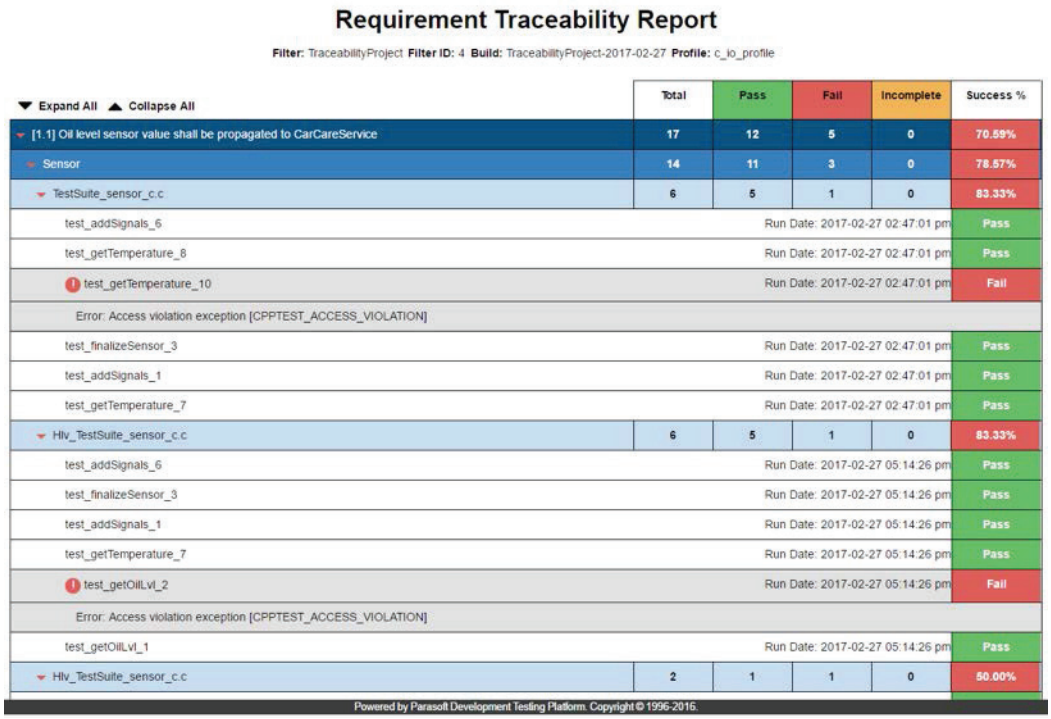



*Figure 5:*
*An example of Parasoft's test unit assistant for C/C++.*

Taking a look at a tests-to-requirements traceability report makes answers to all these questions easy. An example of such a report is shown in below.



**Requirement Traceability Report**

**Filter:** TraceabilityProject **Filter ID:** 4 **Build:** TraceabilityProject-2017-02-27 **Profile:** c_io_profile

| Expand All ▲ Collapse All | Total | Pass | Fail | Incomplete | Success % |
|---|---|---|---|---|---|
| [1.1] Oil level sensor value shall be propagated to CarCareService | 17 | 12 | 5 | 0 | 70.59% |
| Sensor | 14 | 11 | 3 | 0 | 78.57% |
| TestSuite_sensor_c.c | 6 | 5 | 1 | 0 | 83.33% |
| test_addSignals_6 | Run Date: 2017-02-27 02:47:01 pm | | | | Pass |
| test_getTemperature_8 | Run Date: 2017-02-27 02:47:01 pm | | | | Pass |
| test_getTemperature_10 | Run Date: 2017-02-27 02:47:01 pm | | | | Fail |
| Error: Access violation exception [CPPTEST_ACCESS_VIOLATION] | | | | | |
| test_finalizeSensor_3 | Run Date: 2017-02-27 02:47:01 pm | | | | Pass |
| test_addSignals_1 | Run Date: 2017-02-27 02:47:01 pm | | | | Pass |
| test_getTemperature_7 | Run Date: 2017-02-27 02:47:01 pm | | | | Pass |
| Hlv_TestSuite_sensor_c.c | 6 | 5 | 1 | 0 | 83.33% |
| test_addSignals_6 | Run Date: 2017-02-27 05:14:26 pm | | | | Pass |
| test_finalizeSensor_3 | Run Date: 2017-02-27 05:14:26 pm | | | | Pass |
| test_addSignals_1 | Run Date: 2017-02-27 05:14:26 pm | | | | Pass |
| test_getTemperature_7 | Run Date: 2017-02-27 05:14:26 pm | | | | Pass |
| test_getOilLvl_2 | Run Date: 2017-02-27 05:14:26 pm | | | | Fail |
| Error: Access violation exception [CPPTEST_ACCESS_VIOLATION] | | | | | |
| test_getOilLvl_1 | Run Date: 2017-02-27 05:14:26 pm | | | | Pass |
| Hlv_TestSuite_sensor_c.c | 2 | 1 | 1 | 0 | 50.00% |

Powered by Parasoft Development Testing Platform. Copyright © 1996-2016.

*Figure 6:*
*An example of a Parasoft requirements traceability report.*

The ability to correlate requirements with results from different types of testing is a great benefit of using a unified testing tool. Unit testing, system testing, integration testing results, as well as coding standards, and code metrics results can be correlated to provide feedback about the health of critical and non-critical requirements.

## GAUGING CODE COMPLEXITY, PORTABILITY, & MAINTAINABILITY

The complete testing of all requirements and nearly complete code coverage is a certain indication of good progress in a project, however this data provides no information about the future cost of completing, deploying, and maintaining the final product.

- Will the velocity remain the same until the end of the project?
- What will the cost of code maintenance be?
- And the cost of potential code refactoring?

It's almost impossible to answer these questions without relying on code metrics, such as cyclomatic complexity, the average depth of inheritance, the average number of function parameters, and, importantly, the amount of duplicated code. All of these provide valuable information that helps project managers estimate the state of the project's code and the future costs related to its maintenance and modifications.

## EDUCATING DEVELOPERS BY PROVIDING INSTANT FEEDBACK

Keeping code metrics within the desired “green” range, and preventing risky code does not happen by itself. Developers need feedback on the code they are writing, and the sooner they get it, the closer it is to the time they developed it and to make modifications. In this way, instant feedback makes code improvements less expensive, while delaying feedback until nightly builds, or even until continuous integration sessions, increases code improvement costs.

An ideal solution is to integrate a coding standards compliance tool with the IDE to provide feedback interactively, as the code is being written. A unified testing tools should be flexible enough to enable teams to select the coding guidelines they want to enforce, and allow them to install extensions. If teams are able to learn from their own mistakes, they are likely to avoid making them in the future. It’s often the case that implementing automatically enforceable coding guidelines is the most effective way to prevent certain code constructs from reappearing.


## COMBINING RESULTS FROM DIFFERENT TECHNIQUES FOR ADVANCED ANALYTICS

Data from the various testing techniques used in a project, when combined, have a great potential for inferring second level metrics and advanced analytics. A unified testing tool allows teams to look into code from a completely new perspective. A failed unit test case has a different meaning to a developer, depending on whether it occurs in code rated high or low risk by code metrics analysis. If this information is further combined with statistics from source control, and correlated to requirements, so that teams can make better decisions about when and how the code should be corrected.

Another kind of analytics offered by a unified testing tool is changed-based testing, and it has a great potential to improve team productivity. The change-based testing capability of a unified testing tool tries to understand the relation between source code, test cases, and code coverage results to compute the optimal set of test cases for verification and validation of specific code delta. In effect, teams can limit their testing sessions by running only a small subset of tests instead of full regression suites. This focus on testing only what is absolutely required translates to significant savings for any kind of middle or large size projects.

## MINIMIZE TOOL-RELATED COSTS

The sections above have discussed the benefits of using multiple testing activities, which may seem obvious. However, what is not obvious is how to minimize the tool related costs. Although there is no simple and universally applicable answer, following some guidelines helps achieve the goal of minimizing tool usage costs.

Firstly, integrate testing technologies with the existing development process as closely as possible. Most developers are not fond of using more than one tool at the time of writing their code. Testing tools need to be integrated with either the IDE or the build system (sometimes this is equivalent). Otherwise, the distraction caused by switching tools and contexts hampers productivity.

In addition, it's important to unify user interfaces. Even if two different testing technologies are integrated with the same IDE as plugins, if their usage model and user interfaces are completely different, developers may get distracted, and in turn, reluctant to use the testing tools.

Last but not least, the tools should have the ability to exchange information between different elements of the toolchain. If teams decide to use unintegrated tools, it is difficult or even impossible to work out a unified representation of the results that would make them suitable for further analytics.

## PUTTING IT ALL TOGETHER: PARASOFT C/C++TEST WITH A CENTRALIZED REPORTING & ANALYTICS HUB

All of the testing technologies discussed above are available in Parasoft C/C++test, a unified, fully integrated testing solution for C/C++ software development projects. Parasoft C/C++test is available as a plugin to popular IDEs, such as Eclipse and Visual Studio. Close integration with the IDE prevents problems discussed above.

Developers can instantly run coding standards compliance checks or execute unit tests at the time of writing code. QA team members can perform manual test scenarios while the application is monitored for code coverage and runtime errors. Offline analysis, such as flow analysis, can be performed in the continuous integration phase. Server sessions, supported with a convnient command line interface, report analysis results to the centralized reporting and analytics dashboard, which displays aggregated information from the development environment.

Parasoft's extensive reporting capabilities, including the ability to integrate data from third-party tools, send test results to developers' IDEs, and compute advanced analytics, enable users to pinpoint actions at the right time. Continuous feedback provided in each phase of the workflow accelerates agile development, and reduces the cost of achieving compliance with safety standards.

The approach to quality assurance processes and architecture discussed in this paper is not limited to just C/C++projects. Parasoft provides similar solutions for Java, C#, and VB.NET.

## CONCLUSION

Automated software quality activities, such as testing of requirements, collecting code coverage, unit testing, or static analysis, are not new. They are well-known techniques in software testing, but tend to be used in a selective manner. This paper demonstrates that software quality improvement is not a homogeneous endeavor, it requires different technologies to eradicate different types of software defects.

The key challenge is how to do it in an effective and efficient way that avoids devastating the project budget and undermining the morale of developers. A unified testing tool with deep integration into the developer's IDE provides the most productive environment for development testing. A unified tool, such as Parasoft C/C++test, with its ability to unify metrics and results from all aspects of testing, provides additional benefit by allowing teams to focus their testing on high risk and most recently modified code.

## TAKE THE NEXT STEP

Request a custom demo to see how your team can continuously deliver safe and secure code for C/C++ development.

### 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.