Combining Error-Detection Techniques to Find Bugs in Embedded C Software

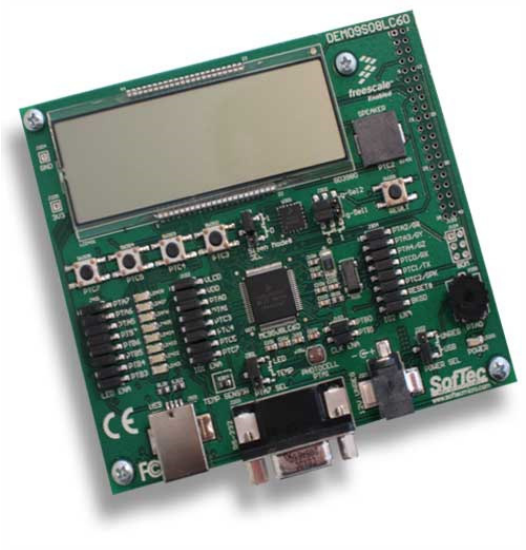
Software verification techniques such as pattern-based static code analysis, runtime memory monitoring, unit testing, and flow analysis are all valuable techniques for finding bugs in embedded C software. On its own, each technique can help you find specific types of errors. However, if you restrict 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.



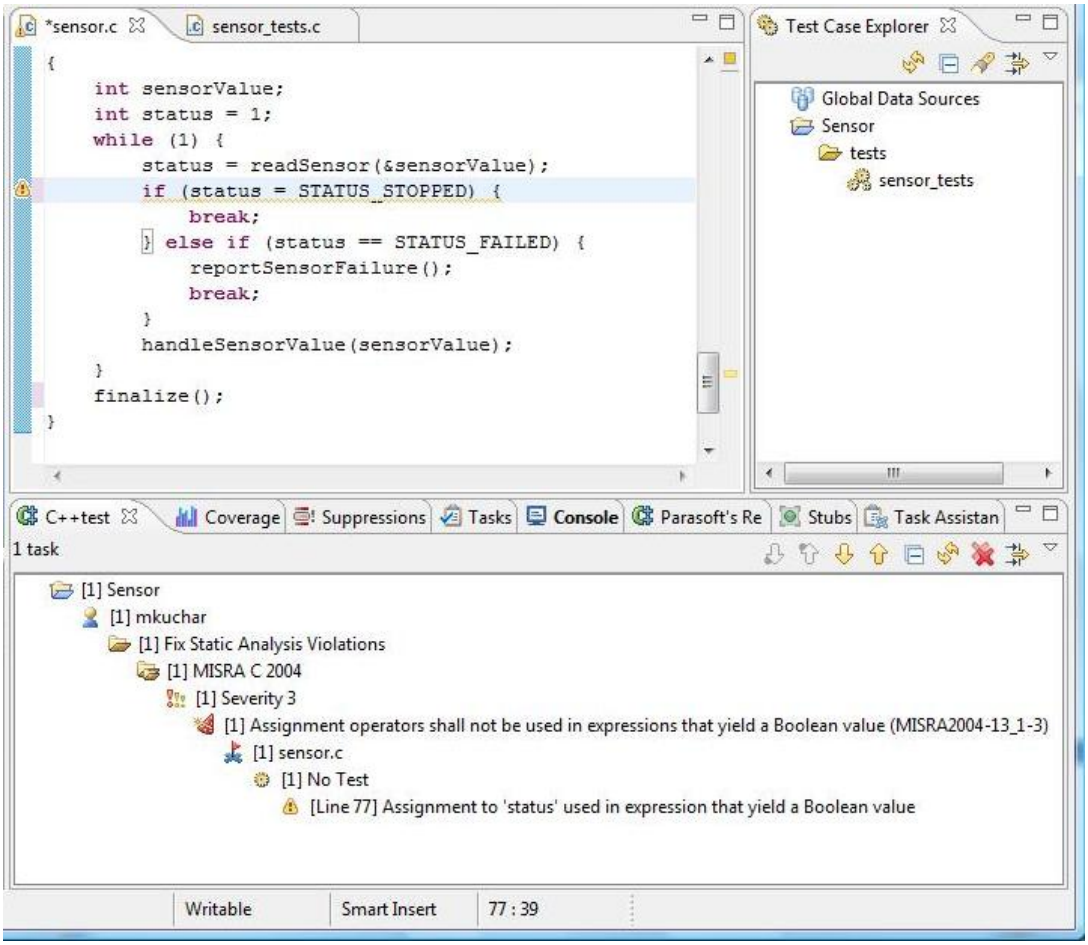
Assume that so far, we have created an application and uploaded it to the board. When we tried to run it, we did not see an expected output on the LCD screen.

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:

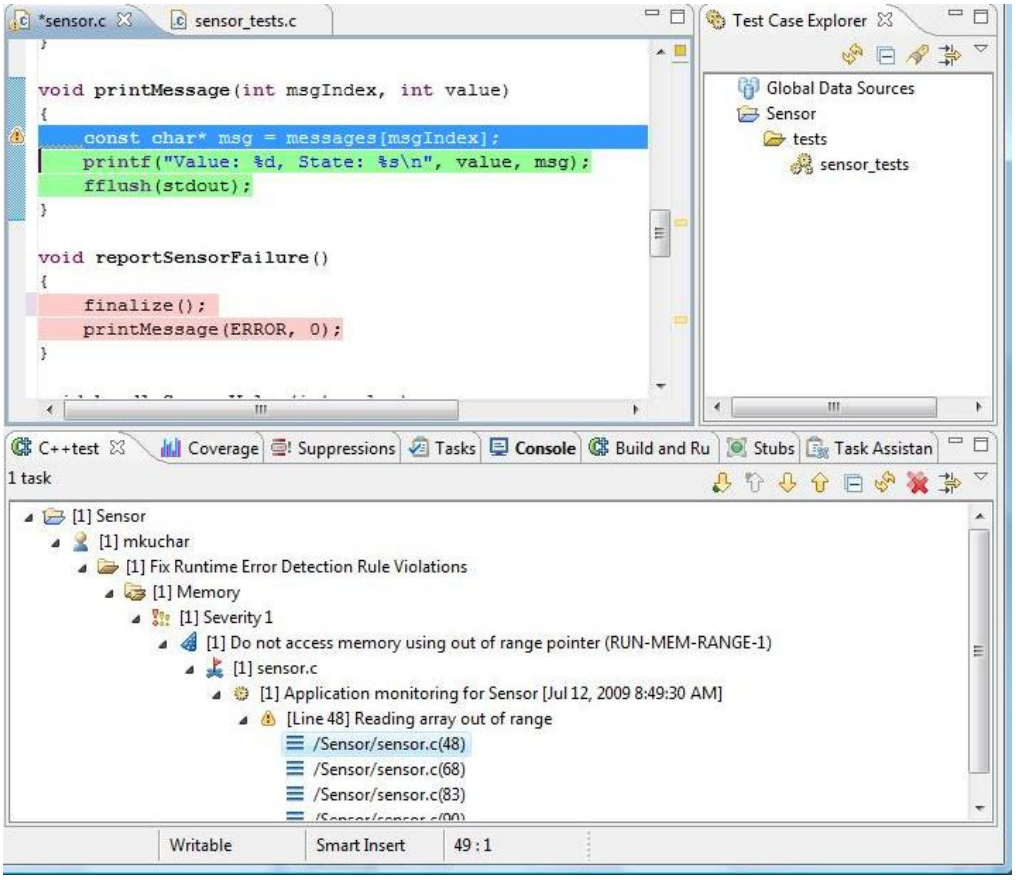


This is a violation of a MISRA rule that says that there is something suspicious with this assignment operator. Indeed, our intention was not to use an assignment operator, but rather a comparison operator. So, we fix this problem and rerun the program.

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);
}
```

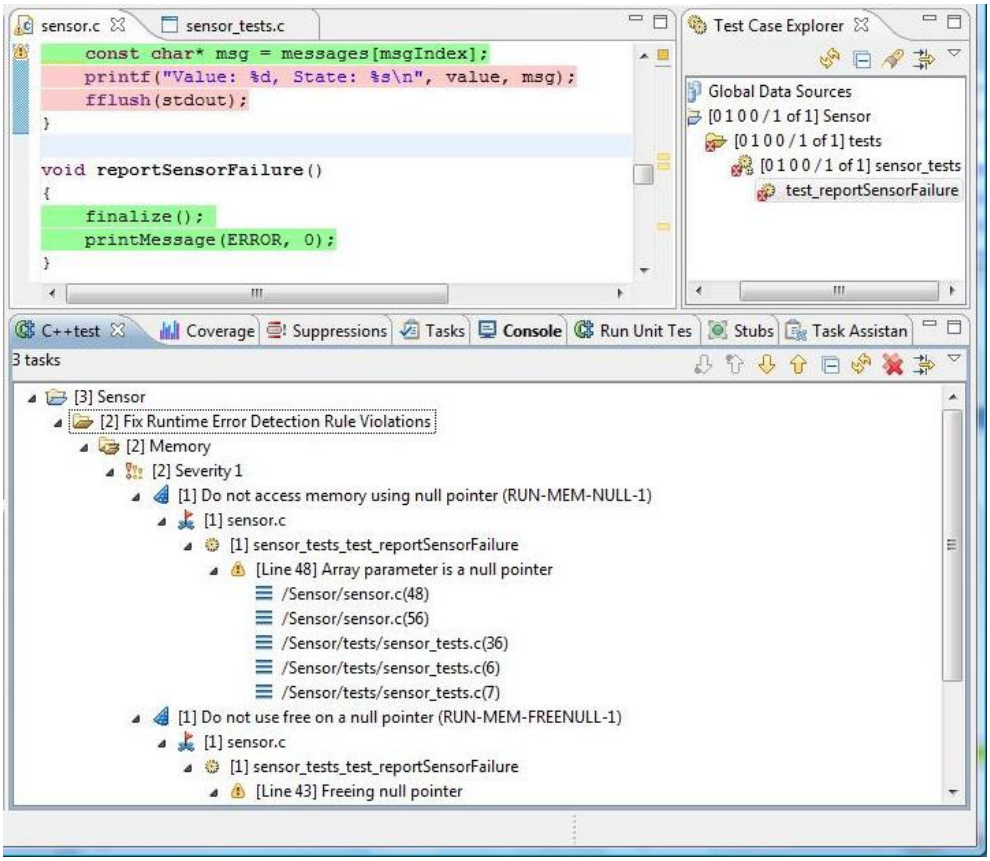
Now, when we rerun the application, no more memory errors are reported. After the application is uploaded to the board, it seems to work as expected. However, we are still a bit worried.

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()`.

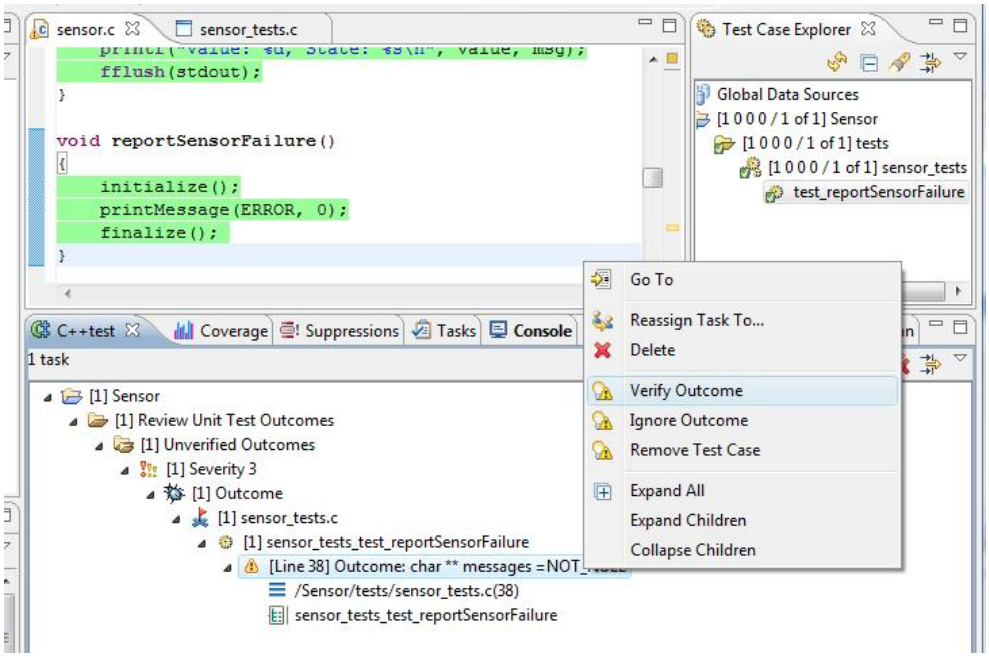
```
void finalize()
{
	if (messages) {
		free(messages[0]);
		free(messages[1]);
		free(messages[2]);
	}
	free(messages);
}
```

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.



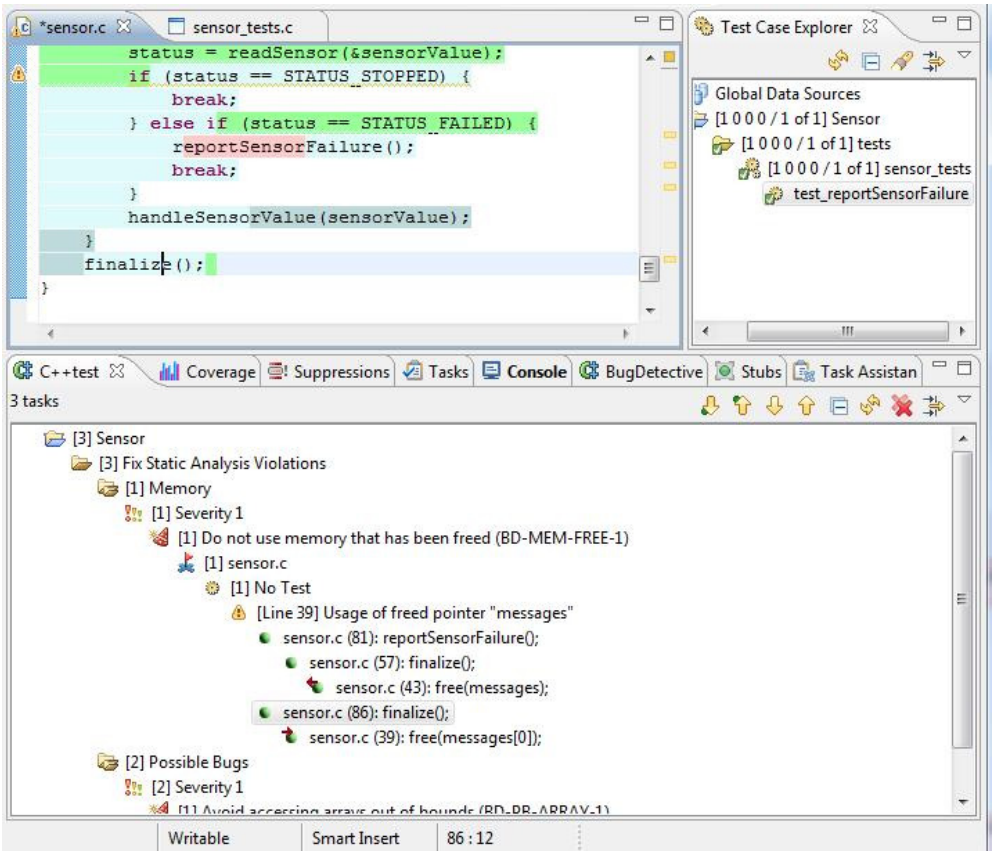
Next, we run the entire application again. The coverage analysis shows that almost the entire application was covered, and the results show that no memory error problems occurred.

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

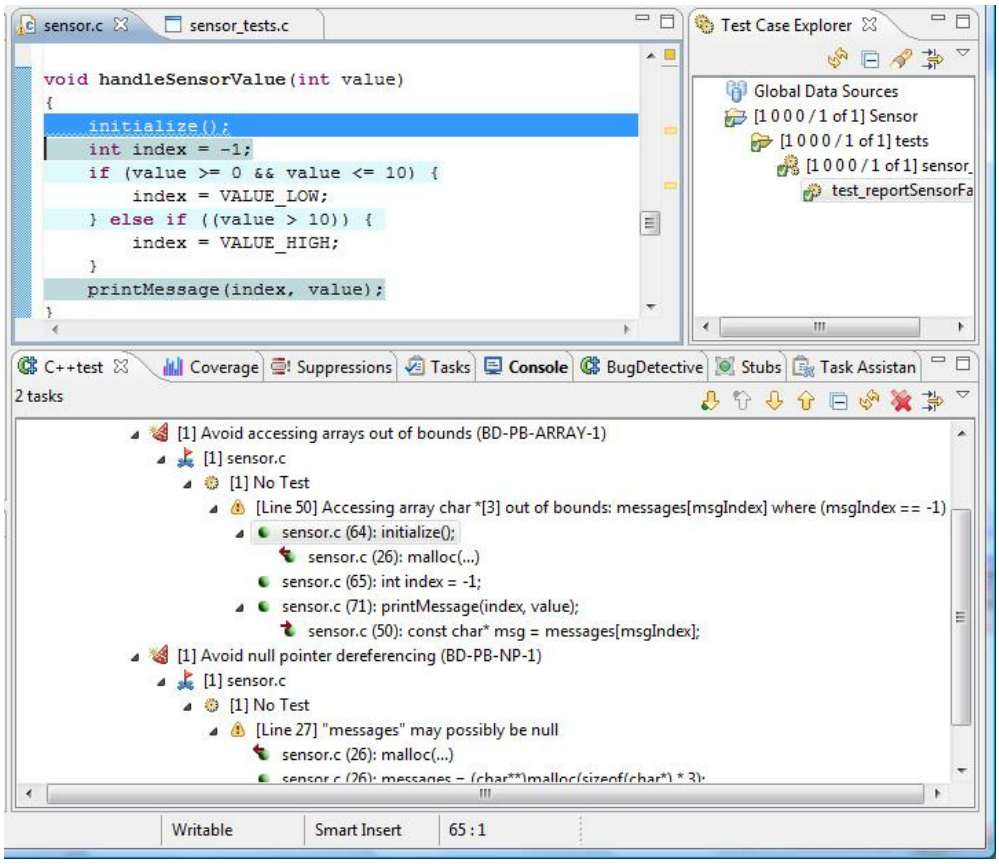
We run flow analysis with C++test's BugDetective, which tries to simulate different paths through the system and check if there are potential problems in those paths. The following issues are reported:



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:



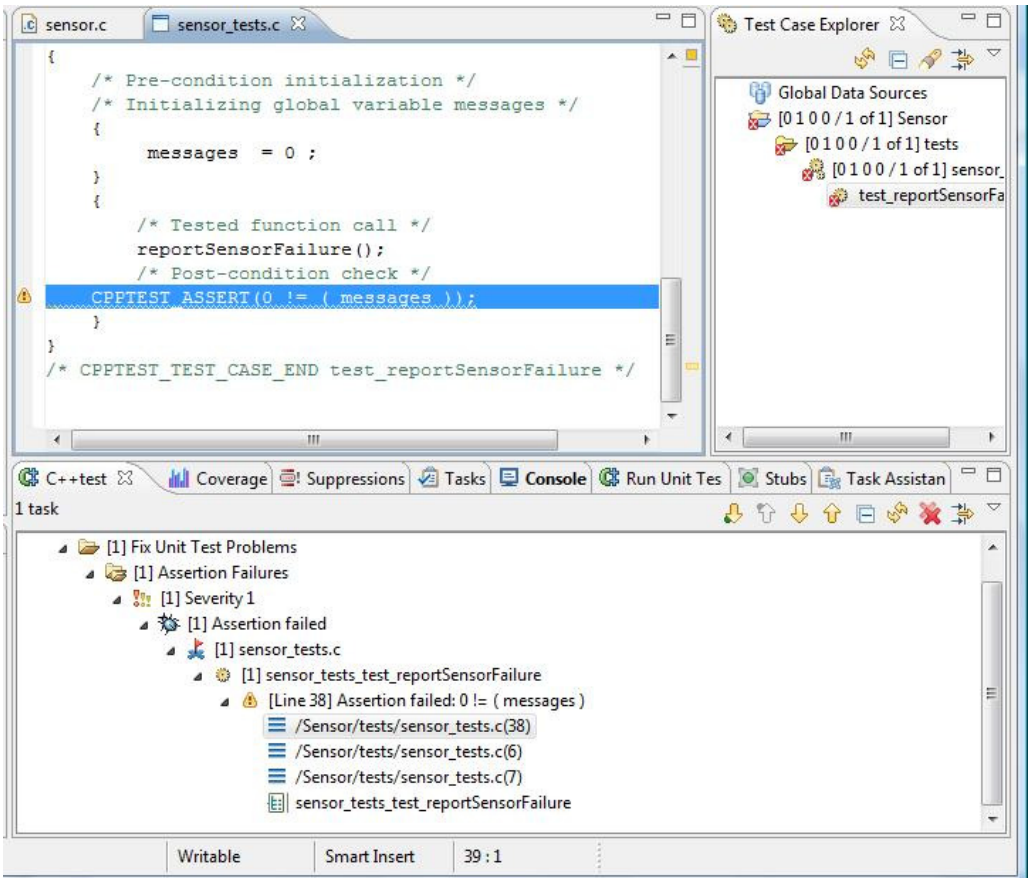
We might be accessing a table with the index -1 here. This is because the integral `index` is set initially to -1 and there is a possible path through the `if` statement that does not set this integral to the correct value before calling `printMessage()`. Runtime analysis did not lead to such a path, 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