

Embedded Technology[®]

Board-Level Electronics & COTS Solutions for Design Engineers

Static vs. Dynamic Detection of Bugs in Safety-Critical Code

In the never-ending quest to produce high-quality software, traditional dynamic testing plays a fundamental role. The weakness of dynamic testing is that it is only as good as the test cases. To be effective, a great deal of effort must go into writing or generating good test cases, and doing so can be very expensive.

Recently, a new breed of static analysis tools has emerged that can find flaws without writing any test cases. These tools, which are also referred to as static testing tools, can find bugs that are difficult or impossible to find using standard testing methodologies. They can locate serious flaws such as buffer overruns, null pointer dereferences, resource leaks, and race conditions. Because they operate by analyzing the source code itself in detail, they can also highlight inconsistencies or contradictions in the code such as unreachable code, useless assignments, and redundant conditions. Such issues often indicate programmer confusion, and correlate well with bugs. Moreover, knowledge of these issues can actually make writing test cases easier.

There is no substitute for rigorous testing at many levels: unit, regression, functionality and integration testing are all essential, but testing effectiveness depends on the quality of the test cases used. The best test suites are those that have good code coverage. Statement coverage and condition coverage are the most commonly used metrics. Full condition coverage is considered essential for safety-critical code, such as flight-control software. Achieving full coverage can be exceedingly time-consuming and expensive. Also, as all programmers know, just because a statement is executed in a successful test case does not mean it will always execute correctly. It may fail under an unusual combination of circumstances that the test cases did not explore.

This is where the strength of static analysis shines through. Static analysis examines paths and considers conditions and variables in the abstract. As such, it can achieve much higher coverage than is usually feasible with dynamic testing. The sooner a bug can be found, the cheaper it is to fix.

Static analysis can be used as soon as the code compiles, so it can find flaws before the program is even complete. This is usually much less expensive than writing a test case or debugging a crash.

Figure 1 shows an example of a simple flaw that was found in an open-source program.

In this example, the flaw will only show up in cases of low memory availability, and only when the first allocation succeeds and the subsequent one fails. This is precisely the kind of unusual combination of circumstances that it is hard to reproduce with traditional testing, but static analysis is good at finding.

Advanced Static Analysis

These new advanced static analysis tools have their roots in theoretical computer science techniques such as model checking and abstract interpretation. This makes them very different from traditional static analysis tools such as *lint*. Although lint-like tools can be used for finding some classes of superficial syntactic flaws and stylistic discrepancies, they can be painful to con-

```

referenced by sbcset 3567
3568     sbcset = (re_bitset_ptr_t) calloc (sizeof (bitset_t), 1); /* Leak (ID: 41) */
3569     #ifdef RE_ENABLE_I18N
3570         mbcset = (re_charset_t *) calloc (sizeof (re_charset_t), 1);
3571     #endif /* RE_ENABLE_I18N */
3572
3573     #ifdef RE_ENABLE_I18N
3574         if (BE (sbcset == NULL || mbcset == NULL, 0))
3575             #else /* not RE_ENABLE_I18N */
3576                 if (BE (sbcset == NULL, 0))
3577             #endif /* not RE_ENABLE_I18N */
3578             {
3579                 *err = REG_ESPACE;
3580                 return NULL;
3581             }
3582
Generated on Thu Sep 6 14:27:27 2007 by GrammaTech CodeSonar 2.1p1 which was built on May 26 2007 05:10:53

```

Figure 1. Fragment of a report warning of a potential memory leak. The memory allocated on line 3568 may be leaked when the function returns on line 3580. Code in red is on the path to the point where the leak occurs. This shows that the true branch of conditional on line 3574 was taken. This is taken if either of the allocations on lines 3568 or 3570 failed. If the first succeeds and the second fails, then the memory will be leaked.

Static vs. Dynamic Detection

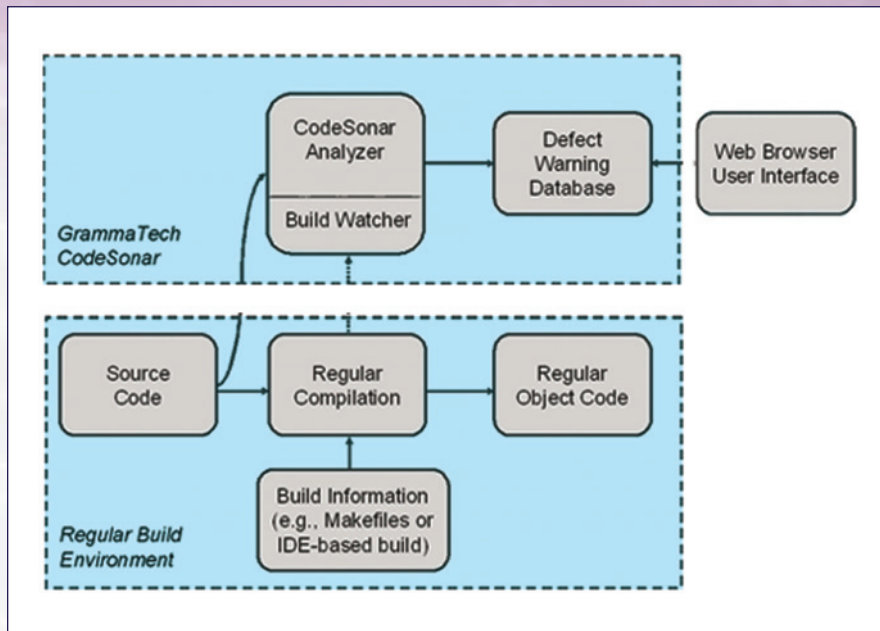


Figure 2. Architecture showing how reports are created by integrating closely with existing build systems and observing how they work.

figure for effective use. Significant effort is required to integrate with existing build systems that do not already use them. Also, they generate a large proportion of false positives, and examining these and suppressing these is tedious and time-consuming. In contrast, these new advanced static analysis tools have been designed to integrate easily and produce actionable reports with a relatively low level of false positives. Figure 2 shows the architecture which is typical of this class of tools.

These tools operate by examining the source code of the program in detail. They create a model of the code and then do an abstract execution of that model. As this pretend execution proceeds, the abstract state of the program is examined and if anomalies are detected, these are reported as warnings.

Static Analysis vs. Dynamic Testing

False positives are when a flaw is reported where no real flaw exists. With dynamic testing, false positives are rare. Usually if a test reports a false positive, it is because the test case itself is flawed and the correct thing to do is to fix it. However, with static analysis, false positive results are unavoidable in practice for non-trivial programs. These arise for several reasons. The most common reason is that there are relationships between program variables that the analysis

does not understand. The analysis algorithms are necessarily approximate, so information that would allow it to conclude that a particular path is infeasible may be lost. Another reason for false positives is that the analysis typically has little information about the environment in which the program is to be executed.

The risk of false positives means that each warning must be inspected in detail by a human to determine if it is real or not. This can be time consuming, so it is important that static-analysis tools provide facilities for helping to manage these. First, the tool should provide the evidence it used to conclude that there was a flaw in a way that is easy for the user to understand. The path to the point of failure can be complicated, taking many conditional branches and spanning multiple procedures in multiple files. A good tool will show this path, highlighting points along it where the state of the program changes in ways that are relevant to the flaw. Program-understanding features are important here too, as understanding the warning can take the user off to constructs not directly on the path such as variable and type declarations.

Second, if the user decides that the warning is a false positive, there should be a way to dismiss it so that it does not appear in subsequent reports. There is typically more than one way to do this. The most

common option is that the tool has a database of warnings that should not be reported again. A better method however is one that allows the user to provide additional information about the program, thereby educating the analysis about its properties. This is the preferred method as it can lead to fewer false negatives too.

Dynamic testing usually has very little to say about the quality of the code that it is targeted at other than it passes or fails the test. One of the great advantages of static analysis, on the other hand, is that it can find places in the code that are not strictly flaws, but indicators that something is not right. For example, if the return value of a function is checked 99% of the time, then that 1% of the time where it is not checked is notable. Finding and fixing these, even if it is just adding a comment that states it is acceptable to ignore the value at those points, contributes positively to code quality.

Finding these inconsistencies and redundancies early in the life cycle can actually reduce the cost of testing. The DO-178B standard for avionics software requires that the riskiest code be tested at 100% modified condition/decision coverage; test cases must be provided so that all conditional sub-expressions evaluate to both true and false. It is expensive to generate these test cases, and many hours have been wasted by engineers trying to figure out how to generate these cases before they realize that the code contains a redundancy that makes it fundamentally impossible to do so.

Where static analysis has little to offer is with the testing of functionality. Static analysis is good for finding places where the fundamental rules of the language are broken, or where an API is being used in an incorrect manner. If a function is intended to compute square roots, but instead computes cube roots, then static analysis will never complain about it. However, the most superficial and cursory test will reveal the flaw. It is with functionality checking that traditional testing is unbeatable.

Static analysis tools should never be considered as a replacement for traditional testing techniques. However, they are effective at finding subtle flaws, even in code that has undergone thorough testing.

This article was written by Paul Anderson, VP of Engineering, GrammarTech, Inc. (Ithaca, NY). For more information, contact Mr. Anderson at paul@grammatech.com, or visit <http://www.grammatech.com/>.