Embedded in Academia

{ 2011 09 20 }

Better Random Testing by Leaving Features Out

[I wrote this post, but it describes joint work, principally with Alex Groce at Oregon State.]

This piece is about a research result that I think is genuinely surprising — a rare thing. The motivating problem is the difficulty of tuning a fuzz tester, or random test case generator. People like to talk trash about random testing but what they usually mean is that random testing doesn't work very well when it is done poorly. Applied thoughtfully and tuned well, random testers are often extremely powerful system-breaking devices.

Previous to our current work, my approach to tuning a random tester was basically:

1. Aim for maximum expressiveness in test cases. In other words, maximize the features and combinations of features seen in test cases.
2. Repeatedly inspect the test cases and their effect on the system under test. Use the results to tune the test case generator.

I even wrote a blog post about this last spring. Alex had a different idea which is basically:

> For each test case, randomly omit a subset of the features that might appear.

The funny thing about this idea is that it doesn't give us any test cases that we couldn't generate before. All it does is change the distribution of test cases. For example, if we're generating test cases for a stack ADT, pure random testing will eventually generate a test case with all "push" operations. But this may be very, very rare. On the other hand, swarm testing (what we are calling Alex's idea) will generate a test case containing only push operations once every 2^N tests, where N is the number of features. An exponentially unlikely event may not seem very likely, but keep in mind the number of features is generally not going to be very big compared to the number of choices made during generation of an individual test case.

The first nice thing about swarm testing is that it's easy. Many random testers (including my most recent one, Csmith) already support omission of many features, and if not it's simple to add in all test case generators I can think of.

The second nice thing about swarm testing is that it works. We took a reasonably fast quad-core machine and let it spend a week trying to crash a collection of 17 C compilers for x86-64 (several versions of GCC, several versions of LLVM, and one version each of Intel CC, Sun CC, and Open64) using the latest and greatest Csmith. It found 73 distinct ways to crash these compilers. A "distinct way" means the compiler, while dying, emitted some sort of message that we can use to distinguish this kind of crash from other crashes. For example, here are two distinct ways that GCC 4.6.0 can be crashed:

- internal compiler error: in vect_enhance_data_refs_alignment, at tree-vect-data-refs.c:1550
- internal compiler error: in vect_create_epilog_for_reduction, at tree-vect-loop.c:3725

Sometimes the compiler isn't so friendly as to emit a nice ICE message and rather it prints something like "Segmentation fault." In those cases we're not able to distinguish different ways of producing segfaults and so we just count it once.

After using Csmith to find 73 distinct crash bugs, we ran another week-long test using swarm. This time, we told Csmith to randomly omit:

- declaration of main() with argc and argv
- the comma operator, as in x = (y, 1);
- compound assignment operators, as in x += y;
- embedded assignments, as in x = (y = 1);
- the auto-increment and auto-decrement operators ++ and --
- goto and labels
- integer division
- integer multiplication
- long long integers
- 64-bit math operations
- structs
- bitfields
- packed structs
- unions
- arrays
- pointers
- const-qualified objects
- volatile-qualified objects
- volatile-qualified pointers

For each test case, each feature had a 50% chance of being included. In the swarm test, 104 distinct compiler crash bugs were found: an improvement of more than 40%. Moreover (I'll spare you the details) the improvement is statically significant at a very high confidence level. This is a really nice improvement and we intend to not only bake swarm into Csmith's default test scripts, but also to add support to Csmith for omitting a lot more kinds of features — the list above is pretty limited.

Why does swarm testing work? Our hypothesis is that there are two things going on. First, sometimes a test case just contains too many different features to be good at triggering bugs. For example, if I have a compiler bug that can only be triggered by highly complex mathematical expressions, then the presence of pointers, arrays, structs, etc. just gets in the way. The stuff clutters up the test case, making it hard to trigger the bug. The second thing that can happen is that a feature actively prevents a bug from being triggered. Consider the example outlined above where a stack ADT contains a bug that only manifests when the stack is full. If my test cases contain lots of pushes and pops, we're going to have a hard time randomly walking all the way to a full stack. On the other hand, swarm testing will frequently create tests that only contain push operations. We've looked at a lot of swarm testing outcomes and found examples of both of these phenomena. I'll post some more detailed results if I get a chance.

The overall outcome of this line of research, Alex and I hope, is to take some of the pain out of developing a highly tuned fuzz tester. This post has focused on compilers because that's the domain I know. We have another case study in the works, but Alex is managing that effort and I'm not going to publish his results. Rather, we're working on a paper about this topic that we hope to submit in the near future.

Embedded in Academia

{ 2011 02 10 }

Probabilities in Random Testing

A typical real computer system has an extremely large input space and no testing method can cover more than an infinitesimal part of it. On the other hand, broad regions of the input space will trigger bugs. The problem is that we do not know the shapes or locations of the buggy parts of the space. Random testing is a shotgun approach to finding these regions; empirically, it works well.

I've written, or supervised the writing of, about 10 random test case generators. The most difficult and unsatisfactory part of engineering a good random tester is setting the probabilities properly. For example, if I'm generating JavaScript programs to stress-test a browser, I need to decide how often to create a new variable vs. referencing an existing one, how often to generate a loop vs. a conditional, etc. The shape of the eventual JS program depends strongly on dozens of little decisions like these.

A few more observations:

- If the probabilities are chosen poorly, the tester will be far less effective than it could have been, and may not find any bugs at all
- Probabilities alone are an insufficient mechanism for engineering good test cases, and need to be combined with other mechanisms such as heuristic limits on the size of various parts of a test case
- Equivalent to the previous bullet, probabilities sometimes want to be a function of the test case being constructed; for example, the probability of creating a new function may decrease as the number of generated functions increases
- The connection between probabilities and high-level properties of test cases may be very indirect
- Alternatives to playing probability games, such as uniform sampling of the input space or bounded exhaustive testing, are generally harder or less effective than just getting the probabilities right

In practice, probability engineering looks like this:

1. Think hard about what kind of test cases will drive the system under test into parts of its state space where its logic is weak
2. Tweak the probabilities to make the generated tests look more like they need to look
3. Look at a lot of generated test cases to evaluate the success of the tweaking
4. Go back to step 1

Random testing almost always has a big payoff, but only when combined with significant domain knowledge and a lot of hard work.

Posted by regehr on Thursday, February 10, 2011, at

10:10 am. Filed under Computer Science, Software