Jakob Sunde (jsunde)
Alex Vrhel (avrhel)
1/11/2017

Dig Dog: Constant Mining and Bloodhound

Motivation

Building a test suite for a software product is a time consuming and difficult process, yet it is vital for ensuring code quality and exposing defects. Randoop is a unit test generation tool that builds a JUnit test suite for Java code. Randoop uses feedback-directed random generation, which allows it to quickly build a massive suite of tests, but it falls short in some areas. Due to the random nature of the test generation, even thousands of tests are often not enough to get satisfying branch coverage, or expose bugs that develop from specific interactions or edge cases.

Guided Random Testing (GRT) is a variation of Randoop that includes 6 major enhancements, combining both static and dynamic analysis techniques to improve the quality of the generated test suites. However, GRT was not made open source, so these enhancements are something of a black box, and the open source Randoop tool does not get to see these benefits. We will attempt to implement two of the enhancements used in GRT, and examine whether they provide significant benefits to Randoop's performance.

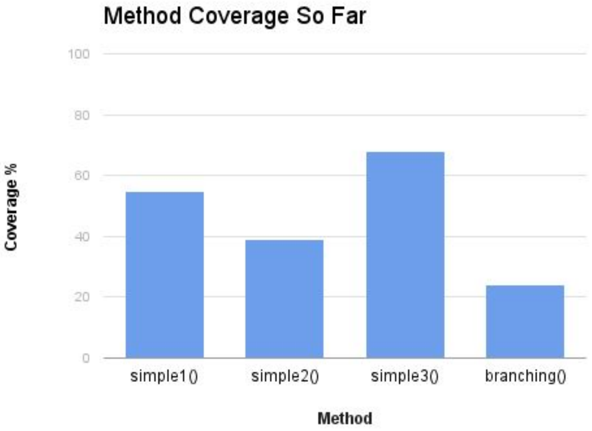
We believe that constant mining will help Randoop improve branch coverage and defect detection through increased selection of boundary values, and bloodhound will further improve branch coverage.

Approach

We plan on implementing two of the enhancements that were used in the Guided Random Testing variant of Randoop, constant mining and bloodhound.

Constant mining involves searching the code being tested for explicit primitive values and adding those to the initial pool of values used to generate test sequences. This improves Randoop's ability to find boundary values, making it more likely to expose off by one errors, and reach more branches of the code earlier in test generation. Roughly, code mining will be implemented by searching through the source code of the classes being tested, finding all primitive constants, and adding them to the pool of initial values when starting Randoop. They will be weighted by their frequency, so more commonly seen values are more likely to be chosen as inputs, and we can also distinguish between constants that are relevant locally (at a method or class level) and globally. A stretch goal for the project will be attempting to add some non-primitive values to the pool by finding newly initialized objects in the source code.

Bloodhound is the process of keeping track of code coverage and weighting the generation of new tests toward methods that are less completely explored so far. As in GRT, we will be keeping track of the execution of methods and re-weight each method's likelihood of being selected for a test sequence based on the branch coverage we have achieved in that method.



Our model will prefer selecting methods that have lower coverage, which should increase coverage overall. We will be attempting to keep track of branch coverage by instrumenting the source code to track which lines have been executed, but we have an alternative strategy (explained below) if that proves to be too difficult.

Challenges and Risks

The biggest challenge in improving constant mining will likely be the addition of non-primitive objects to the pool. It will be difficult to find the objects in the source code, and then once found to determine whether or not it should be usable as an input in the pool. It may or may not have been fully initialized after instantiation. One of the ways we might approach this problem would be to simply grab objects upon instantiation by looking for the "new" keyword, adding it to the pool, and then if tests consistently fail upon using the object it will be thrown out of the pool. It will be interesting to see if this addition to Randoop would be helpful or if it would produce enough false positives to degrade performance.

One of the challenges we might run into in implementing bloodhound would be approximating code coverage of different areas of the program. It is non trivial to calculate code coverage and doing it dynamically will be even more challenging. We believe this will involve instrumenting the code being tested which we are unfamiliar with. However, we do have some ideas for how we might approximate code coverage if instrumenting the code proves too difficult for the scope of the project. Randoop selects which methods to run as a part of each sequence, and we could keep track of how many times a method is run as a part of Randoops testing. By examining the source code and estimating the number of branches in a method, and comparing that to the number of times that method has been run by Randoop, we can estimate branch coverage and favor adding methods with lower coverage to the sequence.

Evaluation

We are planning on evaluating our enhancements of Randoop against Randoop itself and Quick Check. We will run these testing tools on defects4j (https://github.com/rjust/defects4j), a repository filled with known, intentional bugs that can be used to compare the quality of testing tools and strategies. Our goals in evaluating our Randoop enhancements is to see which approach is more beneficial and to obtain a quantitative comparison against existing tools.