Parasoft C/C++ Testing Starter Kit

"C/C++ Error-Detection Techniques" White Paper
Inomed Case Study
NEC Case Study
Bovie Case Study
Parasoft C/C++test Data Sheet

Integrated Error-Detection Techniques: Find More Bugs in Embedded C Software

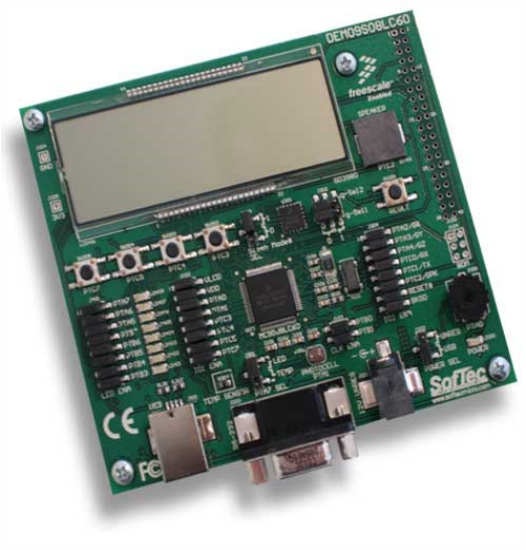
Software verification techniques such as pattern-based static code analysis, runtime memory monitoring, unit testing, and flow analysis are all valuable techniques for finding bugs in embedded C software. On its own, each technique can help you find specific types of errors. However, if you restrict yourself to applying just one or some of these techniques in isolation, you risk having bugs that slip through the cracks. A safer, more effective strategy is to use all of these complementary techniques in concert. This establishes a bulletproof framework that helps you find bugs which are likely to evade specific techniques. It also creates an environment that helps you find functional problems, which can be the most critical and difficult to detect.

This paper will explain how automated techniques such as pattern-based static code analysis, runtime memory monitoring, unit testing, and flow analysis can be used together to find bugs in an embedded C application. These techniques will be demonstrated using Parasoft C++test, an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality.

As you read this paper—and whenever you think about finding bugs—it's important to keep sight of the big picture. Automatically detecting bugs such as memory corruption and deadlocks is undoubtedly a vital activity for any development team. However, the most deadly bugs are functional errors, which often cannot be found automatically. We'll briefly discuss techniques for finding these bugs at the conclusion of this paper.

Introducing the Scenario

To provide a concrete example, we will introduce and demonstrate the recommended bug-finding strategies in the context of a scenario that we recently encountered: a simple sensor application that runs on an ARM board.



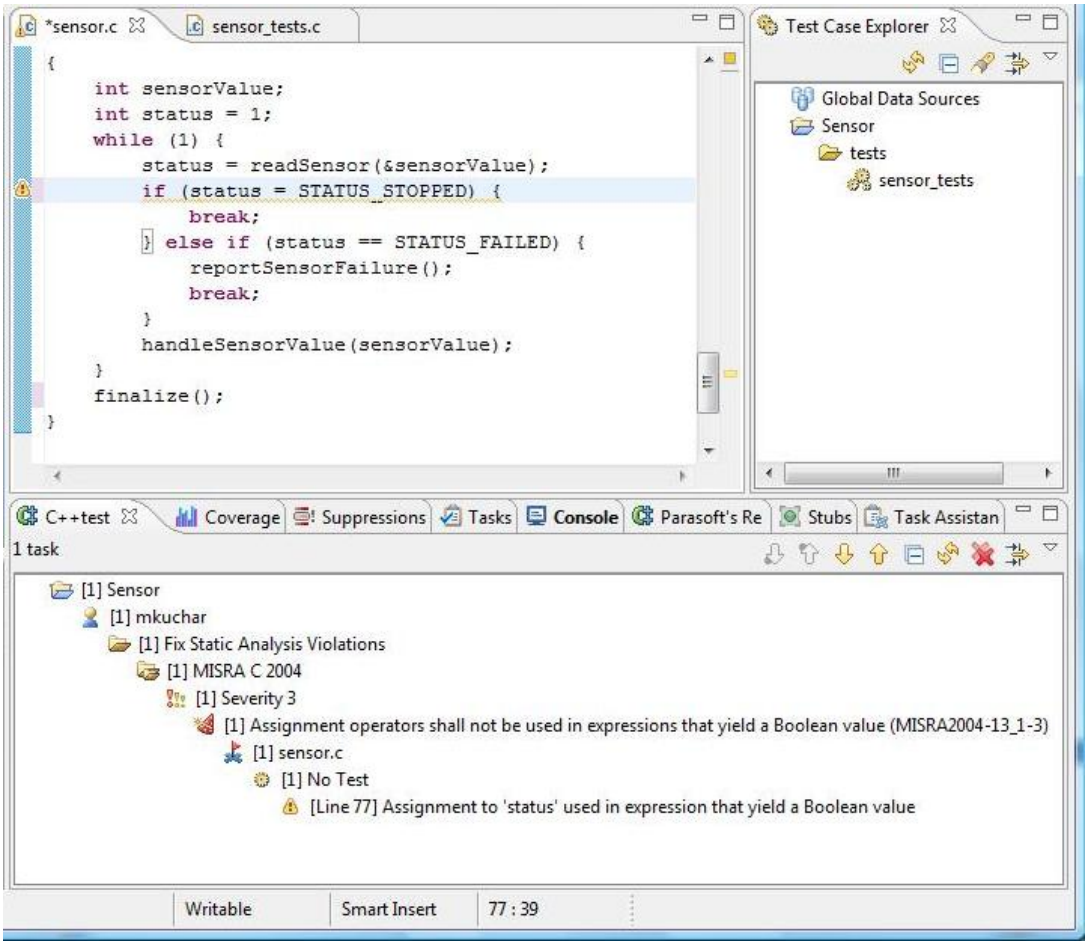
Assume that so far, we have created an application and uploaded it to the board. When we tried to run it, we did not see an expected output on the LCD screen.

It's not working, but we're not sure why. We can try to debug it, but debugging on the target board is time-consuming and tedious. We would need to manually analyze the debugger results and try to determine the real problems on our own. Or, we might apply certain tools or techniques proven to pinpoint errors automatically.

At this point, we can start crossing our fingers as we try to debug the application with the debugger. Or, we can try to apply an automated testing strategy in order to peel errors out of the code. If it's still not working after we try the automated techniques, we can then go to the debugger as a last resort.

Pattern-Based Static Code Analysis

Let's assume that we don't want to take the debugging route unless it's absolutely necessary, so we start by running pattern-based static code analysis. It finds one problem:

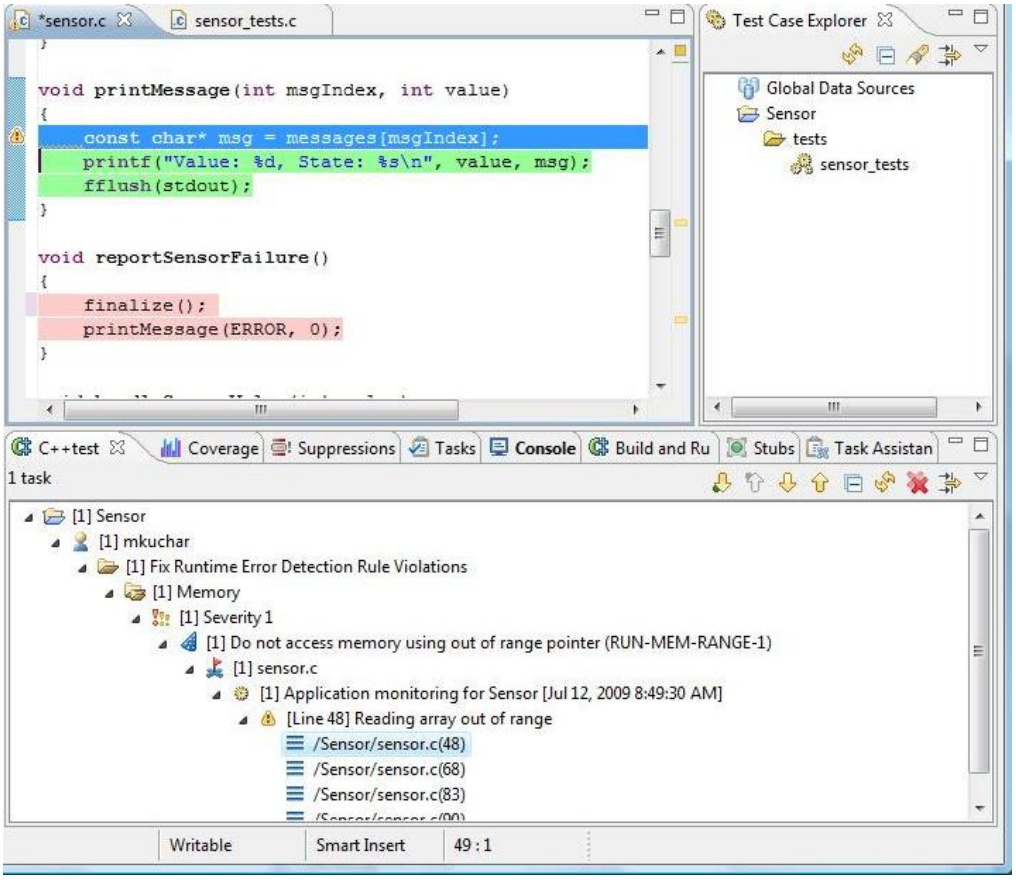


This is a violation of a MISRA rule that says that there is something suspicious with this assignment operator. Indeed, our intention was not to use an assignment operator, but rather a comparison operator. So, we fix this problem and rerun the program.

There is improvement: some output is displayed on the LCD. However, the application crashes with an access violation. Again, we have a choice to make. We could try to use the debugger, or we can continue applying automated error detection techniques. Since we know from experience that automated error detection is very effective at finding memory corruptions such as the one we seem to be experiencing, we decide to try runtime memory monitoring.

Runtime Memory Monitoring of the Complete Application

To perform runtime memory monitoring, we have C++test instrument the application. This instrumentation is so lightweight that it is suitable for running on the target board. After uploading and running the instrumented application, then downloading results, the following error is reported:



This indicates reading an array out of range at line 48. Obviously, the `msgIndex` variable must have had a value that was outside the bounds of the array. If we go up the stack trace, we see that we came here with this print message with a value that was out of range (because we put an improper condition for it before calling function `printMessage() )`. We can fix this by taking away unnecessary conditions `(value <= 20)`.

```
void handleSensorValue(int value)
{
	initialize();
	int index = -1;
	if (value >= 0 && value <= 10) {
		index = VALUE_LOW;
	} else if ((value > 10) && (value <= 20)) {
		index = VALUE_HIGH;
	}
	printMessage(index, value);
}
```

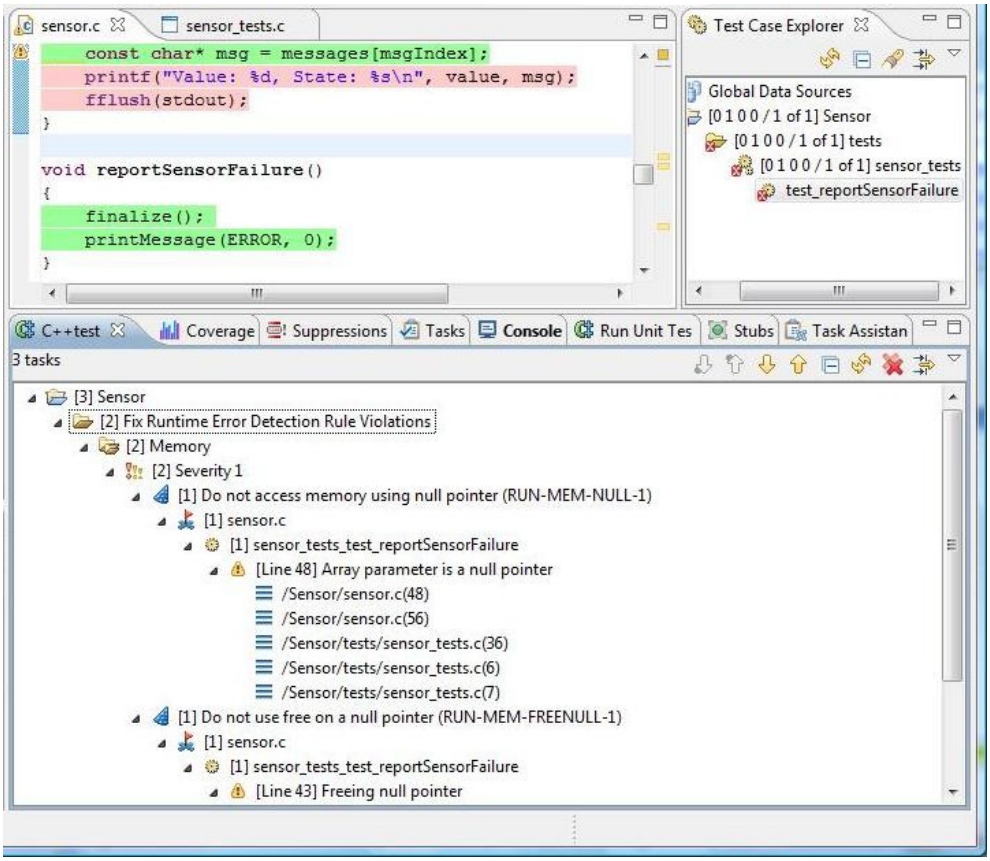
Now, when we rerun the application, no more memory errors are reported. After the application is uploaded to the board, it seems to work as expected. However, we are still a bit worried.

We just found one instance of a memory overwrite in the code paths that we exercised…but how can we rest assured that there are no more memory overwrites in the code that we did not exercise? If we look at the coverage analysis, we see that one of the functions, `reportSensorFailure()`, has not been exercised at all. We need to test this function…but how? One way is to create a unit test that will call this function.

Unit Testing with Runtime Memory Monitoring

We create a test case skeleton using C++test's test case wizard, then we fill in some test code. Then, we run this test case—exercising just this one previously-untested function—with runtime memory monitoring enabled. With C++test, this entire operation takes just seconds.

The results show that the function is now covered, but new errors are reported:



Our test case uncovered more memory-related errors. We have a clear problem with memory initialization (null pointers) when our failure handler is being called. Further analysis leads us to realize that in `reportSensorValue()` we mixed an order of calls. `finalize()` is being called before `printMessage()` is called, but `finalize()` actually frees memory used by `printMessage()`.

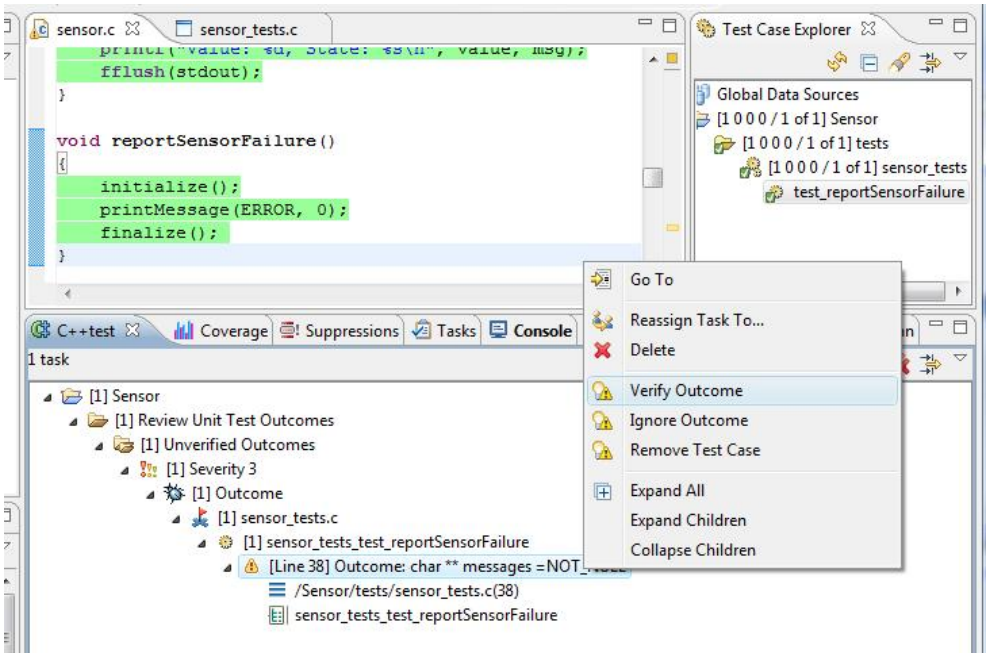
```
void finalize()
{
	if (messages) {
		free(messages[0]);
		free(messages[1]);
		free(messages[2]);
	}
	free(messages);
}
```

We fix this order, then rerun the test case one more time.

That resolves one of the errors reported. Now, let's look at the second problem reported: AccessViolationException in the print message. This occurs because these table messages are not initialized. To resolve this, we call the `initialize()` function before printing the message. The repaired function looks as follows:

```
void reportSensorFailure()
{
	initialize();
	printMessage(ERROR, 0);
	finalize();
}
```

When we rerun the test, only one task is reported: an unvalidated unit test case, which is not really an error. All we need to do here is verify the outcome in order to convert this test into a regression test. C++test will do this for us automatically by creating an appropriate assertion.



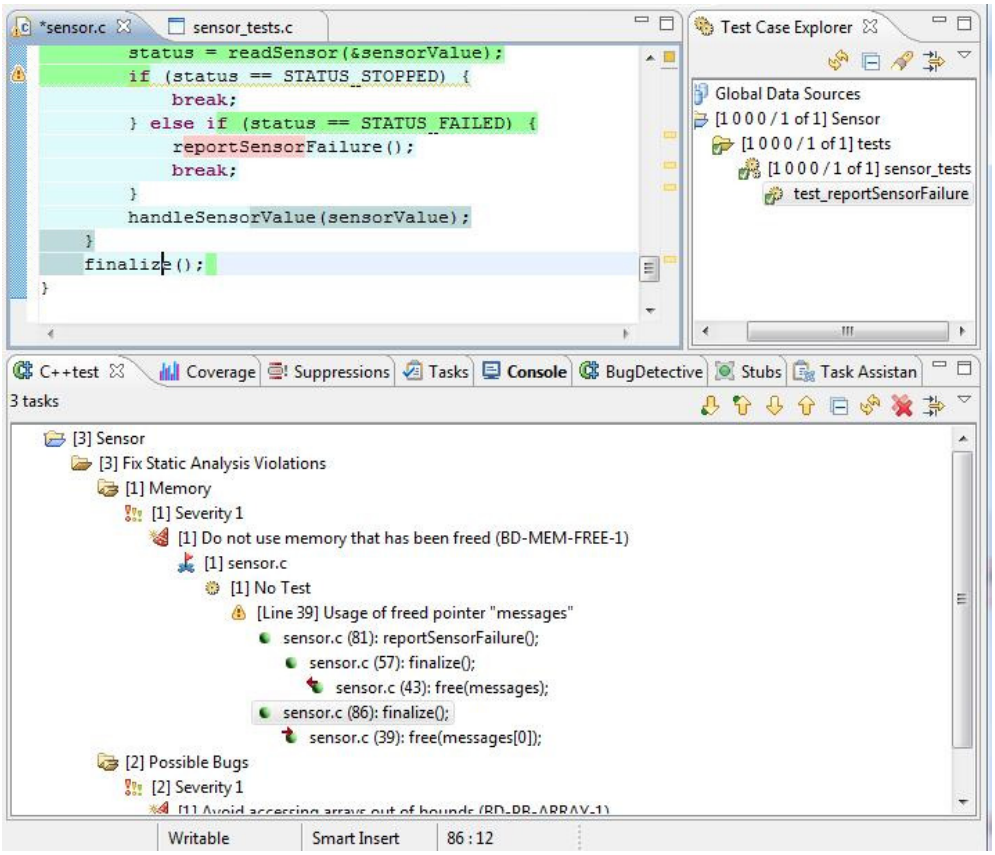
Next, we run the entire application again. The coverage analysis shows that almost the entire application was covered, and the results show that no memory error problems occurred.

Are we done now? Not quite. Even though we ran the entire application and created unit tests for an uncovered function, there are still some paths that are not covered. We can continue with unit

test creation, but it would take some time to cover all of the paths in the application. Or, we can try to simulate those paths with flow analysis.

Flow Analysis

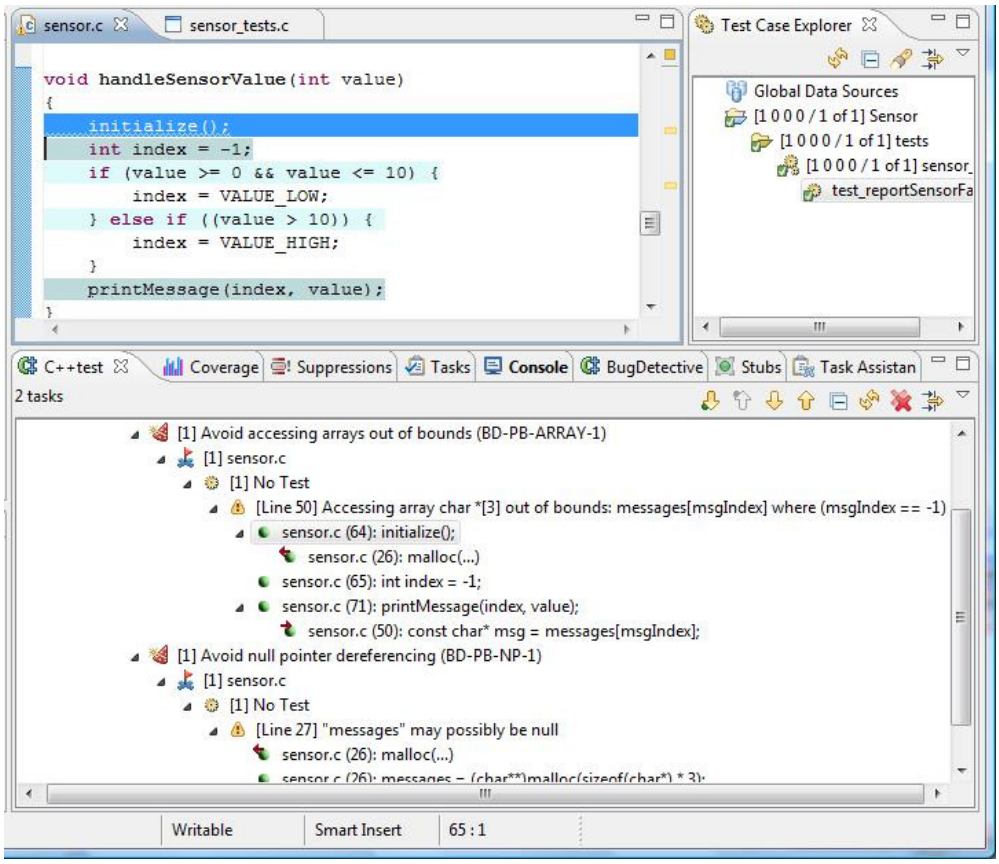
We run flow analysis with C++test's BugDetective, which tries to simulate different paths through the system and check if there are potential problems in those paths. The following issues are reported:



If we look at them, we see that there is a potential path—one that was not covered—where there can be a double free in the `finalize()` function. The `reportSensorValue()` function calls `finalize()`, then the `finalize()` calls `free()`. Also, `finalize()` is called again in the `mainLoop()`. We could fix this by making `finalize()` more intelligent, as shown below:

```
void finalize()
{
	if (messages) {
		free(messages[0]);
		free(messages[1]);
		free(messages[2]);
		free(messages);
		messages = 0;
	}
}
```

Now, let's run flow analysis one more time. Only two problems are reported:



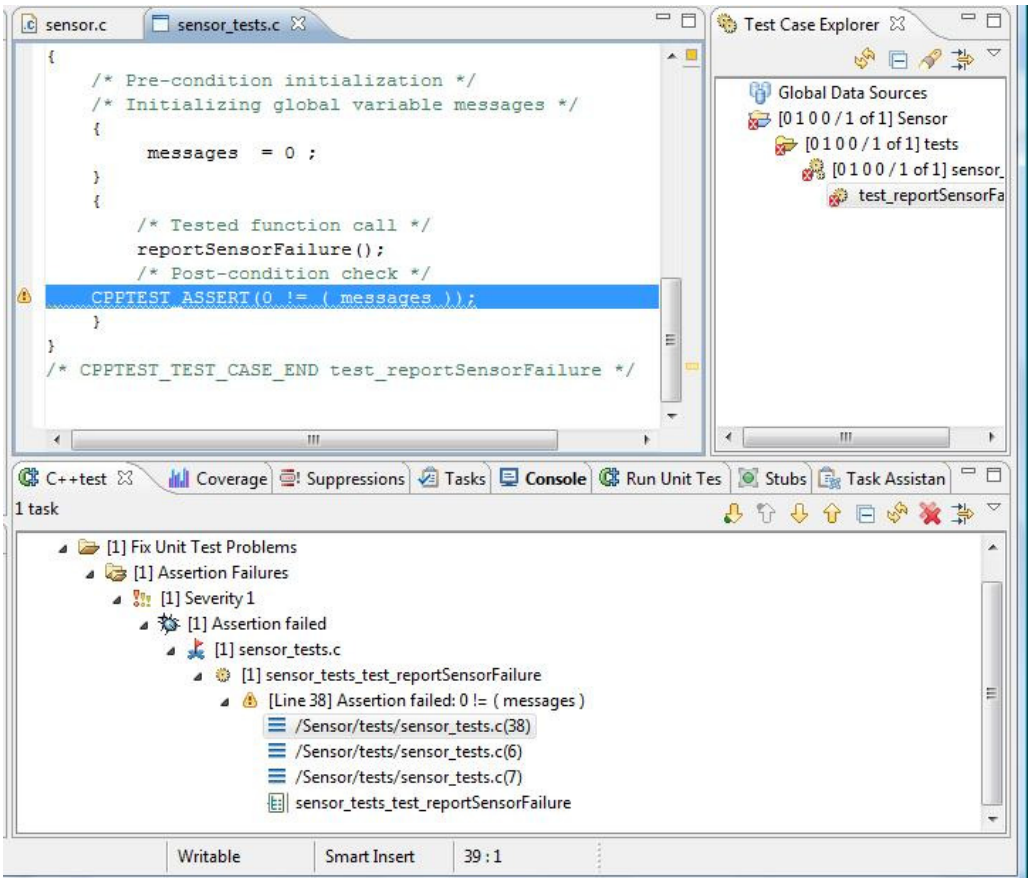
We might be accessing a table with the index -1 here. This is because the integral `index` is set initially to -1 and there is a possible path through the `if` statement that does not set this integral to the correct value before calling `printMessage()`. Runtime analysis did not lead to such a path, and it might be that such path would never be taken in real life. That is the major weakness of static flow analysis in comparison with actual runtime memory monitoring: flow analysis shows potential paths, not necessarily paths that will be taken during actual application execution. Since it's better to be safe than sorry, we fix this potential error easily by removing the unnecessary condition `(value >= 0)`.

```
void handleSensorValue(int value)
{
	initialize();
	int index = -1;
	if (value <= 10) {
		index = VALUE_LOW;
	} else {
		index = VALUE_HIGH;
	}
	printMessage(index, value);
}
```

In a similar way, we fix the final error reported. Now, we rerun the flow analysis, and no more issues are reported.

Regression Testing

To ensure that everything is still working, let's re-run the entire analysis. First, we run the application with runtime memory monitoring, and everything seems fine. Then, we run unit testing with memory monitoring, and a task is reported:



Our unit test detected a change in the behavior of the `reportSensorFailure()` function. This was caused by our modifications in `finalize()`—a change that we made in order to correct one of the previously-reported issues. This task alerts us to that change, and reminds us that we need to review the test case and then either correct the code or update the test case to indicate that this new behavior is actually the expected behavior. After looking at the code, we discover that the latter is true and we update the assertion correct condition.

```
/* CPPTEST_TEST_CASE_BEGIN test_reportSensorFailure */
/* CPPTEST_TEST_CASE_CONTEXT void reportSensorFailure(void) */
void sensor_tests_test_reportSensorFailure()
{
    /* Pre-condition initialization */
    /* Initializing global variable messages */
    {
        messages  = 0 ;
    }
    {
        /* Tested function call */
        reportSensorFailure();
```

```
        /* Post-condition check */
    CPPTEST_ASSERT(0 == ( messages ));
    }
}
/* CPPTEST_TEST_CASE_END test_reportSensorFailure */
```

As a final sanity check, we run the entire application on its own—building it in the IDE without any runtime memory monitoring. The results confirm that it is working as expected.

Wrap Up

To wrap up, let's take a bird's-eye view of the steps we just went over…

We had a problem with our application not running as expected, and we had to decide between two approaches to resolving this: running in the debugger, or applying automated error detection techniques.

If we decided to run code through the debugger, we would have seen strange behavior: some variable always being assigned the same value. We would have had to deduct from this that the problem was actually caused by an assignment operator being used instead of comparison. Static analysis found this logical error for us automatically. This type of error could not have been found by runtime memory analysis because it has nothing to do with memory. It probably would not be found by flow analysis either because flow analysis traverses the execution rather than validate whether conditions are proper.

After we fixed this problem, the application ran, but it still had memory problems. Memory problems are very difficult to see under a debugger; when you are in a debugger, you don't really remember the sizes of memory. Automated tools do, however. So, to find these memory problems, we instrumented the entire application, and ran it with runtime memory analysis. This told us exactly what chunk of memory was being overwritten.

However, upon reviewing the coverage analysis results, we learned that some of the code was not covered while testing on the target. Getting this coverage information was simple since we had it tracked automatically, but if we were using a debugger, we would have had to try to figure out exactly how much of the application we verified. This is typically done by jotting notes down on paper and trying to correlate everything manually.

Once the tool alerted us to this uncovered code, we decided to leverage unit testing to add additional execution coverage to our testing efforts. Indeed, this revealed yet another problem. During normal testing on the target, covering those functions may be almost impossible because they might be hardware error-handling routines—or something else that is only executed under very rare conditions. This can be extremely important for safety critical applications. Imagine that there is a software error in code that should handle a velocity sensor problem in an airplane: instead of flagging a single device as non-working, we have a system corrupt. Creating a unit test case to cover such an execution path is very often the only way to effectively test it.

Next, we cleaned those problems and also created a regression test case by verifying the outcome (as one of the reported tasks guided us to do). Then, we ran flow analysis to penetrate paths that were not executed during testing on the target—even with the unit test. Before this, we had nearly 100% line coverage, but we did not have that level of path coverage. BugDetective uncovered some potential problems. They didn't actually happen and they might have never happened. They would surface only under conditions that were not yet met during actual

execution and might never be met in real life situations. However, there's no guarantee that as the code base evolves, the application won't end up in those paths.

Just to be safe, we fixed the reported problem to eliminate the risk of it ever impacting actual application execution. While modifying the code, we also introduced a regression, which was immediately detected when we re-ran unit testing. Among these automated error detection methods, regression testing is unique in its ability to detect functional changes and verify that code modifications do not introduce functional errors or unexpected side effects. Finally, we fixed the regression, retested the code, and it all seems fine.

As you can see, all of the testing methods we applied—pattern-based static code analysis, memory analysis, unit testing, flow analysis, and regression testing—**are not in competition with one another, but rather complement one another. Used together, they are an amazingly powerful tool that provides an unparalleled level of automated error detection for embedded C software.**

***

In sum, by automatically finding many bugs related to memory and other coding errors, we were able to get the application up and running successfully. However, it's important to remember that the most deadly bugs are actually functional errors: instances where the application is not working according to specification. Unfortunately, these errors are much more difficult to find.

One of the best ways to find to find such bugs is through peer code reviews. With at least one other person inspecting the code and thinking about it in context of the requirements, you gain a very good assessment of whether the code is really doing what it's supposed to.

Another helpful strategy is to create a regression test suite that frames the code, enabling you to verify that it continues to adhere to specification. In the sample scenario described above, unit testing was used to force execution of code that was not covered by application-level runtime memory monitoring: it framed the current functionality of the application, then later, as we modified the code, it alerted us to a functionality problem. In fact, such unit test cases should be created much earlier: ideally, as you are implementing the functionality of your application. This way, you achieve higher coverage and build a much stronger "safety net" for catching critical changes in functionality.

Parasoft C++test assists with both of these tasks: from automating and managing the peer code review workflow, to helping the team establish, continuously run, and maintain an effective regression test suite.

About Parasoft C++test

Parasoft C++test is an integrated solution for automating a broad range of best practices proven to improve software development team productivity and software quality. C++test enables coding policy enforcement, static analysis, runtime memory monitoring, automated peer code review, and unit and component testing to provide teams a practical way to ensure that their C and C++ code works as expected. C++test can be used both on the desktop under common development IDEs, as well as in batch processes via command line interface for regression testing. It also integrates with Parasoft's reporting system, which provides interactive Web-based dashboards with drill-down capability, allowing teams to track project status and trends based on C++test results and other key process metrics.

C++test reduces the time, effort, and cost of testing embedded systems applications by enabling extensive testing on the host and streamlining validation on the target. As code is built on the host, an automated framework enables developers to start testing and improving code quality before the target hardware is available. This significantly reduces the amount of time required for testing on the target. The test suite built on the host can be reused to validate software functionality on the simulator or the actual target.

For more details on C++test, visit Parasoft's C and C++ Testing Tool center.

About Parasoft

For 20 years, Parasoft has investigated how and why software errors are introduced into applications. Our solutions leverage this research to deliver quality as a continuous process throughout the SDLC. This promotes strong code foundations, solid functional components, and robust business processes. Whether you are delivering Service-Oriented Architectures (SOA), evolving legacy systems, or improving quality processes—draw on our expertise and award-winning products to increase productivity and the quality of your business applications. For more information visit http://www.parasoft.com.

Contacting Parasoft

USA

101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Toll Free: (888) 305-0041
Tel: (626) 305-0041
Fax: (626) 305-3036
Email: info@parasoft.com
URL: www.parasoft.com

Europe

France: Tel: +33 (1) 64 89 26 00
UK: Tel: + 44 (0)208 263 6005
Germany: Tel: +49 731 880309-0
Email: info-europe@parasoft.com

Asia

Tel: +886 2 6636-8090
Email: info-psa@parasoft.com

Other Locations

See http://www.parasoft.com/contacts

PARASOFT®
We make software work.™

PARASOFT CASE STUDY

Company: Inomed *www.inomed.com*
Topic: Inomed Streamlines IEC 62304 Medical Device Software Certification

Parasoft WebKing
Parasoft SOAtest
Parasoft Jtest
Parasoft C++test
Parasoft Insure++
Parasoft Concerto
Parasoft dotTEST
Parasoft BPEL Maestro

Inomed Streamlines IEC 62304 Medical Device Software Certification

Inomed (www.inomed.com) is an international medical technology company that develops nerve protection instruments and systems for precisely targeted patient treatments.

Since there is no margin for error in the software driving these innovative devices, Inomed established extensive processes to ensure software integrity. Such full-lifecycle quality processes are essential for complying with IEC 62304, a "harmonized" medical device software safety standard that has been adopted by the European Union and the United States.

With Parasoft's preconfigured Medical Device Compliance solution seamlessly integrated into their existing environment, Inomed was able to rapidly automate their established processes–as well as introduce full requirements traceability. This significantly reduces the work required to achieve and sustain the mandated IEC 62304 certification.

"We have done this [conform to IEC 62304] before using Parasoft tools, but our previous approach was paper-based and we needed much more time for this process.

The Challenge: Streamline Certification for Medical Device Software

Working with doctors and users, Inomed develops new tools and methods in the fields of intraoperative neuromonitoring, neurosurgery, pain therapy and neurological diagnostics. Inomed supplies high-quality products to improve treatment outcomes and uses innovative technologies to ensure safety for both treatment providers and patients.

Jörg Wipfler, Inomed's Head of Development, explains, "Our products are used intraoperative. If any failure occurs during an operation, the operation might have to be aborted. Moreover, since we monitor nerves and their signals, incorrect interpretations and decisions made by our software could lead to wrong decisions by the user...and could cause injuries for the patient."

Since safety is so critical for medical devices, the IEC has recently taken an active role in regulating the software that is developed for medical devices used in Europe. The IEC 62304 standard provides a framework of software life cycle processes with activities and tasks necessary for the safe design and maintenance of medical device software. It enforces traceability and repeatability of the development and maintenance process. The US FDA accepts IEC 62304 compliance as evidence that medical device software has been designed to an acceptable standard.

After establishing a largely manual process for achieving their initial IEC certification, Inomed wanted to automate their risk management processes. Their ultimate goal was to reduce the work involved in sustaining their existing certifications as well as streamline the certification process for their emerging innovations.

continued on page 2

"Using the Parasoft solution significantly increases our efficiency because many manual steps could be automated."

The Solution: Automating Risk Management Processes with Parasoft

Inomed determined that in order to streamline their risk management processes, they needed an integrated system to cover both application lifecycle management (ALM) and the standard's testing requirements for their C++ and .NET language code. After surveying the market, they discovered that Parasoft was a perfect fit for their needs.

Parasoft outfitted Inomed with an integrated Medical Device Compliance solution that included Parasoft Concerto and Parasoft Test (with Parasoft C++test and Parasoft dotTEST). Wipfler explains, "The results from C++test and dotTEST could be used in Concerto…and planned activities from Concerto could be transferred directly into our development environment."

Parasoft's broad range of supported environments enabled easy, rapid integration of the Parasoft solution into Inomed's heterogeneous development environment, which includes Visual Studio, Keil µVision, Bugzilla, and CVS.

Moreover, the solution was deployed to Inomed via a fully pre-configured Virtual Machine— further jumpstarting the adoption process. Immediately after delivery, Inomed could start using the system to validate their software and manage their processes.

The Results: Increased Efficiency through Automation

The Parasoft Test component of the solution allows Inomed to adopt a standardized process for static analysis, code review, and unit testing across their C++, C#, and C++/CLI code. Wipfler appreciates the value of having an integrated, comprehensive solution.

He explains, "We now have a solution doing automated unit tests with the same tool on different development environments and programming languages. Also, the usage of the test products [dotTEST and C++test] is very easy. The automated application of predefined sets of test rules is very useful."

The Parasoft Concerto component of the solution is used to manage projects and document the process. It also correlates requirements with automated and manual tests, source code, and development/testing tasks. The current level of verification for each requirement or task (including task pass/fail status and coverage) can be assessed at any time by back tracing to all associated tests. This full requirements traceability is crucial for IEC compliance.

"We have done this [conform to IEC 62304] before using Parasoft tools," Wipfler says. "But our previous approach was paper-based and we needed much more time for this process. Using the Parasoft solution significantly increases our efficiency because many manual steps could be automated."

After Parasoft's solution was integrated into Inomed's development process, the process was certified by DQS auditors against IEC 62304. "We could show our auditor that we have absolute traceability from every requirement or task to source code and that we could be certain that all work we do is verified," remarks Wipfler. "Having automatically-generated traceability is a huge advantage."

continued on page 3

"[Parasoft provides] absolutely great support. The engineers are very well-qualified and every problem was solved very quickly. Also, we were impressed with their willingness to address company-specific questions and requirements. Parasoft's support is absolutely top-level!"

Why Parasoft?

In summary, Wipfler cites the following reasons for choosing Parasoft:

- Easy adoption and minimal learning curve
- Seamless integration into the existing development process and environment
- Excellent support
- Deep understanding of their requirements and needs

Wipfler was also impressed with Parasoft's commitment to delivering a solution suited to Inomed's specific needs.

Since Inomed used specialized software, Parasoft made an extra effort to fully integrate it with the solution. Moreover, Inomed and Parasoft engineers collaborated to address some unique issues that had to be resolved in order to establish a natural process for the Inomed development team.

He concludes, "[Parasoft provides] absolutely great support. The engineers are very well-qualified and every problem was solved very quickly. Also, we were impressed with their willingness to address company-specific questions and requirements. Parasoft's support is absolutely top-level!"

PARASOFT®
We make software work.™

USA PARASOFT HEADQUARTERS
101 E. Huntington Drive, Monrovia, CA 91016
Phone: (888) 305-0041, Email: info@parasoft.com

FRANCE Phone: (33 1) 64 89 26 00, Email: sales@parasoft-fr.com
GERMANY Phone: +49 731 880309-0, Email: info-de@parasoft.com
POLAND Phone: +48 12 290 91 01, Email: info-pl@parasoft.com
RUSSIA Phone: +7 383 212 5205, Email: info-russia@parasoft.com
UNITED KINGDOM Phone: +44 (0)208 263 6005, Email: sales@parasoft-uk.com
TAIWAN Phone: +886 2 6636 8090, Email: info-psa@parasoft.com

PARASOFT®
We make software work.™

PARASOFT CASE STUDY

Company: NEC Telecom Software Philippines *www.ntsp.com.ph*
Topic: Enforcing internal quality initiatives

Parasoft WebKing
Parasoft SOAtest
Parasoft Jtest
Parasoft C++test
Parasoft Insure++
Parasoft GRS
Parasoft .TEST
Parasoft BPEL Maestro

NEC Telecom Software Philippines Streamlines Internal Quality Initiatives with Parasoft C++test

NEC Telecom Software Philippines (NSP) is located in Manila and is a subsidiary of NEC Corporation of Japan. NSP develops technologies for broadband and mobile communications, as well as IT and network solutions that meet the highest level of customer satisfaction with their high-quality output.

One of NSP's development teams, Virtual Target Solution (VTS) is made up of 15 people. With such a large number of individuals, enforcing internal quality initiatives has been challenging. The majority of people on the development team are young engineers who are just learning NSP's ever-important internal quality initiatives, such as code review, scope, and range of testing.

The VTS team cut both the time and cost of their code reviews. Using Parasoft C++test helps them get the job done within 2 to 3 hours for 8,000 to 10,000 lines of application code.

Until recently, only the two senior level engineers performed reviews of all of the source code for any given project assigned to the team. This source code is typically between 10,000 to 12,000 lines of code.

Working Under Pressure

In December of 2005, the VTS team found themselves working on a project with an extremely tight schedule. It was impossible for the senior developers to code review everything.

As a result, an abundant number of coding errors and potential errors passed through the coding phase. This, in turn, adversely affected unit and integration tests.

Due to the high degree of difficulty that accompanied reviewing all of the source code and verifying all of the check points listed in their draft QA plan, Joel Calderon, software design supervisor for NSP's VTS team, decided to begin searching for an alternative solution to manual code reviews and unit testing.

NSP has a high commitment to quality and Joel was determined to stay true to that commitment. He had a previous, positive experience with Parasoft Jtest, a testing solution for Java, and so decided to pay Parasoft's website a visit. There, he found what he was hoping for: Parasoft C++test, an automated C/C++ unit testing and coding standard analysis solution.

Saving Time with Automated Code Review

NSP's VTS team had two kinds of issues that they searched for during code reviews in their pursuit of high quality:

- General coding violations
- Logical and design errors

continued on page 2

Since finding and solving coding violations took so much time, the VTS team wanted to automate that portion of the code review process so that they could focus their manual code reviews efforts toward finding logical and design errors. With fewer types of issues to search for, the manual code reviews became less prone to insufficient coverage and less time-consuming.

To assist in detecting coding violations, the VTS team used Parasoft C++test RuleWizard. Cherry Ann Alib, software design engineer III, stated, "It's very useful because we can customize our rules with RuleWizard for the code review and coding standards analysis."

Code reviews once performed by more expensive senior-level engineers can now be entrusted to their most junior team members because the task only entails learning how to use the solution.

After putting Parasoft C++test to use, Joel said, "It worked wonders for detecting the coding violations." According to Joel, the VTS team cut both the time and cost of their code reviews. Using Parasoft C++test helps them get the job done within 2 to 3 hours for 8,000 to 10,000 lines of application code.

Previously, the intricate manual code reviews performed by the more expensive senior-level engineers took 10 to 15 hours for the same amount of code. Now, the VTS team can entrust the task of performing automated code reviews to their most junior team members because the task only entails learning how to use the solution.

Prior to Parasoft C++test, performing effective manual code reviews required extensive experience in software development – often in the length of years. Joel pointed out that there are some engineers who never develop the required skills to perform effective manual code reviews. He states, "Parasoft C++test removes this impediment."

NEC Telecom Software Philippines' Virtual Target Solution Team

Automating Unit Testing

For the same tightly scheduled project mentioned earlier, the VTS team had to create more than 500 unit test cases within two weeks. Cherry says of this project, "We were having a hard time creating unit test items manually. It was a tedious task, so we started looking for a tool that would automate the creation of test items."

With Parasoft C++test, the VTS team greatly reduced the amount of development time that they spend writing test cases, as well as the amount of time that they spend performing unit tests and regression tests. Parasoft C++test runs the tests during the night, and then the team gets the results of those tests in the morning.

The reduction in time saved here can also be attributed to the fact that with Parasoft C++test, 75% to 80% of errors were detected during the upstream processes (design phase up to unit test phase), leaving only 20% to 25% left for the downstream processes (integration testing phase to product release).

continued on page 3

" Parasoft C++test has made it easy to transfer knowledge to new people, reducing the negative impact when experienced developers leave and new ones come in to replace them."

Adhering to Quality Initiatives – Efficiently

Joel says that Parasoft C++test enabled the VTS team to organize and streamline their quality initiatives. All of their coding rules and test requirements are concentrated in one location – their Parasoft C++test application – as opposed to spread among different team members' personal knowledge and experience.

Now, everyone has a common mindset, so to speak, when it comes to internal quality initiatives because the team has a single, consistent set of QA parameters as defined by Parasoft C++test's built-in coding rules and user-defined coding rules created by the team via the Coding Rule Wizard.

All team members are trained in Parasoft C++test, so anyone can run it. According to Joel, "Parasoft C++test has made it easy to transfer knowledge to new people, reducing the negative impact when experienced developers leave and new ones come in to replace them."

He goes on to say, "It's easier to teach new people how to merely use the software rather than the concepts of QA and all the things that they need to know in order to adhere to those internal quality initiatives, such as code review, scope, and range of testing. Automation makes it a lot easier for us."

Reducing Overtime and Stress

As a result of Joel's decision to check out Parasoft's website for a testing solution to meet his team's needs, the VTS team was able to meet their client's deadline, which they never thought they would do on such a tight schedule. What's more, the VTS team was able to win two additional projects from this same client. One is in the works now.

Beyond the success and high-quality that NSP's client's see, the VTS team is more productive in less time. Before using Parasoft C++test, their average working hours ranged from 12 to 15 hours per day. Now, their norm is right at 8 hours. Joel says, "We barely have overtime work now.

"For example, for unit testing and even code review, we just leave it with the machine performing Parasoft C++test, and then we get results in the morning. Now, projects are much less stressful.

"Parasoft C++test enhances the quality of our products and the quality of our lives."

PARASOFT®
We make software work.™

USA PARASOFT HEADQUARTERS
101 E. Huntington Drive, Monrovia, CA 91016
Phone: (888) 305-0041, Email: info@parasoft.com

FRANCE Phone: (33 1) 64 89 26 00, Email: sales@parasoft-fr.com
GERMANY Phone: +49 89 461 3323-0, Email: info-de@parasoft.com
POLAND Phone: +48 12 259 1550 ext. 204, Email: info-pl@parasoft.com
RUSSIA Phone: +7 383 212 5205, Email: info-russia@parasoft.com
UNITED KINGDOM Phone: +44 (0)1923 858005, Email: sales@parasoft-uk.com
TAIWAN Phone: +886 2 6636 8090, Email: info-psa@parasoft.com

PARASOFT®
We make software work.™

PARASOFT CASE STUDY

Company: Bovie Medical *www.boviemedical.com*

Topic: Bovie Medical successfully moves V&V testing in-house with Parasoft Embedded solutions

Parasoft WebKing
Parasoft SOAtest
Parasoft Jtest
Parasoft C++test
Parasoft Insure++
Parasoft Concerto
Parasoft .TEST
Parasoft BPEL Maestro

Bovie Medical Cuts Costs and Time by Automating Testing with Parasoft Embedded

Bovie Medical Corporation (AMEX: BVX) is a manufacturer and marketer of electrosurgical products. They set the standard for surgi-center and hospital-based electrosurgical generators and accessories.

With an entire line of electrosurgical devices, Bovie Medical has been manufacturing ESUs (electrosurgical units) for about eight years.

Recently, Bovie Medical decided to introduce a generator into the market that hospitals could interact with via a software user interface. The interface is an LCD screen that enables doctors to view pertinent patient medical information while performing surgery.

Over the past 10 years in the medical industry, the FDA has taken an extremely active role in regulating the software that's developed for medical devices. There are strict FDA regulations and requirements that must be met before devices can be sold to hospitals and put to use by doctors so that no harm comes to patients.

With help from Parasoft Embedded solutions, Bovie Medical was able to move verification and validation (V&V) testing in-house for their medical devices with embedded softwarecutting these costs approximately in half.

Parasoft Embedded solutions also empowered Bovie Medical to develop a comprehensive and controllable V&V process.

Weighing OptionsVendor versus In-house V&V

Part of meeting FDA requirements and regulations includes performing clinical trials and providing documented proof that their medical devices are safe for patients. Before clinical trials can begin, Bovie Medical performs extensive software testing.

Previously, Bovie Medical turned to a vendor for their embedded software testing. When it came time to begin development of their latest project, management presented the Software Development and Quality Assurance (QA) departments with a few choices for testing.

Option 1—Returning to the Vendor

The first option was to return to the vendor and have V&V performed outside Bovie Medical. However, the development team saw several drawbacks to this choice.

Although the vendor did a good job of breaking down the 45 to 50 thousand lines of code and analyzing it, they did not find all of the bugs.

continued on page 2

Being a general testing company, the vendor did have the advantage of automated tools to perform static analysis, code reviews, and even system testing. However, this was outweighed by a significant disadvantage: the vendor lacked understanding of the products being tested.

The Bovie Medical development team knows their system, what they're developing, and what the end user will do with it. They know the industry and are best equipped to test what they develop themselves.

Gary Malfa, Software Engineer at Bovie Medical, specifies some other disadvantages of returning to the vendor. "We would have to give them all of our proprietary source code.

"Plus, we would still have to spend a lot of time with a company like thatexplaining how to interact with our software from a user point of view. They would have to understand how our software works before they could test it. Based on the previous project they did for us, handling all of their questions was like a full-time customer support job."

"One of the major reasons why we went with Parasoft is because we needed to do unit testing with two different languages: C++ and C#,"

Of course, the greatest concern was the cost. When this vendor tested Bovie's previous release, it cost $300,000. Going forward with this choice would mean going back to the same company and paying them near that amount again to perform another round of V&V.

Option 2—Working with Consultants

A second option was to hire a couple of independent consultants to come in and perform the V&V. The challenge here was finding qualified people to do the job. Bovie Medical could not seem to find a consultant who was a true expert in V&V.

Beyond that challenge, this option entailed putting heavy resources toward training the consultants as they simultaneously performed the V&V work at handa major drawback.

Option 3—Implementing Automated Testing In-house

The last option combined hiring another QA tester with purchasing automated testing software so that the development team could perform all V&V in-house.

From a capital expenditure point of view, this option was the most cost-effective of the three options. In fact, according to Gary Pickett, CFO, going this route would save Bovie Medical roughly 50% of its validation costs for the release at hand.

The Bovie QA department found support and encouragement to pursue this option after they attended an FDA seminar. A team member asked, "Is it admissible for a developer to test his own code using automated testing tools for static and dynamic testing?"

The representative replied, "We think automated testing is on the cutting edge of software validation in the medical arena. We are encouraging people to do that."

With that, the QA and development teams concluded that it was worth searching for a way to bring the software validation in-house and cut costs.

After finding and talking with several different companies, Bovie Medical decided that Parasoft Embedded solutions were the best fit for its needs.

Why Parasoft?

"One of the major reasons why we went with Parasoft is because we needed to do unit testing with two different languages: C++ and C#," Malfa explains.

He goes on, "We found about four other companies that have automated software testing products for C++, but none had testing products for C#. Parasoft was the only company that provided quality solutions for both."

continued on page 3

"We are able to get our product to market approximately 6 months sooner with Parasoft Embedded solutions than we could have if we had gone back to the testing vendor."

Malfa went on to say that it just made sense to go with one company for both languagesto ease the cost and simplify product support. (Parasoft C++test supports C++, while Parasoft .TEST supports C#.)

Malfa stated, "Licensing was friendlier at Parasoft than at other companies." He was also impressed by Parasoft's excellent customer service.

Meeting FDA Guidelines—No Sweat!

Bovie Medical's machines do not hook up to an internal network within the hospital. The machines are not client server applications. As mentioned earlier, Bovie Medical's medical devices are stand-alone and use embedded software programs.

Doctors use these embedded software medical devices in their operating rooms. They look at the screen to view essential medical information of the patient on whom they are performing surgery.

Because the medical devices require specialized embedded software programs, Bovie Medical uses a custom compiler that was not previously supported by Parasoft C++test . Malfa says, "The Parasoft development and support teams did a tremendous job of customizing Parasoft C++test to meet our embedded needs."

To test a C++ embedded project, Bovie Medical developers load a Parasoft C++test executable based on the actual C++ embedded project directly onto the embedded target device, and then run unit tests directly on the hardware that's going to be running in the hospital environment.

Data is fed into each individual unit. After unit tests are finished running, the results are retrieved and uploaded to the developer's PCinto Parasoft C++test.

All of that data, along with documentation, is stored in Bovie Medical's testing archive. In the event that they get audited by the FDA, that vital information is just a click away.

Finding and Removing Hidden Logic Issues—Early

Both Parasoft C++test and Parasoft .TEST enable Bovie Medical developers to run static analysis on their source code before moving onto dynamic testing. The static analysis verifies that all established language rules are not violated.

Malfa says, "It teaches our development team better coding habits so that we can make our source code more fault-tolerantmore robust."

Beyond that, Parasoft Embedded solutions enable Bovie Medical's development team to find bugs and design flaws before projects go into production.

Malfa explains, "I'm not talking about obvious bugs that software developers can find without automated testing tools; I'm talking about difficult-to-find logic issues that can take many hours of manual unit and integration testing to discover."

Malfa discusses a memory leak that Parasoft C++test found in his code. He says, "This was the type of bug that could not be found by doing all of the human testing in the world. The software could be released and run fine for ten years. Then, one day out of the blue, it would run in a hospital with a certain sequence and certain data, then fail."

He goes on to say that his team could investigate an error like that for months and never be able to reproduce it. He states, "Only automated products like Parasoft's C++test and .TEST could find it."

continued on page 4

"The Parasoft development and support teams did a tremendous job of customizing Parasoft C++test to meet our embedded needs."

Getting to Market Sooner

Malfa is emphatic that Parasoft Embedded solutions have saved Bovie Medical an immense amount of time. "If we had to manually do code reviews and manually run all of our unit tests, we'd have to hire and train a dedicated staff of people to get it done in the same amount of time."

He also talks about the alternative of going back to that testing vendor. He states, "We are able to get our product to market approximately 6 months sooner with Parasoft Embedded solutions than we could have if we had gone back to the testing vendor."

He goes on, "We found about four other companies that have automated software testing products for C++, but none had testing products for C#. Parasoft was the only company that provided quality solutions for both."

Malfa went on to say that it just made sense to go with one company for both languagesto ease the cost and simplify product support. (Parasoft C++test supports C++, while Parasoft .TEST supports C#.)

Malfa stated, "Licensing was friendlier at Parasoft than at other companies." He was also impressed by Parasoft's excellent customer service.

Leaping Forward with Automation

In closing, Malfa states, "From what we researched, I think Parasoft is on the leading edge of automated testing. Developing software without using an automated software testing solution that finds bugs and design flaws during the development cycle is like developing software in the dark ages. Not only is it expensive, but it also increases the chances of embarrassment by a customer finding a bug.

"I think Parasoft is on the leading edge of automated testing."

"With Parasoft Embedded solutions, we find and fix software issues in the development stage long before the QA department starts testing the code. Developing software products with an automated testing solution process in place is a quantum leap forward for the software industry".

"If anyone I know is looking for automated software solutions and wants to start increasing productivity in their software testing, I would definitely tell them to evaluate Parasoft quality solutions. I would highly recommend Parasoftnot only for their solutions, but also, for their customer service, which exceeded our expectations."

PARASOFT®
We make software work.™

USA PARASOFT HEADQUARTERS
101 E. Huntington Drive, Monrovia, CA 91016
Phone: (888) 305-0041, Email: info@parasoft.com

FRANCE Phone: (33 1) 64 89 26 00, Email: sales@parasoft-fr.com
GERMANY Phone: +49 731 880309-0, Email: info-de@parasoft.com
POLAND Phone: +48 12 290 91 01, Email: info-pl@parasoft.com
RUSSIA Phone: +7 383 212 5205, Email: info-russia@parasoft.com
UNITED KINGDOM Phone: +44 (0)208 263 6005, Email: sales@parasoft-uk.com
TAIWAN Phone: +886 2 6636 8090, Email: info-psa@parasoft.com

PARASOFT®

C++test™

C/C++ Development

DATA SHEET

Parasoft® C++test™ – Comprehensive Code Quality Tools for C/C++ Development

Parasoft C++test enables teams to produce better code, test it more efficiently, and consistently monitor progress towards their quality goals. With C++test, critical time-proven best practices–such as static analysis, comprehensive code review, runtime error detection, unit and component testing with integrated coverage analysis–are automated on the developer's desktop, early in the development cycle. A command line interface enables fully automated execution within regression and continuous integration environments, providing data for monitoring and analyzing quality trends. Moreover, C++test integrates with Parasoft's Concerto, which provides interactive Web-based dashboards with drill-down capability. This allows teams to track project status and trends based on C++test results and other key process metrics.

For embedded and cross-platform development, C++test can be used in both host-based and target-based code analysis and test flows. See page 3 for details.

Automate Code Analysis for Monitoring Compliance

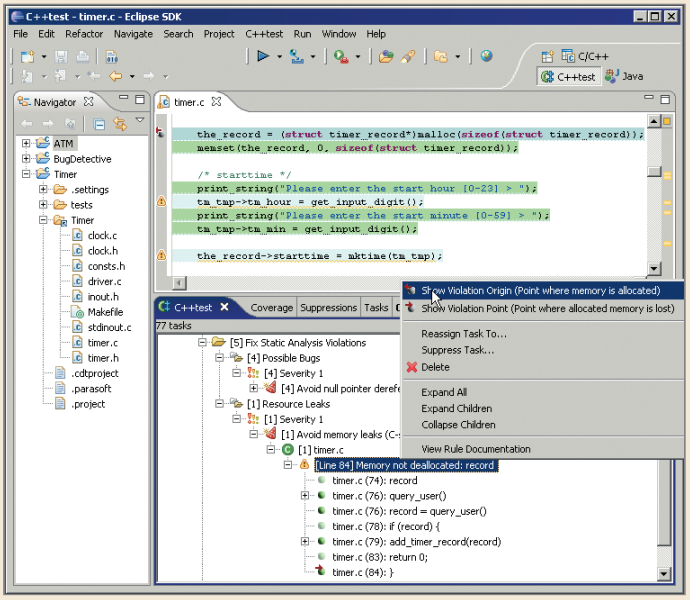
A properly implemented coding policy can eliminate entire classes of programming errors by establishing preventive coding conventions. C++test statically analyzes code to check compliance with such a policy. To configure C++test to enforce a coding standards policy specific to their group or organization, users can define their own rule sets with built-in and custom rules. Code analysis reports can be generated in a variety of formats, including HTML and PDF.

Hundreds of built-in rules–including implementations of MISRA, MISRA 2004, and the new MISRA C++ standards, as well as guidelines from Meyers' Effective C++ and Effective STL books, and other popular sources–help identify potential bugs from improper C/C++ language usage, enforce best coding practices, and improve code maintainability and reusability. Custom rules, which are created with a graphical RuleWizard editor, can enforce standard API usage and prevent the recurrence of application-specific defects after a single instance has been found.

Identify Runtime Bugs without Executing Software

BugDetective, the advanced interprocedural static analysis module of C++test, simulates feasible application execution paths–which may cross multiple functions and files–and determines whether these paths could trigger specific categories of runtime bugs. Defects detected include using uninitialized or invalid memory, null pointer dereferencing, array and buffer overflows, division by zero, memory and resource leaks, and various flavors of dead code.

C++test greatly simplifies defect analysis by providing a complete path trace for each potential defect in the developer's IDE. Automatic cross-links to code help users quickly jump to any point in the highlighted analysis path.



C++test's static analysis identifies critical bugs without executing the code (Eclipse version shown)

Benefits

- **Increase team development productivity** – Apply a comprehensive set of best practices that reduce testing time, testing effort, and the number of defects that reach QA.
- **Achieve more with existing development resources** – Automatically vet known coding issues so more time can be dedicated to tasks that require human intelligence.
- **Build on the code base with confidence** – Efficiently construct, continuously execute, and maintain a comprehensive regression test suite that detects whether updates break existing functionality.
- **Gain instant visibility into C and C++ code quality and readiness** – Access on-demand objective code assessments and track progress towards quality and schedule targets.
- **Reduce support costs** – Automate negative testing on a broad range of potential user paths to uncover problems that might otherwise surface only in "real-world" usage.

DATA SHEET

A multi-metric test coverage analyzer, including statement, branch, path, and MC/DC coverage, helps users gauge the efficacy and completeness of the tests, as well as demonstrate compliance with test and validation requirements, such as DO-178B. Test coverage is presented via code highlighting for all supported coverage metrics–in the GUI or color-coded code listing reports. Summary coverage reports including file, class, and function data can be produced in a variety of formats.

Automated Regression Testing

C++test facilitates the development of a robust regression test suite that detects if incremental code changes break existing functionality. Whether teams have a large legacy code base, a small piece of just-completed code, or something in between, C++test can generate tests that capture the existing software behavior via test assertions produced by automatically recording the runtime test results. As the code base evolves, C++test reruns these tests and compares the current results with those from the originally captured "golden set." It can easily be configured to use different execution settings, test cases, and stubs to support testing in different contexts (e.g., different continuous integration phases, testing incomplete systems, or testing specific parts of complete systems). This type of regression testing is especially critical for supporting agile development and short release cycles, and ensures the continued functionality of constantly evolving and difficult-to-test applications.

Configurable Detailed Reporting

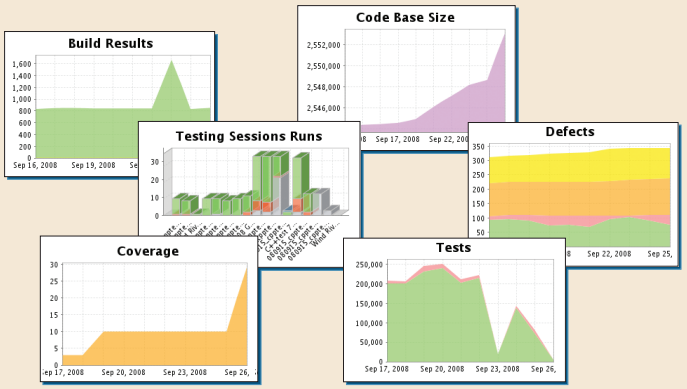
C++test's HTML, PDF, and custom format reports can be configured via GUI controls or an options file. The standard reports include a pass/fail summary of code analysis and test results, a list of analyzed files, and a code coverage summary. The reports can be customized to include a listing of active static analysis checks, expanded test output with pass/fail status of individual tests, parameters of trend graphs for key metrics, and full code listings with color-coding of all code coverage results. Generated reports can be automatically sent via email, based on a variety of role-based filters. In addition to providing data directly to the developers responsible for the code flagged for defects, C++test sends summary reports to managers and team leads.

Efficient Team Deployment

C++test establishes an efficient process that ensures software verification tasks are ingrained into the team's existing workflow and automated–enabling the team to focus on tasks that truly require human intelligence. Defect review and correction are facilitated through automated task assignment and distribution. Each defect detected is prioritized, assigned to the developer who wrote the related code, and distributed to his or her IDE with full data and cross-links to code. To help managers assess and document trends, centralized reporting ensures real-time visibility into quality status and processes. This data also helps determine if additional actions are needed to satisfy internal goals or demonstrate regulatory compliance.

Advanced Unit Test Features

- Automatic generation of tests and stubs
- Automatic generation of assertions based on observed test results
- Graphical Test Case Wizard for interactive definition of tests
- Complete visibility into test and stub source code
- Intelligent, test-case-sensitive stubs
- Parameterization of tests and stubs
- Multi-metric coverage analysis for DO-178B (including MC/DC)
- Flexible support for continuous regression testing
- Annotation of tests against bug and requirement IDs
- Execution of tests under debugger
- Special mode for testing template code



Dashboards track key development metrics

Embedded and Cross-Platform Development

As the software components in embedded systems are becoming increasingly critical, the attention to quality in embedded software increases across the board. Long-standing quality strategies such as testing with a debugger are no longer efficient or sufficient. To further complicate matters, many developers cannot readily run a test program in the actual deployment environment because they lack access to the final system hardware. To address these challenges, code quality needs to be realized throughout the development lifecycle–using a synergy of time-proven techniques for early defect prevention, assisted by automation for implementation and monitoring.

For highly quality-sensitive industries, such as avionics, medical, automobile, transportation, and industrial automation, the addition of Parasoft's Web-based audit and reporting system, with interactive Web-based dashboards and drill-down capability powered by a SQL database, enables an efficient and auditable quality process with complete visibility into compliance efforts.

Features

- Static analysis of code for compliance with user-selected coding standards
- Graphical RuleWizard editor for creating custom coding rules
- Static code path simulation for identifying potential runtime errors
- Automated code review with a graphical interface and progress tracking
- Application monitoring/memory analysis
- Automated generation and execution of unit and component-level tests
- Flexible stub framework
- Full support for regression testing
- Code coverage analysis with code highlighting
- Code coverage analysis for unit testing and beyond (including application-level tests)
- Full team deployment infrastructure for desktop and command line usage

Runtime Error Detection

- Identify complex memory-related problems through simple functional testing–for example:
  - memory leaks
  - null pointers
  - uninitialized memory
  - buffers overflows
- Collect code coverage from application runs
- Increase test result accuracy through execution of the monitored application in a real target environment

Streamline Code Review

Code review is known to be the most effective approach to uncover code defects. Unfortunately, many organizations underutilize code review because of the extensive effort it is thought to require. The C++test Code Review module automates preparation, notification, and tracking of peer code reviews, enabling a very efficient team-oriented process. Status of all code reviews, including all comments by reviewers, is maintained and automatically distributed by the C++test infrastructure. C++test supports two typical code review flows:

- **Post-commit code review.** This mode is based on automatic identification of code changes in a source repository via custom source control interfaces, and creating code review tasks based on pre-set mapping of changed code to reviewers.
- **Pre-commit code review.** Users can initiate a code review from the desktop by selecting a set of files to distribute for the review, or automatically identify all locally changed source code.

The effectiveness of team code reviews is further enhanced through C++test's static analysis capability. The need for line-by-line inspections is virtually eliminated because the team's coding policy is monitored automatically. By the time code is submitted for review, violations have already been identified and cleaned. Reviews can then focus on examining algorithms, reviewing design, and searching for subtle errors that automatic tools cannot detect.

Monitor the Application for Memory Problems

Runtime Error Detection is the best known approach to eliminating serious memory-related bugs with zero false positives. The running application is constantly monitored for certain classes of problems–like memory leaks, null pointers, uninitialized memory, and buffer overflows–and results are visible immediately after the testing session is finished.

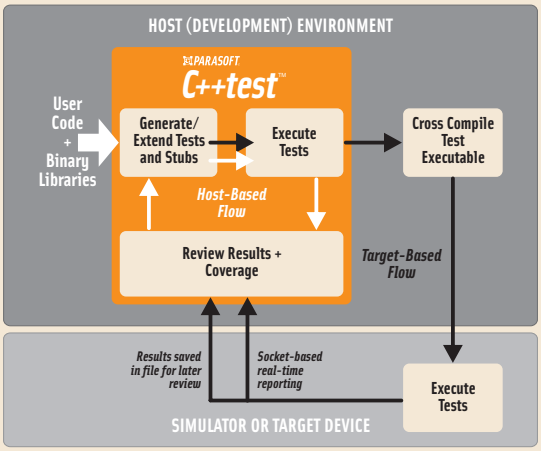
Without requiring advanced and time-consuming testing activities, the prepared application goes through the standard functional testing and all existing problems are flagged. The application can be executed on the target device, simulated target, or host machine. The collected problems are presented directly in the developer's IDE with the details required to understand and fix the problem (including memory block size, array index, allocation/deallocation stack trace etc.)

Coverage metrics are collected during application execution. These can be used to see what part of the application was tested and to fine tune the set of regression unit tests (complementary to functional testing).

Unit and Integration Test with Coverage Analysis

C++test's automation greatly increases the efficiency of testing the correctness and reliability of newly-developed or legacy code. C++test automatically generates complete tests, including test drivers and test cases for individual functions, purely in C or C++ code in a format similar to CppUnit. These tests, with or without modifications, are used for initial validation of the functional behavior of the code. By using corner case conditions, these automatically-generated test cases also check function responses to unexpected inputs, exposing potential reliability problems.

Test creation and management is simplified via a set of specific GUI widgets. A graphical Test Case Wizard enables developers to rapidly create black-box functional tests for selected functions without having to worry about their inner workings or embedded data dependencies. A Data Source Wizard helps parameterize test cases and stubs–enabling increased test scope and coverage with minimal effort. Stub analysis and generation is facilitated by the Stub View, which presents all functions used in the code and allows users to create stubs for any functions not available in the test scope–or to alter existing functions for specific test purposes. Test execution and analysis are centralized in the Test Case Explorer, which consolidates all existing project tests and provides a clear pass/fail status. These capabilities are especially helpful for supporting automated continuous integration and testing as well as "test as you go" development.



C++test's customizable workflow allows users to test code as it's developed, then use the same tests to validate functionality/reliability in target environments

Test on the Host, Simulator, and Target

C++test automates the complete test execution flow, including test case generation, cross-compilation, deployment, execution, and loading results (including coverage metrics) back into the GUI. Testing can be driven interactively from the GUI or from the command line for automated test execution, as well as batch regression testing. In the interactive mode, users can run tests individually or in selected groups for easy debugging or validation. For batch execution, tests can be grouped based either on the user code they are liked with, or their name or location on disk.

High Degree of Customization

C++test allows full customization of its test execution sequence. In addition to using the built-in test automation, users can incorporate custom test scripts and shell commands to fit the tool into their specific build and test environment.

C++test can be utilized with a wide variety of embedded OS and architectures, by cross-compiling the provided runtime library for a desired target runtime environment. All test artifacts of C++test are source code, and therefore completely portable.

Supported Host Environments

Host Platforms
- Windows NT/2000/XP/2003/Vista/7
- Linux kernel 2.4 or higher with glibc 2.3 or higher and an x86-compatible processor
- Linux kernel 2.6 or higher with glibc 2.3 or higher and an x86_64-compatible processor
- Solaris 7, 8, 9, 10 and an UltraSPARC processor
- IBM AIX 5.3 and a PowerPC processor

IDEs
- Eclipse IDE for C/C++ Developers 3.2, 3.3, 3.4, 3.5, 3.6, 3.7
- Microsoft Visual Studio 2003, 2005, 2008, 2010 with Visual C++
- Wind River Workbench 2.6 or 3.0-3.3
- ARM Workbench IDE for RVDS 3.0, 3.1, 4.0, 4.1
- QNX Momentics IDE 4.5 (QNX Software Development Platform 6.4)
- Texas Instruments Code Composer Studio v4, v5

IDEs with Project Import Support
- ARM ADS 1.2
- Green Hills MULTI 4.0.x
- IAR Embedded Workbench 5.3/5.4
- Keil RealView MDK 3.40/uVision3
- Microsoft eMbedded Visual C++ 4.0
- Microsoft Visual Studio 6
- Texas Instruments Code Composer 3.1 and 3.3
- Wind River Tornado 2.0, 2.2

Host Compilers
- Windows
  - Microsoft Visual C++ 6.0, .NET (7.0), .NET 2003 (7.1), 2005 (8.0), 2008 (9.0), 2010 (10.0)
  - GNU and MingW gcc/g++ 2.95.x, 3.2.x, 3.3.x, 3.4.x
  - GNU gcc/g++ 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
  - Green Hills MULTI for Windows x86 Native v4.0.x
- Linux (x86 target platform)
  - GNU gcc/g++ 2.95.x, 3.2.x, 3.3.x, 3.4.x, 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
- Linux (x86_64 target platform)
  - GNU gcc/g++ 3.3.x, 3.4.x, 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
- Solaris
  - Sun C++ 5.3 (Sun Forte C++ 6 Update 2), Sun C++ 5.5 (Sun ONE Studio 8), Sun C++ 5.6 (Sun ONE Studio 9), Sun C++ 5.7 (Sun ONE Studio 10), Sun C++ 5.8 (Sun ONE Studio 11), Sun C++ 5.9 (Sun ONE Studio 12)
  - GNU gcc/g++ 2.95.x, 3.2.x, 3.3.x, 3.4.x, 4.0.x, 4.1.x, 4.2.x, 4.3.x, 4.4.x, 4.5.x
  - Green Hills MULTI for SPARC Solaris Native v4.0.x
- AIX
  - IBM XL C/C++ compiler 8.0
  - GNU gcc/g++ 4.1.x

Target/Cross Compilers
- Altera (Linux hosted)
  - Nios GCC 2.9 (static analysis only)
  - NIOS II 5.1 GCC 3.4 (static analysis only)
- ARM (Windows hosted)
  - ARM RVCT 2.2, 3.x, 4.x
  - ARM ADS 1.2
- Cosmic (Windows hosted)
  - Cosmic Software 68HC08 C Cross Compiler V4.6.x (static analysis only)
- eCosCentric (Linux hosted)
  - GCC 3.4.x (static analysis only)
- Fujitsu (Windows hosted)
  - FR Family SOFTUNE C/C++ Compiler V6
- GNU (Windows, Linux, Solaris hosted)
  - gcc 2.9 - 4.5
- Green Hills
  - Windows hosted
    - Green Hills MULTI v5.1.x optimizing compilers for Embedded V800
  - Windows, Solaris hosted
    - Green Hills MULTI v4.0.x optimizing compilers
- IAR (Windows hosted)
  - IAR ANSI C/C++ Compiler 5.3x, 5.4x, 5.5x for ARM
  - IAR ANSI C/C++ Compiler 6.1x for ARM (C only)
- Keil (Windows hosted)
  - ARM/Thumb C/C++ Compiler, RVCT3.1 for uVision
  - ARM C/C++ Compiler, RVCT4.0 for uVision
  - C51 Compiler V8.18 (static analysis only)
- Microsoft (Windows hosted)
  - Microsoft Visual C++ for Windows Mobile 8.0, 9.0
  - Microsoft Embedded Visual C++ 4.0
- QNX (Windows hosted)
  - GCC 2.9.x, 3.3.x, 4.2.x, 4.4.x
- Renesas (Windows hosted)
  - Renesas SH SERIES C/C++ Compiler V9.03
- STMicroelectronics (Windows hosted)
  - ST20 (static analysis only)
  - ST40 (static analysis only)
- TASKING
  - Windows and Solaris hosted
    - 80C196 C Compiler v6.0 (static analysis only)
  - Windows hosted
    - TriCore VX-toolset C/C++ Compiler 2.5 (C only)
    - TriCore VX-toolset C/C++ Compiler 2.5, 3.3, 3.4, 3.5
- Texas Instruments (Windows hosted)
  - Windows hosted - CCS 5.x
    - TMS320C6x C/C++ Compiler v7.3
    - TMS320C2000 C/C++ Compiler v6.0
    - MSP430 C/C++ Compiler v4.0
  - Windows hosted - CCS 4.x:
    - TMS320C6x C/C++ Compiler v6.1.x
    - TMS320C2000 C/C++ Compiler v5.2.x
    - TMS320C55x C/C++ Compiler v4.3
    - TMS320C54x C/C++ Compiler v4.2 (static analysis only)
    - MSP430 C/C++ Compiler v3.2.x
  - Windows hosted - CCS 3.x:
    - TMS320C6x C/C++ Compiler v5.1
    - TMS320C6x C/C++ Compiler v6.0
    - TMS320C2000 C/C++ Compiler v4.1 (static analysis only)
  - Solaris hosted:
    - TMS320C2x/C2xx/C5x Version 7.00 (static analysis only)
    - TMS320C6x C/C++ Compiler v. 4.3 (static analysis only)
    - TMS320C6x C Compiler v. 4.00 (static analysis only)
    - TMS320C6x C/C++ Compiler v. 5.1 (static analysis only)
- Wind River
  - Windows, Solaris, Linux hosted
    - GCC 2.96, 3.4.x, 4.1.x, 4.3.x
    - DIAB 5.0, 5.5, 5.6, 5.7, 5.8, 5.9
  - Windows hosted
    - GCC 3.3.x for VxWorks 653 (static analysis only)
    - EGCS 2.90

Build Management
- GNU make
- Sun make
- Microsoft nmake
- JAM
- Other build scripts that can provide an option of overriding a compiler via an environment variable

Source Control
- AccuRev SCM
- Borland StarTeam
- CVS
- Git
- IBM/Rational ClearCase
- Microsoft Team Foundation Server
- Microsoft Visual SourceSafe
- Perforce SCM
- Serena Dimensions
- Subversion (SVN)
- Telelogic Synergy

www.parasoft.com

Contact info:
Parasoft Corporation, 101 E. Huntington Dr., 2nd Flr., Monrovia, CA 91016
Ph: (888) 305.0041, Fax: (626) 256.6884, Email: info@parasoft.com