Finding Bugs in C Programs with Reactis for C

Generate tests from C code. Detect run-time errors. Track coverage: Statement, Decision, Condition, MC/DC. When an error is detected, replay a concrete execution sequence leading to the problem in order to understand, diagnose, and fix the underlying bug. Build better embedded software faster.

RSITR 2.5

August 29, 2011

Reactive Systems, inc

www.reactive-systems.com

Contents

About Reactive Systems, Inc.

Reactive Systems, founded in 1999, is a privately held company based in Cary, NC. The company's Reactis product line provides automated testing and validation tools to support the development of embedded control software. Reactis, Reactis for C Plugin, Reactis for EML Plugin, Reactis Model Inspector, and Reactis for C support model-based design with Simulink, Stateflow, Embedded MATLAB, and C code. Reactis Tester automatically generates comprehensive yet compact test suites from a Simulink model or C code. Reactis is used at companies worldwide in the automotive, aerospace, and heavy-equipment industries.

Reactive Systems, Inc.
341 Kilmayne Dr.
Suite 101
Cary, NC 27511
USA

Tel.: +1 919-324-3507
Fax: +1 919-324-3508

Web: www.reactive-systems.com
E-mail: info@reactive-systems.com

Abstract

This white paper discusses how Reactis® [a] for C, an automated test-generation tool, may be used to find bugs in C programs. Reactis for C consists of three primary components: *Tester*, *Simulator*, and *Validator*. *Reactis Tester* automatically generates test cases that stress a program, often discovering run-time errors. The generated tests aim to maximize coverage with respect to a number of test coverage metrics including Statement, Decision, Condition and Modified Condition/Decision Coverage (MC/DC). *Reactis Simulator* is a simulation environment which supports interactive execution and debugging of C programs and graphical display of test coverage results. *Reactis Validator* enables an engineer to formalize application requirements (as assertions or coverage targets) and perform an automatic check for requirement violations. Validator performs these checks by thoroughly simulating the program with the goal of violating assertions and covering targets. When a runtime error or assertion failure occurs, Validator returns a test which produces a concrete execution sequence that leads to the error, which greatly facilitates the diagnosis and repair of the underlying software bug.

[a] Reactis is a registered trademark of Reactive Systems, Inc.

1. Introduction

Reactis for C helps engineers build better software faster by finding bugs earlier.

Reactis for C is an automated debugging and test-generation tool which discovers defects in C programs using the Reactis Simulate/Test/Validate paradigm. Using Reactis for C, engineers may:

- Generate test suites from C code which comprehensively exercise all parts of a program (structural testing).
- Find run-time errors, such as memory errors, overflow errors, divide-by-zero errors, etc.
- Interactively execute a program while tracking progress towards full coverage of a variety of coverage targets including MC/DC targets.
- Perform functional tests to determine if a program can ever violate any of its requirements, including safety properties.
- Replay a specific execution sequence which triggers a defect in order to understand, diagnose, and fix a bug.

Reactis for C also includes an array of sophisticated debugging features including breakpoints, value-tracking scopes, immediate detection of memory errors, and the ability to view macro-expanded code. In this paper, we discuss how Reactis for C may be used to automate different verification and validation activities in your software quality assurance process.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

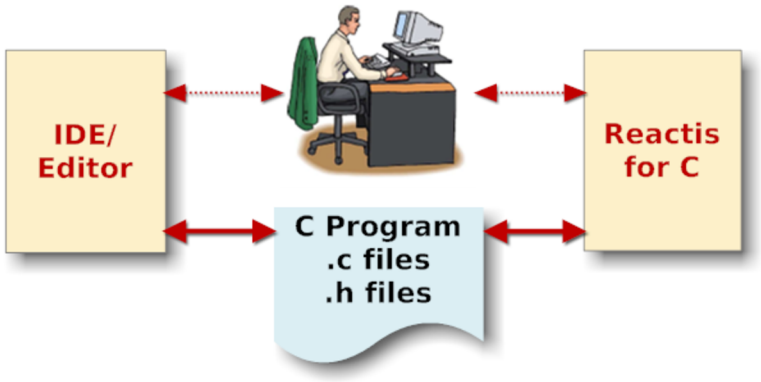

Figure 1: Reactis for C is a standalone application that reads the .c and .h files comprising a program to generate tests and find bugs.

2. Reactis Tester

Reactis Tester dramatically reduces the time and effort required to generate comprehensive test suites.

As shown in Figure 2, Reactis Tester automatically generates tests for C programs. The test suites generated by Tester provide comprehensive coverage of a variety of test coverage metrics, including the *Modified Condition/Decision Coverage* (MC/DC) test coverage mandated by the US Federal Aviation Administration (FAA) in its DO-178/B guidelines. In addition to maximizing coverage, Tester-generated test suites are automatically minimized to eliminate redundancy, saving testing time. Each test case in a test suite consists of a sequence of program *inputs* and a sequence of program *outputs* generated in response to those inputs.

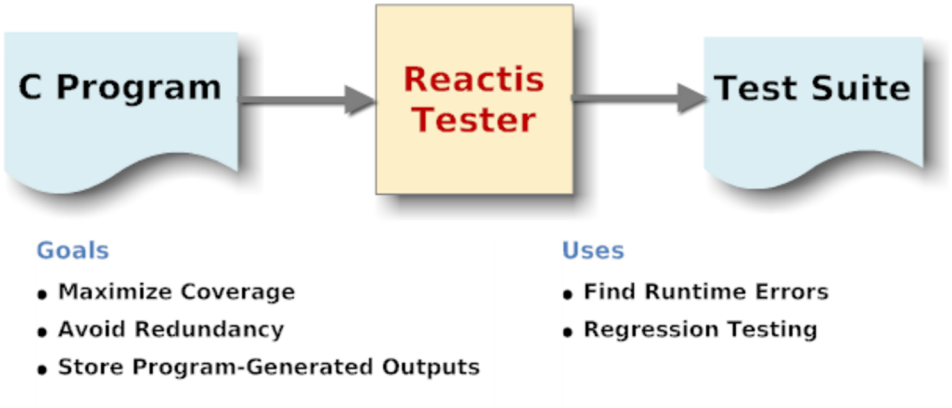

Figure 2: Reactis Tester automatically generates comprehensive yet compact test suites.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

The automatically-generated test data may then be used for a variety of purposes, including the following:

Finding runtime errors. The tests help uncover runtime errors in C programs, including memory errors, overflows, and divide-by-zero errors.

Regression testing. When a program is modified, tests generated from the older version may be run on the newer version to understand the impact of the changes on program behavior.

Reactis Tester enables engineers to maximize the effectiveness of testing while reducing the time actually spent on testing.

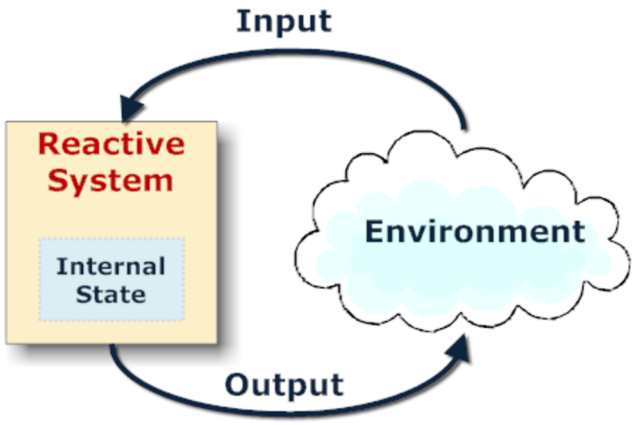

Figure 3: A reactive control system.

The embedded software applications that Reactis for C initially targets are *reactive control systems*, which typically operate as depicted in Figure 3. Such systems use a *control loop* to interact with their external environment, which consists of the following steps:

1. Read one or more input values from environment.
2. Perform internal calculations based on the input values and the internal system state.
3. Update outputs to control the environment.
4. Repeat.

To support this framework, Reactis for C lets you specify a *test harness* consisting of an *entry function*, a set of inputs, and a set of outputs. The entry function is a C function which acts as interface between the application and the test environment. Each argument of the entry function acts as either an input or an output. In addition, global variables can be selected for use as inputs or outputs. When testing a program, Reactis for C repeatedly performs the following steps:

1. Calculate a set of inputs for the harness. Record the values of all inputs.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

2. Call the entry function with the selected inputs.

3. When the entry function returns, record the values of all outputs.

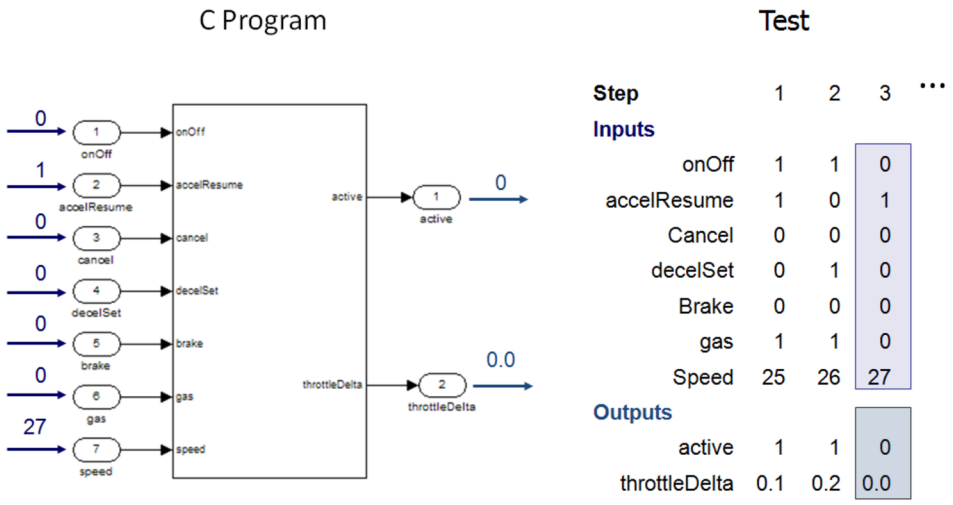

Figure 4: Structure of a Reactis for C-generated test.

Figure 4 shows the structure of a Tester-generated test. Tests are comprised of *simulation steps*. Each step is a single call to the program entry function, including the *inputs* (arguments passed to the entry function and global variables) and *outputs* (value returned by the entry function, final values of any arguments passed by reference and global variables). A test can be represented by a matrix in which each row contains the sequence of values for a specific input or output and each column contains all the input and output values for a single step. A test suite consists of a set of tests, as shown in Figure 5. When a test suite is executed the program is reset to its initial state before each test begins.

Test suites are constructed by executing a program while recording the input and output values at each step. Outputs are computed by the program at each step, but there are several ways in which the input values that drive the testing can be chosen. Input data could be manually constructed or recorded during field testing, but these are expensive tasks. Alternatively, input data can be randomly generated. Unfortunately the latter approach produces tests with poor coverage.

Reactis Tester tracks several different classes of *coverage targets*. A coverage target is a program element that should be exercised during testing. A *coverage criterion* or *coverage metric* defines how to calculate a set of coverage targets from the syntax of a program. For example, the *Statement Coverage* metric identifies all statements in a program as coverage targets, i.e. every statement should be executed to satisfy Statement Coverage. The coverage criteria tracked by Tester include Statement, Decision, Condition, and MC/DC coverage. *Decisions* are boolean expressions which determine the path of program execution. *Conditions* are atomic predicates

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

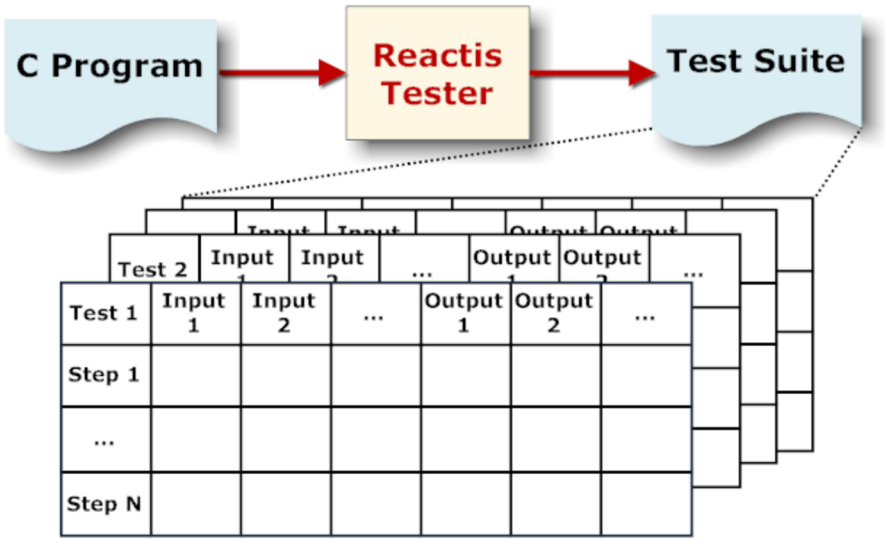

Figure 5: Structure of a Reactis for C-generated test suite.

contained within decisions. *Modified Condition/Decision Coverage* (*MC/DC*) targets are used to ensure that each condition independently effects the outcome of the enclosing decision.

Reactis Tester employs a novel, patented approach called *guided simulation* to generate quality input data automatically. Guided simulation uses algorithms and heuristics to automatically select inputs which exercise coverage targets that have not yet been covered during each simulation step.

3. Reactis Simulator

Reactis simulator extends traditional debugging with advanced error-detection and testing.

Reactis Simulator provides an environment in which program execution can be carefully controlled and monitored. Simulator's basic features are similar to traditional source-level debuggers: you can single-step through individual source statements, set breakpoints, and display data values.

However, Simulator also has a number of unique features which make it much more powerful than a traditional debugger.

Runtime error detection. Simulator instantly detects a host of runtime errors, including memory errors, uninitialized variable accesses, and integer overflows. When a runtime error occurs, execution is interrupted, unlike most C environments, in which execution proceeds using a corrupted data value. This is discussed in more detail in Section 5.

Test replay. Simulator provides the ability to replay tests, so that any point in the execution sequence leading to an error can be examined. Execution can be reversed to the beginning of

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

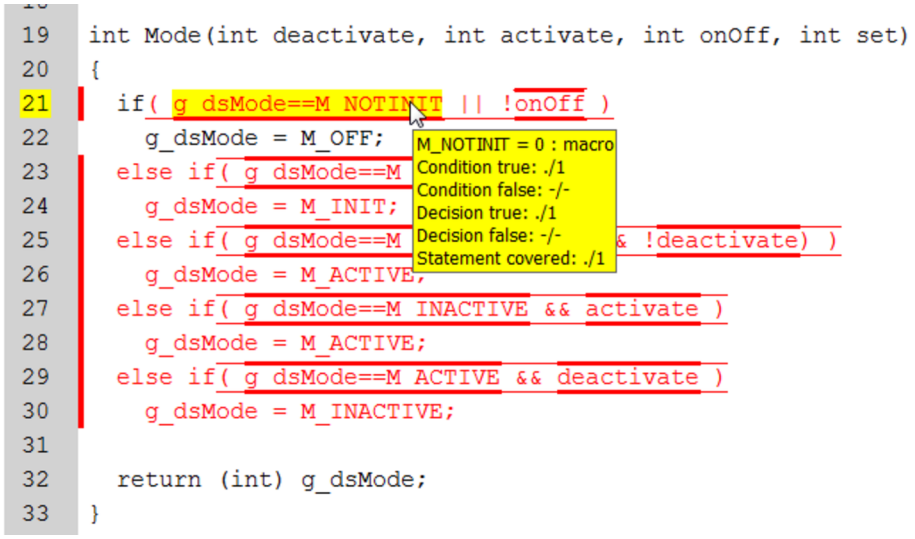

Figure 6: Coverage metrics and a C macro expansion displayed during a Simulator session.

a step at any point, making it is easy to go back and check the value of a variable at an earlier point in time.

Graphical display of coverage metrics. Simulator highlights source code according to coverage status, so that uncovered statements, conditions, decisions and MC/DC targets can be quickly found and uncovered code regions visualized. In Figure 6, code containing uncovered targets is displayed in red. A thin red overline indicates a decision which has never been true, and a thin red underline indicates a decision which has never evaluated to false. Similarly, a thick red overline indicates a condition which has never been true, and a thick red underline indicates a condition which has never evaluated to false.

Macro expansion visualization. Simulator provides the ability to view C preprocessor macro-expansions while browsing the source code. When a macro is hovered over with the mouse, its expansion is displayed. In Figure 6, the user is hovering on `M_NOTINIT`, which is a macro whose expansion is `0` (a single character string).

Test suite tuning. Simulator gives you the capability to fine tune automatically-generated test suites. Additional tests can be added, and existing tests can be modified or extended with additional steps.

4. Reactis Validator

The verification and validation capabilities of Reactis for C help engineers detect bugs earlier, when they are less costly to fix.

Reactis Validator automatically searches C programs for violations of user-specified requirements. When a violation is discovered by Reactis Validator, a test which triggers the violation is produced. This test can then be executed in Reactis Simulator to gain an under-

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

standing of the sequence of events that leads to the violation. Validator help detect defects early in the software life-cycle and reduces the effort required for code reviews. Some checks that may be performed with Validator include the following:

- Will a particular program variable ever fall outside a specified range?
- Will a thermostat maintain ambient temperature within acceptable limits of the desired temperature?
- Will engaging a vehicle's brake pedal always override the electronic throttle control?
- Will a radiation therapy machine ever deliver a dangerous dose of radiation?
- Will anti-lock brakes disengage whenever a vehicle begins to skid?

Figure 7 illustrates how Validator is used. First, a program is instrumented with *Validator objectives*. There are two kinds of Validator objectives, *assertions* and *user-defined coverage targets*. An *assertion* is a boolean C function which should always return a true (i.e., non-zero) value. A *user-defined coverage target* is a boolean C function which should return a true result at least once during the test process. The actual instrumentation process involves inserting `reactis_assert` and `reactis_target` statements into your code. The purpose of Validator is to take a program and a set of objectives and produce a test suite which triggers assertion violations and covers all user-defined coverage targets.

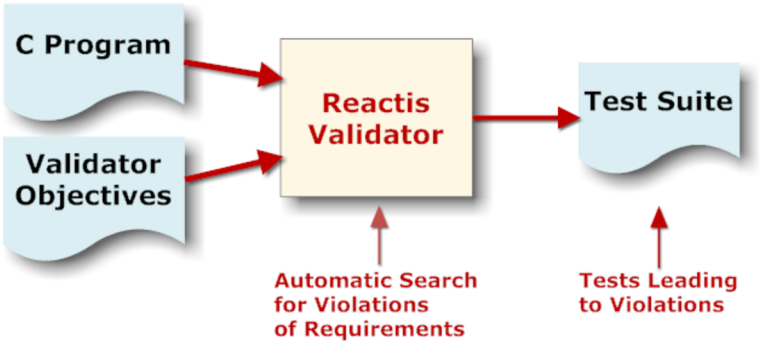

Figure 7: Reactis Validator automates functional testing.

Once the program has been instrumented, Reactis Validator performs an automated search for a sequence of input values which, when executed, leads to the violation of an assertion. Whenever an assertion violation is found, Validator produces a test that triggers the violation. This test may then be used within the interactive execution environment of Reactis Simulator in order to understand the sequence of events which causes the problem.

Reactis Validator makes it possible to detect software defects prior to code release with less effort, which in turn reduces overall development costs.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

5. Finding and Fixing Runtime Errors

Reactis for C immediately stops execution when a runtime error occurs, making it easy to find and fix.

Whenever Reactis for C is simulating C code in Simulator or generating tests in Tester, it is also performing a multitude of checks for runtime errors. The result is a powerful tool to find, diagnose, and fix a variety of runtime errors in your C code. The runtime errors detected by Reactis for C include:

- **Overflow** Numeric calculations which produce a result too large to represent.
- **Divide by Zero** Dividing a numeric value by zero.
- **Invalid Shift** Shifting an integer value by an amount which produces an undefined result according to the C standard.
- **Memory Errors** Accessing an invalid memory region in a way which produces an undefined result, such as accessing an array outside its bounds or accessing heap-allocated memory after the memory has been freed.
- **Uninitialized Data Access** Accessing memory before the memory has been initialized, so that the result of the access is undefined under C semantics.

In a typical C environment, most of the above errors do not stop program execution, but instead produce an unintended result. This result is then used for subsequent program calculations and may not result in an observable program malfunction (such as an incorrect output) until much later, making the source of the error difficult to track down. In Reactis for C, all of these errors can be immediately detected, allowing the source of the error to be quickly determined. Furthermore, the inputs which lead to the error are recorded, allowing the execution sequence to be replayed up to the point where the error occurs, making it easy to observe prior calculations which could be the ultimate root cause of the runtime error.

Figure 8 shows what happens when an integer overflow occurs in a C program. In this case, the program uses 16-bit arithmetic to calculate 1000^2. The program compiles without any errors and, when executed, generates output and terminates normally. However, instead of the expected value of one million, the value output is 16960. This is because when integer calculation results are too large to fit in the container type, the result is truncated by the most significant bits which do not fit. The result is a value which wraps around from a very large value to a much smaller value or vice-versa. Reactis for C can be configured to immediately interrupt program execution whenever wrapping would occur, making it easy to find and fix such bugs.

5.1. Memory Errors

Memory errors are particularly easy to make in C and can be very hard to debug. Reactis for C automatically detects memory errors. A memory error occurs whenever a program reads-from or writes-to an invalid address. Memory errors are particularly common in C programs

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

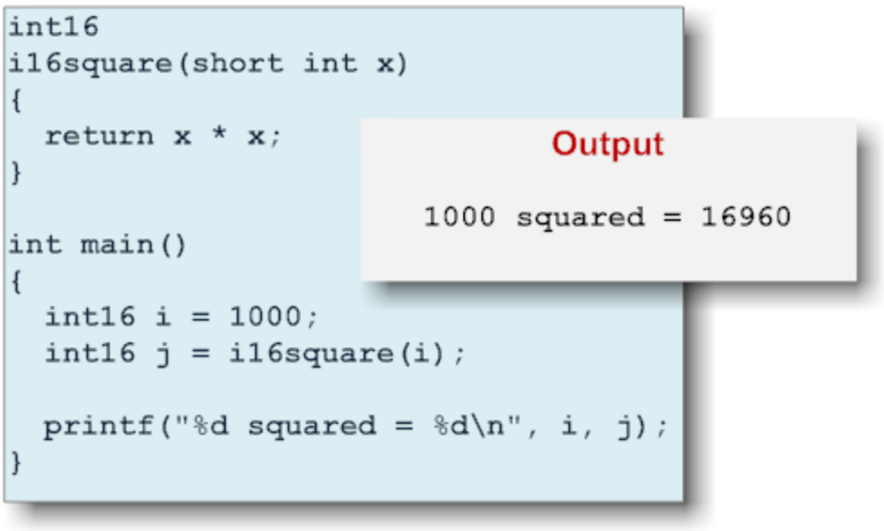

Figure 8: A program containing an overflow and its output.

because the C programming language gives the programmer direct access to the program's memory, which can boost performance but also allows software defects to access arbitrary memory locations. Typical memory errors include out-of-bounds array indexes, buffer overruns, dangling heap pointers (accessing a region of heap-allocated memory after the memory has been freed), dangling stack pointers (accessing a pointer to a local variable of a function after the function has returned) and the use of pointers cast from incorrect numeric values.

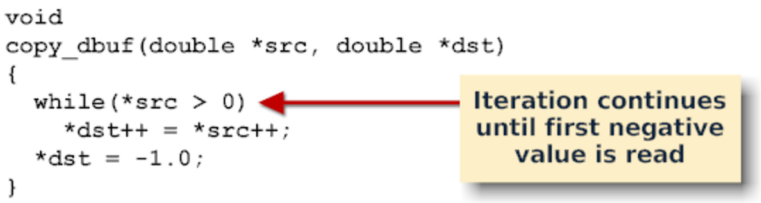

Figure 9: A function containing a potential memory error.

Memory errors can be very difficult to debug using a traditional debugger because there is often a long delay between the point where the memory error occurs and the point where the program crashes or produces an invalid output. With Reactis for C, memory errors are detected immediately as they occur, allowing the cause of the error to be quickly identified and fixed.

A function containing a typical memory error vulnerability is shown in Figure 9. The function `copy_dbuf` copies values of type `double` from one array to another until a negative value is encountered. If the number of positive values in `src` exceeds the length of `dst`, then the memory after `dst` will be overwritten. In a typical C environment, this type of error does not

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

result in an immediate error. Instead, the values stored after the array pointed-to by `dst` are overwritten. The corrupted values do not have any harmful effects on the program behavior until they are used in a subsequent calculation. Hence, there is a significant gap between the point in the program execution where the error actually occurs and the point where the error produces an observable effect. This gap in time makes the diagnosis of memory errors very difficult.

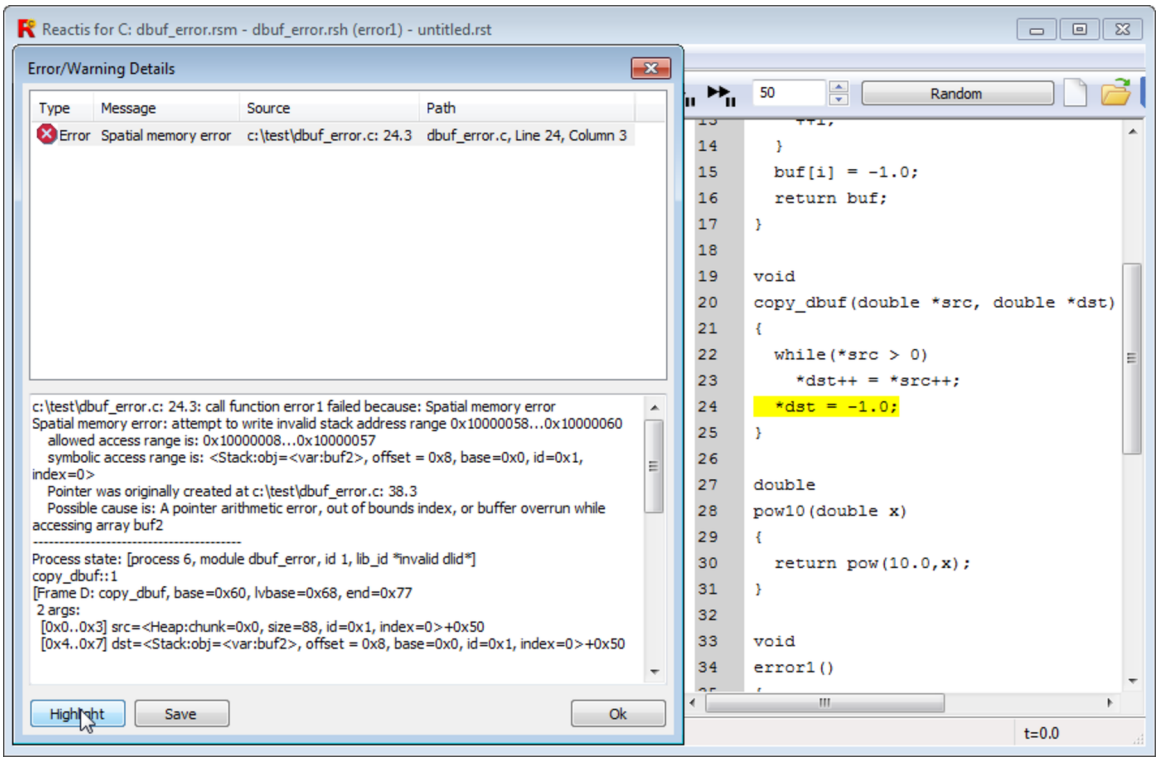

Figure 10: Memory error detected by Reactis for C.

In Reactis for C, memory errors are detected immediately (either when running a program in Reactis Simulator or generating tests). When the function `copy_dbuf()` from Figure 9 is called and the size of the `dst` buffer is smaller than the `src` buffer, an error occurs at the point where the first write beyond the bounds of `src` occurs. Program execution is suspended and an error dialog appears, as shown in Figure 10. When the *highlight* button in the error dialog is clicked, the source line where there error occurred flashes yellow, as shown in Figure 11.

A typical memory error summary and description is shown in Figure 12. (Note that for the sake of brevity, the stack trace which appears after the description text has been omitted.) The error message includes the source location of the error, the kind of error, the memory address that was being accessed at the time of the error, the allowed numeric access range and the allowed symbolic access range. The latter is particularly helpful in many cases because, when a variable is accessed via pointer, the symbolic information will include the name and source code location of the variable pointed-to. In a traditional debugger, only the numeric

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

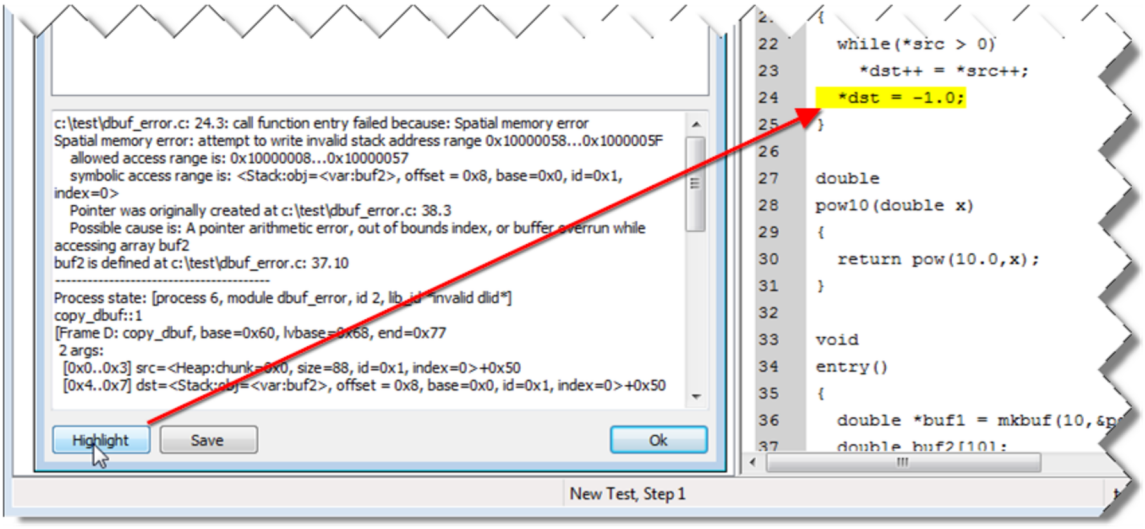

Figure 11: Highlighting the location of a memory error.

address contained within the pointer is available, and this address no longer corresponds to the original target of the pointer. This is one of the factors which makes memory error diagnosis difficult. In Reactis for C, the target of the pointer is immediately available. In this case the variable is `buf2`.

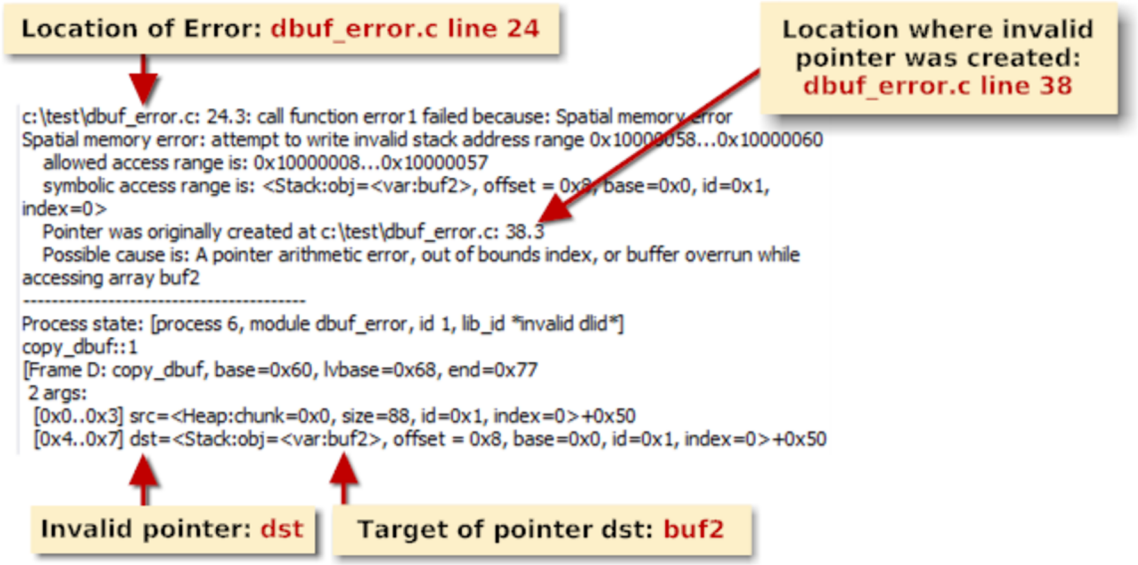

Figure 12: Closeup of error message from Figure 10.

Memory errors can be divided into two categories, *temporal* and *spatial*. *Spatial memory*

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

errors are cases where an address access occurs outside the bounds of the intended target. *Temporal memory errors* occur when memory is accessed after it has been recycled, so that the intended target may have been overwritten with new data.

Spatial memory errors include the following:

Invalid array index Accessing `A[i]` when `i` is outside the bounds of `A`.

Buffer overrun Accessing `*p` when the value of `p` has been incremented to point past the end of its target.

Invalid pointer Accessing `*p` when `p` has been overwritten with a non-pointer value (this can happen when using a `union` construct).

A *temporal memory error* occurs when a pointer is used to access heap or stack memory which has been deallocated or reallocated for some other purpose. Temporal errors can be divided into 2 categories:

Heap error Accessing `*p` when `p` points to a chunk of heap-allocated memory which has been previously deallocated via the `free()` function.

Stack error Accessing `*p` when `p` points to a local variable of a function `f()` after `f()` has returned.

Temporal memory errors are usually more complex than spatial memory errors and are hence also more difficult to diagnose and fix.

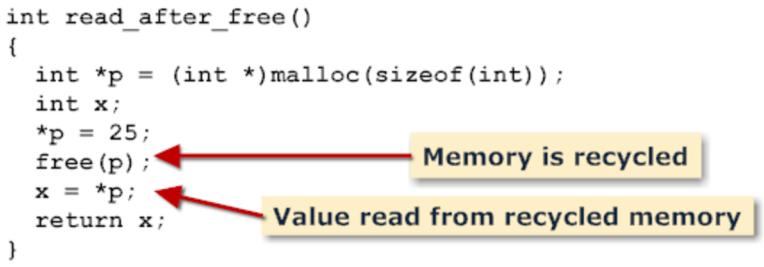

Figure 13: A function which reads from recycled heap memory.

Figure 13 shows a function which reads from heap memory after the memory has been freed. This function will compile and run without any obvious error in almost any C execution platform. However, the value returned may not be 25. This type of execution error leads to insidiously intermittent malfunctions which can be a nightmare diagnose.

Fortunately, Reactis for C detects temporal memory errors and interrupts program execution at the point where the invalid memory access occurs. Figure 14 shows the result of executing `read_after_free()` in Reactis Simulator. The memory error is immediately caught and its location (the assignment `x = *p`) is highlighted.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

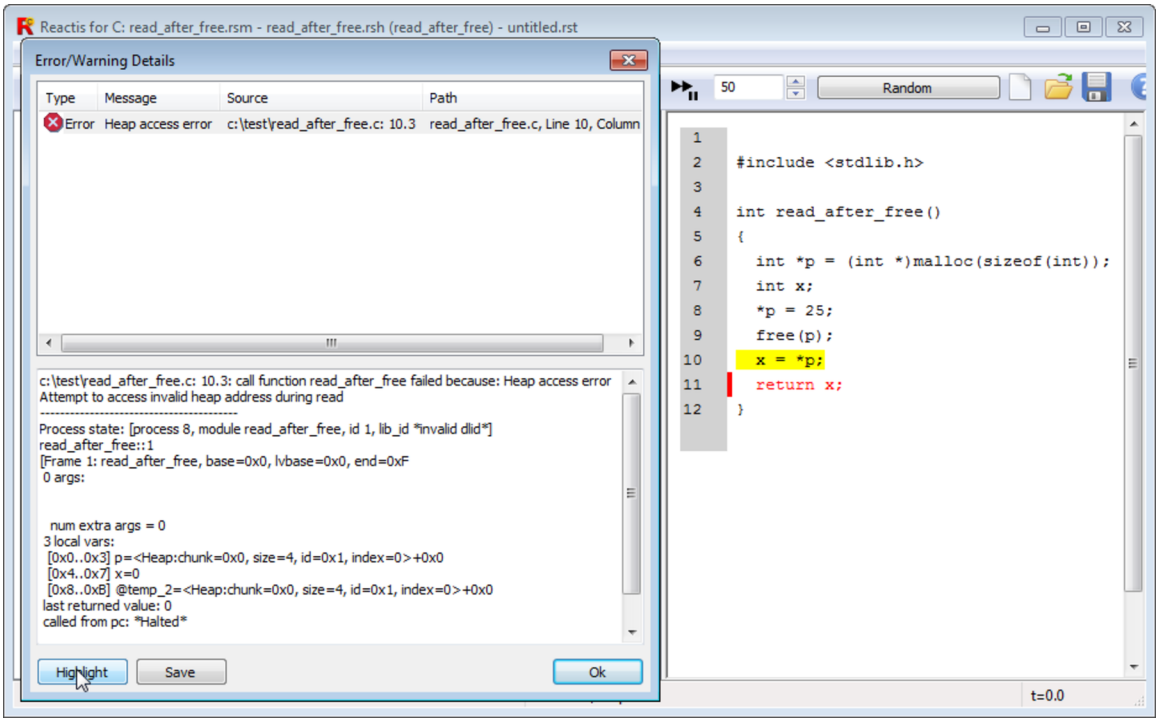

Figure 14: Reactis for C detects the error in the function of Figure 13.

5.2. Uninitialized Memory

Another class of error which is also difficult to debug in C programs is reading from uninitialized memory. There are two ways uninitialized memory reads can occur in a C program:

Uninitialized heap memory Heap memory is allocated via `malloc()` and some of this memory is not initialized before it is read.

Uninitialized local variable A local variable of a function is not initialized before it is read.

In both cases, whatever value happens to be stored in the allocated memory is used. As is the case with other memory errors, there is often a delay between the point where the uninitialized memory read occurs and the point where observable erroneous behavior occurs. An example of this is the function `sum()` in Figure 15.

In the body of function `sum()` the summation variable `x` is not initialized. This code will compile and execute on almost any C platform. The value returned by `sum()` will be equal to the sum of the first `n` values stored in `A` plus whatever value happens to be stored in the memory allocated for variable `x` when `sum()` is called.

When using Reactis for C, uninitialized memory reads trigger an immediate suspension of program execution and an error message that gives the location where the error occurred and the program variables involved. Figure 16 shows the result of executing the function `sum()` with Reactis for C.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

```
int
sum(int *A, int n)
{
  int x, i = 0;

  while(i < n) {
    x = x + A[i];
    ++i;
  }
  return x;
}
```

x is not initialized

Figure 15: A function with an uninitialized local variable x.

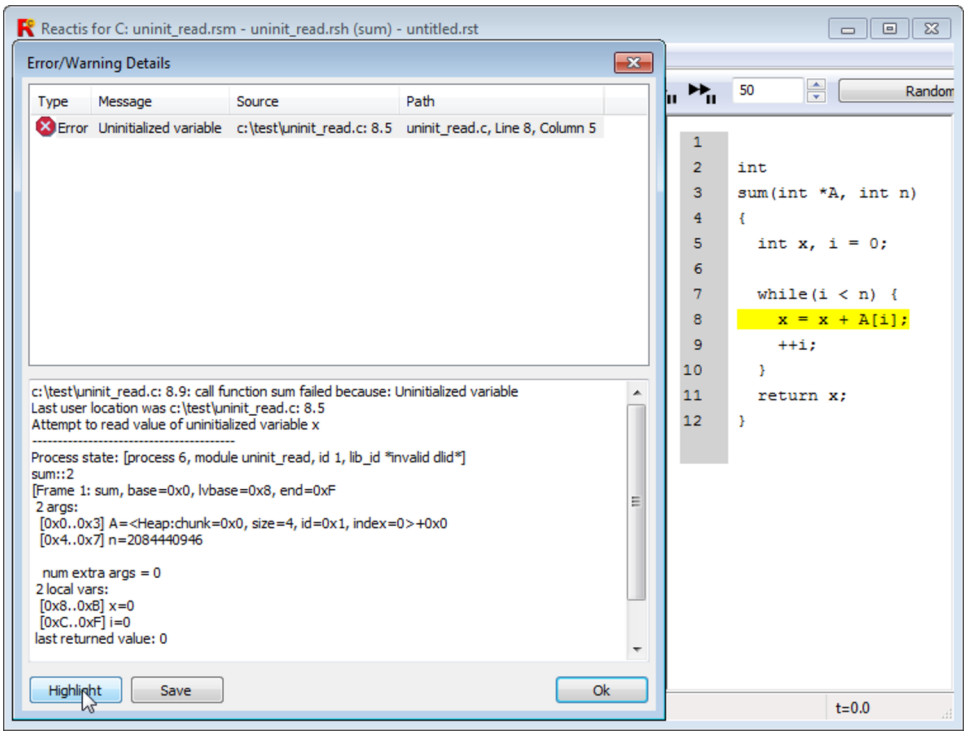

Figure 16: Reactis for C detects the error in the function of Figure 15.

Spatial memory errors, temporal memory errors and uninitialized memory reads often have subtly corrupting effects on program execution. These errors essentially inject random data into the program, causing the program to intermittently malfunction. It is also common for memory errors to only occur in rare circumstances, such as when a very large buffer size is requested or a complex boolean expression becomes true. A major strength of Reactis for C is its ability to immediately catch memory errors as they occur and to generate test inputs which are likely to trigger memory errors.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

6. Regression Testing

The automatic test-generation and execution offered by Reactis for C enables engineers to easily check whether a program conforms to the behavior of a previous version.

A crucial aspect of the tests generated by Reactis Tester is that they include all program outputs as well as inputs. Hence these tests contain all the information needed to ensure that a revised version of a program conforms to a previous version.

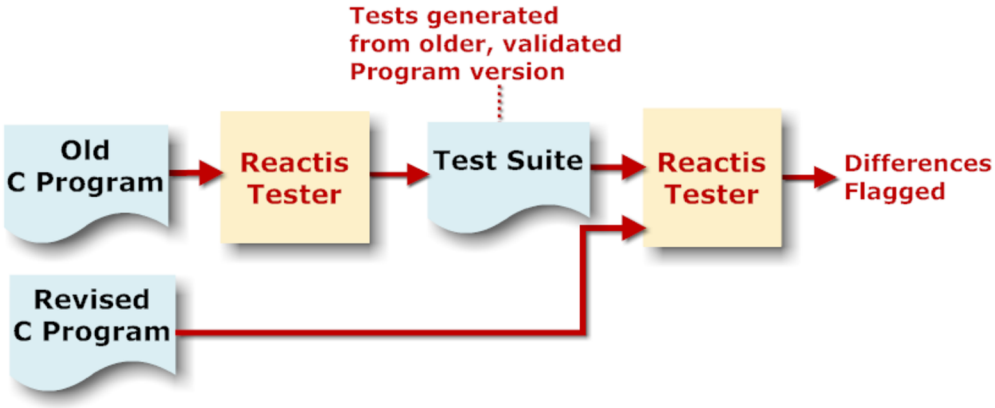

Figure 17: Performing regression testing with Reactis for C.

Figure 17 shows the work-flow when using Reactis for C to perform regression testing. The basic steps required to perform regression testing with Reactis for C are as follows:

1. Generate a test suite from the older version of the C program.
2. For each test in the suite, execute the new version of the software using the input values contained in the test.
3. Compare the output values produced by the software with those stored in the test.
4. Record any discrepancies.

The net effect of regression testing with Reactis for C is better-quality software at a lower cost. Because good test data is generated and run automatically, less engineer-time is required to create and run tests.

7. Synergy with Reactis for Simulink

Combining Reactis for C with Reactis for Simulink in a model-based development process produces synergistic benefits.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

Engineers performing model-based design using Simulink®/ Stateflow® [1] as a design tool and C as an implementation language can benefit significantly from using Reactis for C in combination with its sibling product, *Reactis for Simulink*.

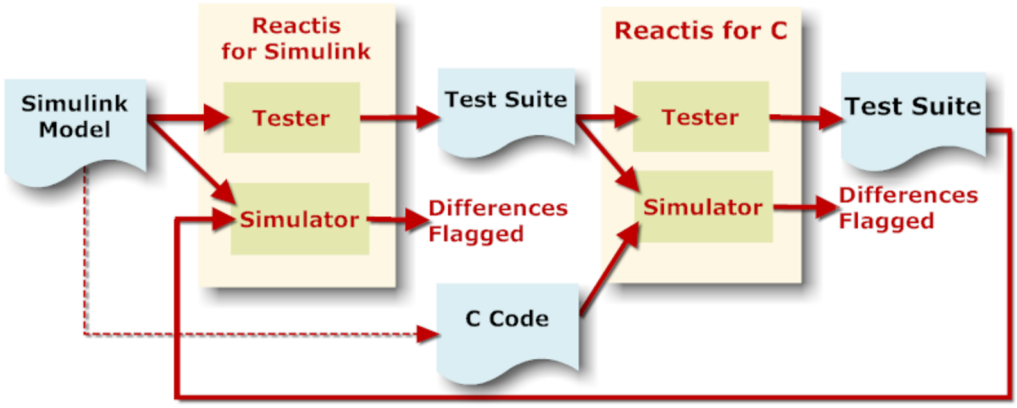

Figure 18: Validating a C program developed from a Stateflow/Simulink model.

Synergistic benefits occur when test suites are shared between both versions of Reactis. There are two basic use-case scenarios. The first case is shown in Figure 18, which depicts the work-flow in a typical model-based development environment. In this scenario, the conformance of an implementation to a model can be checked by generating a test suite using Reactis for Simulink, and then testing the implementation against the test suite with Reactis for C.

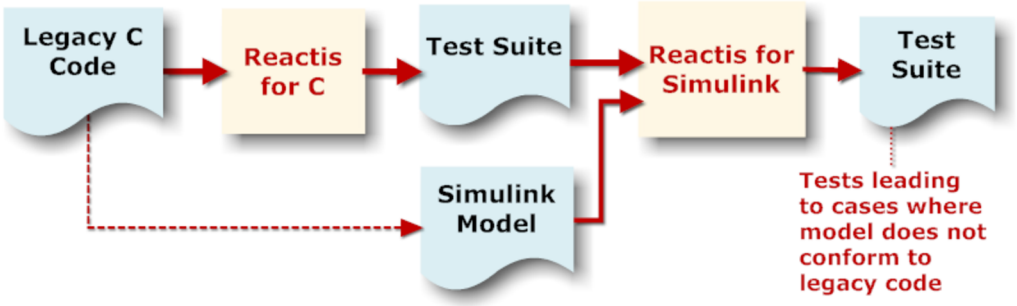

Figure 19: Validating a Stateflow/Simulink model developed from legacy C code.

Figure 19 shows the work-flow when using both Reactis products to support the reverse engineering of models from legacy C code. In this scenario, the conformance of the model to the legacy implementation can be checked by generating a test suite from the implementation using Reactis for C, and then testing the model against the test suite using Reactis for Simulink.

[1] Stateflow and Simulink are registered trademarks of The MathWorks, Inc.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.

8. Conclusions

Reactis for C improves code quality and reduces development costs.

The Reactis for C tool suite will improve the quality of your C code and reduce development costs in several ways. Reactis Tester generates comprehensive yet compact test suites, which can dramatically reduce the costs uncovering runtime errors in your code. The advanced debug capabilities of Reactis Simulator help you understand, diagnose, and fix defects in C programs. Reactis Validator automatically finds execution sequences in which program requirements are violated, including safety properties. The Reactis for C tool suite can be used to automate tasks which currently require significant manual effort, cutting development costs. Furthermore, by supporting the thorough testing and validation of programs, Reactis for C enables errors to be detected and fixed before systems are fielded, thereby reducing recall and liability costs.

Please see the Reactive Systems website at www.reactive-systems.com for ordering information and for instructions on how to download a free 30-day evaluation copy of the software.

Copyright © 2002-2011 Reactive Systems, Inc. All rights reserved.