Code Coverage Analysis Exposes Invisible Bug in the GNU C++ Library

Marcel Beemster & Vladislav Yaglamunov – Solid Sands

Code coverage analysis improves confidence in a test suite. It demonstrates which parts of the source code are actively used as a result of test execution. We run our test suite for the implementation of the C and C++ standard. Code coverage analysis tells us which lines of the library implementation are "stimulated" by the tests.

We are actually most interested in the parts of the library implementation that are *not* executed as a result of the test run. They indicate our test suite is incomplete. For every line of code that is not executed, we add a new test if possible. In Figure 1, the segment to the right is almost completely covered. The segment to the left is from a similar function that has a very similar implementation. It is not covered at all, while the tests for the left and the right segment are closely matching. What is going on here? More about that later.

```
  // We only get to here if futex_unavailable
  // was true or has just been set to true.
x struct timeval tv;
x gettimeofday (&tv, NULL);

  // Convert the absolute timeout value to a relative
  // timeout
x auto rt = relative_timespec(__s, __ns, tv.tv_sec,
                              tv.tv_usec * 1000);

  // Did we already time out?
x if (rt.tv_sec < 0)
x   return false;

x if (syscall (SYS_futex, __addr,
               futex_wait_op, __val, &rt) == -1) {
```

```
  // We only get to here if futex_unavailable
  // was true or has just been set to true.
2 struct timespec ts;
2 clock_gettime(CLOCK_MONOTONIC, &ts);

  // Convert the absolute timeout value to a relative
  // timeout
2 auto rt = relative_timespec(__s, __ns, ts.tv_sec,
                              ts.tv_nsec);

  // Did we already time out?
2 if (rt.tv_sec < 0)
x   return false;

2 if (syscall (SYS_futex, __addr,
               futex_wait_op, __val, &rt) == -1)
```

Figure 1: Code coverage of parts of the functions **_M_futex_wait_until()** *on the left side and* **_M_futex_wait_until_steady()** *on the right. Edited for legibility.*

As another example, in the math library implementation there are sections that deal with subnormal numbers. Subnormal floating point numbers have values that are close to zero and use a special floating point encoding. Because of the alternative encoding scheme, special case code is used to handle very small numbers. If our test suite does not include test cases with subnormal numbers, it will not generate coverage for these special cases. We will then add tests to the suite that verify correct operation of the library with close-to-zero values.

Thus, code coverage analysis often points our attention to special case code. Such code may handle corner cases that do not occur during normal execution, is less frequently used, and may therefore hide subtle errors in the implementation. We also find bugs by focusing on these spots and writing specific tests for them. That is the normal operation of test development in combination with code coverage analysis.

Diving Deeper

But that is not the whole story. What if we cannot write a test to cover the code that is not executed?

It happens that a particular condition in a conditional statement can never be true (or false) because it is redundant. If a condition has already been shown to be false, the code after a recheck of the same condition is simply unreachable by any possible test case. This happens more often than you might imagine and it is frequently related to optimization of special cases. For example, a string handling function may have an early check on empty strings with a quick exit, while the generic part of the function may also have its own case for zero-length strings. It is not wrong for the generic part to recheck the special case. It makes the generic code less dependent on specific preconditions and it makes its implementation independent of future changes to the earlier special case optimization. For the robustness of the code under future code changes, it may be better to leave the unreachable case in place.

The analysis of unreachable code is not trivial. One has to dive into the code and fully understand what is going on. If the code is not your own, this can be a real effort. We must preserve our analysis to avoid doing it again after the next coverage run.

For these cases, instead of writing a test, we write a *justification*. The justification is simply text that documents why the code is unreachable. The language has to be so clear that future reviewers do not have to repeat the analysis. In this way, a record is created of the analysis effort. If your code coverage tool supports it, it can recognize how the justification is attached to the code and it will not flag it again.

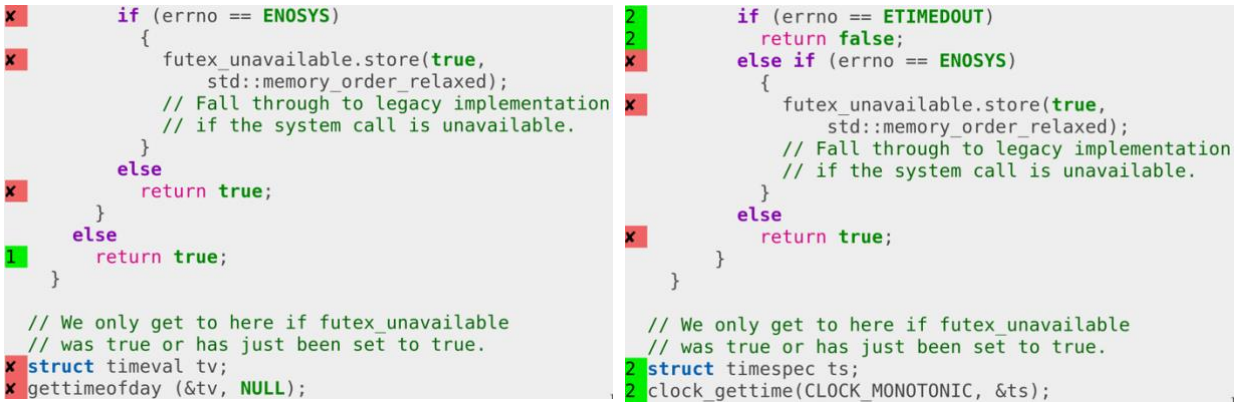
Uncovering a Performance Error in the GNU C++ Library

And sometimes, unreachable code can point to an error in another part of the code. We recently filed an error for the function **`_M_futex_wait_until_steady()`** in the **`<future>`** header of the GNU C++ library implementation. You can follow its progress at **`https://gcc.gnu.org/bugzilla/show_bug.cgi?id=105673`**. This is what happened.

Code coverage analysis of the **`<future>`** header showed that a significant portion of the second part of the **`_M_futex_wait_until()`** function is not executed by our tests. That is what you can see in Figure 1 on the left side. Analysis revealed that the implementation of **`_M_futex_wait_until()`** makes a call to the kernel to ask if it can perform a certain service efficiently. If the kernel cannot do that, the **`_M_futex_wait_until()`** function has a fallback to perform its operation less efficiently. Since modern kernels can perform the service efficiently, the fallback method is unreachable for the test suite. So far so good and we can write a justification that is valid for modern kernels.

Now, there is also a function **`_M_futex_wait_until_steady()`** that performs a very similar task as **`_M_futex_wait_until()`**, and which contains the same test and fallback. In this case we found that *the fallback code was covered!* (See Figure 1, right panel.) That was unexpected and it pointed to a coding mistake. Once observed, it was not hard to find.

In Figure 2, we see the code that is just before the fallback code. In the left panel, with the correct scenario, there are two **`else`** cases. The second **`else`** is covered as indicated by the green label on the left. That is the exit of the function if the kernel provides the efficient service. In the right panel, the second **`else`** is missing, and that case end up in the fallback code.



```
x        if (errno == ENOSYS)
           {
x            futex_unavailable.store(true,
                 std::memory_order_relaxed);
             // Fall through to legacy implementation
             // if the system call is unavailable.
           }
         else
x          return true;
       }
     else
1      return true;
   }

 // We only get to here if futex_unavailable
 // was true or has just been set to true.
x struct timeval tv;
x gettimeofday (&tv, NULL);
```

```
2        if (errno == ETIMEDOUT)
2          return false;
x        else if (errno == ENOSYS)
           {
x            futex_unavailable.store(true,
                 std::memory_order_relaxed);
             // Fall through to legacy implementation
             // if the system call is unavailable.
           }
         else
x          return true;
       }
   }

 // We only get to here if futex_unavailable
 // was true or has just been set to true.
2 struct timespec ts;
2 clock_gettime(CLOCK_MONOTONIC, &ts);
```

Figure 2: Correct control flow for **_M_futex_wait_until()** *on the left and a missing* **else** *in* **_M_futex_wait_until_steady()** *on the right. Edited for legibility.*

Instead of speeding up kernel interaction, both the fast and the slow service are engaged in the function **`_M_futex_wait_until_steady()`**. This coding mistake does not lead to a functional error in this instance, but the slowdown is certainly not intended.

Security and Backdoors

In the case of the futex error, we only found it by being smart enough to observe that some part of the code should not be covered. "Being smart enough" unfortunately does not fit into safety and security processes because we cannot reliably reproduce it.

Yet, code coverage analysis deserves a place in safety and security processes for software. Backdoors in code placed by adversaries are typically triggered by conditions that are not part of the functional specifications. If code coverage reveals that the code after **`if (password == "LegendaryWarrior")`** is not executed, it warrants a second look. That does not require being smart enough, just a procedure for checking code coverage analysis results.

The Case for Code Coverage Analysis

Requirements-based testing is a great standard for creating reliable software. The reality is that test suite creation is hardly ever driven just by the requirements. This is because the requirements are almost never defined at the same, detailed, level as the implementation itself. Code coverage analysis is a great help to point out missing tests. Coverage analysis is not easy. Yet, when executed judiciously, it verifies the

completeness of functional requirements and the rigor of the code itself. To have a test suite that fully covers all application code is a backbone for the future development of that code. It will prevent code to be broken inadvertently and it serves as a vehicle to communicate the purpose of the code that it covers. Make code coverage analysis part of your development process too.

Solid Sands' SuperGuard test suite for the Standard C Library is a requirements-based test suite. Out of the box, it has high structural code and branch coverage for standard C library implementations. It is used for the qualification and certification of standard C library implementations and it reports about the traceability between requirements and test cases.