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.

Posted by regehr on Tuesday, September 20, 2011, at 12:57 pm. Filed under Computer Science, Software Correctness. Follow any responses to this post with its comments RSS feed. Both comments and trackbacks are currently closed.

{ 14 } Comments

1. Pascal Cuoq | September 20, 2011 at 1:28 pm | Permalink

   Hello, John.

   Great post. Incidentally, your content management system interpreted the auto-decrement operator as code for a long dash.

2. Mauro | September 20, 2011 at 2:35 pm | Permalink

   It looks to me like you are doing "category partition" for the constructs of the language. Typically to avoid explosion of combinations one can use pairwise combination or more (3-wise) if feasible... did you try something like this?

3. Alex | September 20, 2011 at 4:14 pm | Permalink

   Mauro: I don't think what we're doing is too much like category partitioning or partition testing, in a way — no formal spec like category partitioning, and we don't care if the configurations are disjoint, overlapping, etc. at all. Nor do we care much about getting all pairwise combinations — but the P(appearing) for each feature was 50% — so if you have 1000 configurations and run even a small number of tests with each config, you'll run plenty of tests with any pair you want, and fewer with 3 tuples or more, of course.

4. regehr | September 20, 2011 at 4:28 pm | Permalink

   I'm kind of interested in figuring out what's a good probability for each feature. I mean, is it 10% or 50% or 90%? Obviously this will be specific to the tester and SUT, and also it'll be expensive to discover via experimentation, so maybe I shouldn't be interested...

5. Derek Jones | September 20, 2011 at 7:05 pm | Permalink

   I would guess that most crashes are caused by a few kinds of language constructs (a lot of distributions in software engineering have a power law-like form). Not generating those constructs provides an opportunity for other constructs to cause a crash, resulting in an increase in diversity of crash sites.

   Not generating the comma operator or goto/labels will increase the average size of basic blocks, providing more optimization opportunities which can go wrong (typical code has short basic blocks and few optimization opportunities).

   To increase diversity, if that is your aim, you need to not generate those constructs that cause the greatest number of crashes. How many unique crashes can the compilers generate?

What model of the crash generation process did you use to do the statistical significance test?

6. regehr | September 20, 2011 at 8:28 pm | Permalink

   Hi Derek,

   The number of distinct crash symptoms is (very nearly) bounded above by the number of asserts in the compiler source code. So, ~12,000 for LLVM and ~16,000 for GCC, if I'm counting correctly. The true number of distinct errors is almost certainly a lot less than this.

   Regarding a model, we simply assume that the numbers of distinct crashes found in multiple test runs follows a normal distribution. I didn't verify this assumption but have no reason to suspect that the approximation is a bad one.

   Using swarm testing, seven 24-hour test runs found 44 39 39 33 37 36 39 distinct bugs. Using default Csmith, seven 24-hour test runs found 32 29 32 29 29 29 36 distinct bugs. A two-tailed t-test without assumption of equal variance gives p = 0.00087, corresponding to a confidence >99.9% that the populations are not the same.

7. Alex | September 20, 2011 at 11:56 pm | Permalink

   If you have a file system, I think you want rename with > 50%, but that's all I'm able to hazard. I think it might matter, because for things like serious compilers, or file systems, or flight software, we're going to test it more than once. Or twice. And we don't mind spending a few weeks compute time. Unfortunately, we probably fix it and the goal shifts — once we find a bug we don't care if we find it again, and if we find a "good" swarm we tend to fix the bugs it is good at finding...

8. regehr | September 21, 2011 at 12:36 am | Permalink

   Alex, yeah — I think over-fitting is the main worry for any kind of reinforcement-based strategy for tuning a random tester. But solutions to this problem (if I knew what they were) seem very hard to evaluate.

   On the other hand, we've already spent a few years reporting compiler bugs found with random testing. Perhaps there's a way to mine the revision history of GCC and LLVM and combine that with what we know about bugs we've reported / they've fixed in order to better understand over-fitting and how to deal with it?

9. Derek Jones | September 21, 2011 at 6:34 am | Permalink

   John, Thanks for the numbers.

   Fault data is very rarely normally distributed in any field (although your second set of numbers does pass the Shapiro Wilk test), with Poisson being a favorite choice for hardware (I'm not so sure this distribution applies to software). The two data distributions look very different and a Friedman test rejects the null hypothesis that they are the same.

   The most interesting figures are the number of asserts. I imagine some a very hard to

hit corner cases or developers not being sure that some case can never occur. Perhaps the ones being missed are in heavily nested if statements (the Coccinelle tool can be used as a semantic grep tool).

10. Mauro | September 21, 2011 at 12:38 pm | Permalink

    Alex: I think you start from very similar observations respect to n-wise testing:
    1) statesapace is too big to explore exaustively
    2) tests combining few features reveal most of the faults

    Maybe a comparison between the two technical solutions could be interesting

11. Alex | September 24, 2011 at 12:49 pm | Permalink

    Mauro:

    True, though I think the motivation here is really more the other way: tests removing some features help expose faults. That's the flipside — I'm not a combinatorial testing guy, but iirc, you often aim to "kill multiple birds with minimal stones" by doing some kind of hitting set to get combinations that cover multiple value pairs, right? Is there much emphasis on avoiding certain combinations (not actually knowing which ones to avoid!). The description in combinatorial is more low-level, rather than a pretty abstract predicate over tests features, typically, as I understand it. It's obviously a related area, and maybe for the kinds of input domains n-wise looks at, adding more emphasis on what NOT to combine (even at efficient small-test-set coverage) might be interesting!

12. Jakob Engblom | September 24, 2011 at 2:36 pm | Permalink

    Interesting result. Trying to fit it with my experience as a user, developer, tester, and product manager of a complex piece of software (with a much more set of inputs than a C compiler), I think what you find is that you need to use a few different features at once to find "interesting" bugs. Feature interaction is what tends to crash or produce unexpected or invalid results. On the other hand, it is nice to take tests to some kind of extreme.

    Doing a "reasonable mix" of operations on a system tends to get close to "expected behavior" and therefore close to what developers have (intuitively, mostly) wrote for and tested for.

    Doing a "degenerate mix" (such as only push operations for a stack) will tend to uncover the code paths and system states that nobody thought would happen. More often than not, the initial reply to a problem report based on some extreme use of a system is "why did you do that"? Or "you did WHAT?".

    My intuition is that omitting certain features will tend to get closer to "extreme" uses of a piece of software, finding the bugs that far from the typical use. That means that they might hit a smaller population of users, but still they are valid bugs.

13. Alex | September 24, 2011 at 6:25 pm | Permalink

    Jakob — I think that's true. Here, the nice thing is that statistics suggests you'll see nothing too outlierish, too often (other than in the sense than random tests all tend to be odd behavior). But I certainly do plenty of interactions with a file system that involve

only mkdir/rmdir/rename and little file creation, for example. Or the reverse. But random tests look very different than “reasonable” tests.

14. Jakob Engblom | September 25, 2011 at 1:22 pm | Permalink

    To me, as a product guy, the important thing is to find as many odd behaviors and plain bugs before my customers do. What is “extreme” or “outlandish” to the developers could well be “standard practice” for a customer with a very specific and peculiar use case.

    Unfortunately, unless you have tons of data like Microsoft, it is very hard to know what a “typical” use case is in practice.