Embedded in Academia — John Regehr, Professor of Computer Science, University of Utah, USA

How to Fuzz an ADT Implementation

regehr February 14, 2013 9 Comments

Sometimes the standard libraries don't meet your requirements. When this happens you might grab an open source b-tree, splay tree, Judy array, or whatever, or you might implement something yourself. This post is about the situation where you have some code for a data structure, you also have some unit tests, but even so you are not yet quite convinced it is trustworthy enough to be included in a larger project. The problem is how to gain confidence in it. Part of the solution is random testing, or fuzzing. Previously I **wrote about the general case**. This post gives a detailed example for C code, although the techniques are mostly independent of the language. I'll repeat the exercise later for a Python library.

We'll start with an open source red-black tree. Although you could look at the **original code**, it's going to be more useful to peek into the Github repo that I setup to accompany this post. The **baseline tag** is the initial checkin after a bit of cleaup. As a side note, this is the first time I've tried to use Github to help tell a coding story in a blog post; hopefully it'll work out.

The contents of this ADT include:

- **red_black_tree.h** defines the API for a generic red-black tree; the interface is commented and looks pretty sane as long as we bear in mind that generics aren't exactly C's strong point
- **red_black_tree.c** is the implementation, again it's fine at first glance although there isn't any checkRep() or repOK()
- **test_red_black_tree.c** is an interactive test driver: it issues API calls to the red-black tree based on commands from STDIN

- **simple_test.sh** contains the single unit test for this red-black tree, it works by passing commands to the test driver

Before writing any code let's make sure we know what we're doing here. We just want to answer a simple question:

> **Are we confident that this module will work properly in production code?**

The answer right now is "don't know yet" since we haven't even seen the code run. Eventually we'll either discover some nasty bugs in which case we'll probably abandon the code, or else we'll satisfy ourselves that it works as intended.

In some situations we might not need to run the code that we're reusing. For example, we might perform a detailed code inspection and then conclude that the code is solid. Alternatively, if this red-black tree was included in a heavily-used version of Boost, we might be comfortable using it out of the box, trusting that other users have tested it sufficiently. But that's not that case here. Finally, some friendly **Frama-C** user might prove once and for all that the red-black tree is correct, in which case we might feel kind of silly fuzzing it (I wouldn't, but others might).

Step 0: Run the Unit Tests

The provided unit tests are extremely minimal: just `simple_test.sh`, which instantiates a red-black tree and does a few operations on it. What can we learn from running this test? All we know is that for this short sequence of operations:

- The code doesn't crash.
- No assertions are violated.
- The printed output is plausible.

There are two problems. First, we have no quantitative information about the adequacy of `simple_test.sh`. Second, we haven't monitored the execution of the red-black tree very closely: it could be doing all sorts of things wrong and we'd never know it. We'll deal with these concerns one after the other.

Step 1: Get Coverage Information

We'll start with line coverage, which is usually the most basic kind of coverage that tells us anything useful. The **diffs for step 1** have a modified makefile which tells GCC to record coverage information. If we recompile, run the test script, and ask gcov about the coverage of red_black_tree.c, it says that 76% of lines were covered. **Looking a bit closer**, there are big chunks of code that aren't covered and one of the main red-black utility functions, `RightRotate()`, didn't even get called.

> **What did we accomplish in step 1?** First, we confirmed our suspicion that simple_test.sh is seriously inadequate. Second, we set a baseline for coverage that we had better be able to improve on.

Step 2: Improve Coverage Using a Fuzzer

Fuzzing an ADT implementation is never difficult, but here it's particularly simple since `test_red_black_tree.c` already contains most of the code that we need. All we have to do is drive the method calls randomly instead of taking instructions from STDIN. I made a copy of this file called `fuzz_red_black_tree.c` and made the appropriate changes. The tag is **step_2** and you can see the **changes here**. Running the fuzzer gives 99.6% line coverage of `red_black_tree.c`, a nice improvement for such a small amount of work.

> **What did we accomplish in step 2?** We almost completely closed the gap between the baseline and maximal amounts of code coverage.

Step 3: Strengthen the Test Oracles

Just covering code isn't enough: this code could be doing totally crazy stuff and we'd never know it. Here's an incomplete list of things that we want to know:

- Are there memory leaks?
- Are any non-crashing undefined behaviors executed, such as integer errors or accessing memory that is uninitialized or just a little bit out of bounds?
- Are the red-black tree's invariants respected?
- Is the red-black tree functionally correct? In other words, is it adding, deleting, and sorting elements as required?

Questions like these are answered by ***test oracles***, An oracle tells us if a test case worked or not. Since we're doing random testing, we require oracles to be automated.

The first question above can be answered by Valgrind. The second question is partially answered using Valgrind and also Clang's new `-fsanitize=integer` mode. We could add more tools to the mix but these are a good start. As it happens, neither of them reports any problems during a fuzzing run. This is a great sign that the code we're dealing with here is pretty high quality. Lots of codes will have problems that are uncovered by these tools.

Checking the red-black invariants takes a bit more work. Traditionally, the function that accomplishes this is called `checkRep()` or `repOK()`. Basically any nontrivial data structure needs one of these and here we're going to have to write it. However, even if this function existed already, we should give it a close look since many sloppy invariant checkers exist. Here are a few bits of advice on invariant checkers:

- Don't play games in the invariant checker such as trying to fix things up: just bomb the program with an assertion violation if anything is wrong.

- Be absolutely sure the invariant checker does not modify the data structure.
- In the fuzzer, call the invariant checker every time the data structure is at a stable point following a modification. In other words, check the invariants any time they are supposed to hold, but might not.
- The invariant checker should traverse every piece of memory that has been allocated for the data structure.
- All nontrivial fields of the data structure should be checked for consistency, even if it seems kind of stupid to do so.
- It's easy to accidentally write a vacuous invariant checker. To combat this, during development you can add conditions that are too strong and make sure they fail, and also print a few statistics such as the number of nodes visited. Actually, a better idea (suggested by Alex Groce in a comment) is to break the tree code, not the invariant checker.
- BE ABSOLUTELY SURE THE INVARIANT CHECKER DOES NOT MODIFY THE DATA STRUCTURE.

Here's the recursive helper function for `checkRep()`; the top-level checking function is just below. Running the fuzzer (after modifying it **to call the checker**) finds no problems.

Finally we're ready to ask if the red black tree actually works. Is it possible that it doesn't work even if it has passed all of our tests so far? Certainly. For example, a logic error could just drop a node while leaving the tree in a consistent shape. This last oracle requires a bit more work than the others because we require a reference implementation for this kind of ADT. In a nice programming language, all sorts of container classes would be sitting around, but this is C. We'll just hack up a quick and dirty container rather than trying to reuse something; this is simple if we can tolerate operations that have up to O(n log n) running time. The other thing we need to do is modify the fuzzer a bit. First, it must perform operations on the red-black tree and the reference implementation in parallel. Second, it must contain assertions ensuring that the results of these operations are the same. **Here are the diffs accomplishing all this**. This is all pretty straightforward, but one tricky thing we needed to do is modify the fuzzer to avoid adding multiple nodes with the same key. This is not great as far as fuzzing goes (we want to make sure that red-black tree properly deals with duplicated keys). However, I could not think of a way to test functions such as `treePredecessor()` against the reference implementation in the presence of duplicated keys, which stop the predecessor from being uniquely defined. Anyhow, we'll fix this later.

> **What did we accomplish in step 3?** We greatly strengthened the test oracles; now we can be sure that a fuzzing run does not violate various C language rules, it does not violate any red-black invariants, and it does not cause the red-black tree to get out of sync with a reference implementation of a sorted container.

Steps 4-9

The hard parts—writing the fuzzer, the `checkRep()`, and the functional correctness oracle—are finished, but we're not quite done. At this point I'll start to move faster and cover each point only briefly.

- **Step 4:** Let's take another look at the **coverage information**. There are two things we need to address. First, we missed line 591. This could be due to a flaw in the tree code, a flaw in our fuzzer, or we may have simply not run the fuzzer for long enough. The second problem is that a number of lines are executed just once. Let's try to fix these both by wrapping the entire create/fuzz/destroy sequence in a loop. This is a common idiom seen in fuzzers. The desired effect is achieved: line 591 gets hit many times and also we cover the tree create and destroy code more times. **Diffs for step 4.**
- **Step 5:** The extra loop nest has slowed down the fuzzer; it now takes about 7 seconds to execute even when its output is redirected to /dev/null. Let's remove the output and just trust it to exit with a useful message when something goes wrong. After **these changes** it runs in 2.5 seconds with the

existing compilation options and 1.2 seconds with optimization turned on. We no longer cover the `RBTreePrint()` function but we can probably live with that.

- **Step 6:** Let's return to the problem where the fuzzer is never adding elements to the tree that have duplicate keys. A reasonable solution is: for every tree instance, randomly decide if duplicate keys are permitted. If so, we'll run a slightly weaker checker that doesn't look at the results of the predecessor and successor queries. This turns out to be **super easy**.
- **Step 7:** Another issue we might consider is what proportion of duplicated keys will lead to the best fuzzing results. This is hard to figure out ahead of time so we'll just randomize it. **Diffs here**.
- **Step 8:** Up until now we've only looked at the keys of tree nodes. We should also check that the info field (a `void *`) is retained across all tree operations. This requires a quick hack to the reference container and also a few additional assertions in the fuzzer. Again, we have to be careful about duplicate keys. Also in this step we'll randomize the number of fuzzing steps per tree instance. **Diffs here**.
- **Step 9:** Finally we'll look at coverage once more and then re-run the fuzzer under valgrind and under the integer sanitizer. As far as we can tell this red-black tree is solid code and we're going to call this job finished.

Our general strategy during these refinement steps is to keep iterating on the various aspects of the fuzzer until it feels done. Anything that seems like it might profitably be randomized, should be randomized. We must bear in mind the various ways that bugs can **hide from tests with good coverage**. We should also remember that while poor code coverage indicates a bad random tester, good coverage does not imply that a fuzzer is any good.

What Have We Missed?

Turns out there are a few obvious things we haven't yet tested:

- Our red-black tree supports multiple instances but we have only tested one instance at a time. Thus, we'll have missed any bugs where the tree code inadvertently refers to global state.
- We never try negative key values.
- Although we have fuzzed the API provided by the red-black tree, we haven't fuzzed the APIs that it uses, such as `malloc()/free()`.

In a real-world fuzzing situation we might do these things or we might simply inspect the code in order to convince ourselves that these factors don't affect the behavior of the tree code.

Conclusion

This post has two points. First, you should write ADT fuzzers. It is often more effective (for a given amount of your time) at finding obscure bugs than hand-written unit tests are. Second, fuzzing an ADT is really easy. It took me around 90 minutes to do the work reported here, and wouldn't have even taken that long if I hadn't needed to read up on red-black invariants. Besides being easy, writing a custom fuzzer also gives you a very solid feel for some important characteristics of the code you intend to reuse: Is it fast? Do the method calls perform reasonable error checking? Is the code tightly written? Furthermore, if you do this testing work yourself, as opposed to running someone else's fuzzer, you get to tailor the fuzzer to emphasize the use cases that are important to your application. Testing is necessarily incomplete but if you beat the crap out of the subset of the API that you actually intend to use, it's possible to become reasonably confident that it will work in practice. Of course if your use of the API changes over time, it might be a good idea to revisit the fuzzer.

I learned a couple of things while writing this post. First, Github is more awesome than I knew and I really enjoyed integrating it into a blog post about coding. Second, I probably have little more data structure OCD than I had realized.

9 replies on "How to Fuzz an ADT Implementation"

David R. MacIver
February 14, 2013 at 3:24 pm

In the past when I did some collection implementations in Scala, I found the ScalaCheck stateful testing implementation (see https://github.com/rickynils/scalacheck/wiki/User-Guide for details) very useful for this sort of thing.

The idea of scalacheck in general is that you're generating random data to test assertions (it's based on Haskell's quickcheck). The stateful testing essentially inverts that: You generate random sequences of operations, each of which comes with a set of invariants it can check after executing. An especially nice feature is that once it's found a sequence of operations that will cause an invariant violation it then attempts to minimize that sequence.

regehr
February 14, 2013 at 3:34 pm

Hi David, I hadn't seen ScalaCheck but it looks great. I love the QuickCheck paper, it contains a lot of good wisdom about random testing, though I don't use the tool since I don't write Haskell.

Jesse Ruderman
February 14, 2013 at 6:17 pm

This kind of fuzzing gets hairier when you consider higher-order functions, especially in a language like C that doesn't let you temporarily freeze a data structure.

* Would you consider it a bug if the red-black tree crashed when given a Compare function that returned random results? What about a Compare function that attempted to modify the tree?

* In the future you might be tempted to add a version of RBEnumerate that calls a callback rather than returning a stack. Then you'll have to deal with the possibility of the tree being modified during that callback.

regehr

February 14, 2013 at 9:06 pm

Hi Jesse, I would definitely not consider it a bug if the tree crashed when the compare function violated its contract in either of these ways, since this is a tightly-coupled internal interface.

The modify-during-iteration example is maybe trickier, we probably want it to behave at least slightly gracefully.

Anyway you are asking hard questions about API design here! I was just trying to write an easy piece about fuzzing.

Alex Groce

February 15, 2013 at 9:38 am

I have comments here, as you might guess; we're in the middle of PL faculty candidates week and I'm finishing up "bottling up" the testing class for our ecampus, but I'll chime in shortly. 🙂

Great post.

Alex Groce

February 15, 2013 at 10:11 am

"Itâ€™s easy to accidentally write a vacuous invariant checker. To combat this, during development you can add conditions that are too strong and make sure they fail, and also print a few statistics such as the number of nodes visited."

I've found handmade "mutant testing" to be even more effective for this: add a subtle but not too-subtle break to something in the code; if you don't find it, either your invariant checker is bad or the tests are lousy.

Alex Groce

February 15, 2013 at 10:21 am

The main reason for the mutant thing is that I find breaking an invariant subtly easier than making one too strong, and once I've written a RepOk, I like to touch that code as little as possible because in C I live in constant fear of modifying whatever I'm crawling over.

regehr

February 15, 2013 at 12:40 pm

Alex breaking the code is definitely a better idea than breaking the checkRep()!

David R. MacIver

February 16, 2013 at 8:34 am

I think it's generally a good rule that you shouldn't trust any test you've never seen fail. Breaking the code is usually the best way to get that.

Comments are closed.