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

Correctness. Follow any responses to this post with its comments RSS feed. Both comments and trackbacks are currently closed.

{ 2 } Comments

1. Phillip Mates | February 11, 2011 at 12:53 am | Permalink

   After seeing you guest lecture about this topic in Prof. Might's Compilers class I've been thinking about this a bit.

   The comments/suggestions below might be completely useless but I'm interested in what you have to say about them.

   Would it be possible to randomly test different probabilities or probability functions and select the ones that generate bugs when used?

   One problem you raised is only being able to randomly generate a subset of C due to the possibility of creating ambiguous programs. Is there any way to identify ambiguous programs in a automatic or semi-automatic fashion? If so could the code generation algorithms be trained to avoid those areas of the input space?

   Is it possible to plot bug-revealing programs in the input space (or a subset due to its size) and look for clustering?

   What languages, other than C, have you implemented your random testing for? Is there a general testing framework built to enable the addition of new programming languages?

2. regehr | February 11, 2011 at 4:36 pm | Permalink

   Hi Phillip- These are good questions, feel free to stop by my office to chat about this more.

   It would definitely be useful to do a parameter space search for a collection of probability assignments that leads to effective bug-finding. We haven't done it for Csmith because it would be slow– evaluating a single data point with high confidence takes maybe a week of CPU time, and the space to be searched is very large. No doubt there are ways to be clever but we haven't taken the time to think about it very carefully.

   Detecting undefined and unspecified behaviors in C programs in a general way is hard, but there are tools for doing this. So far we haven't tried them out, since it's a lot simpler to have a self-contained program generator.

   I think the dimensionality of the program space is too high for effective visualization. There are people at Utah who know a lot about this stuff — I haven't talked to them. I suspect the program space is fundamentally hard to visualize.

   Re. a general framework for random testing, we've just barely starting on an effort in this direction. Let me know if you're looking for a project  . I'm not sure that we'll have

anything at a useful point in time for Matt's class.

{ 1 } Trackback

1. Tweets that mention Embedded in Academia : Probabilities in Random Testing -- Topsy.com | February 10, 2011 at 12:39 pm | Permalink

   [...] This post was mentioned on Twitter by Fabian Knopf and Marc Ruef, John Regehr. John Regehr said: Probabilities in Random Testing: A typical real computer system has an extremely large input space and no testi... http://bit.ly/eHzb1M [...]