2-Phase Coverage Annealing Random Testing

Milestone 1 Report for CS569: Static Analysis/Model Checking, Spring 2016

Nicholas Nelson

May 5, 2016

1 Strategy

Since we have only discussed Random Testing (RT), Breadth-First Search (BFS) Testing, and augmenting Random Testing using statement coverage information, I decided to abandom my previously proposed Adaptive Random Testing (ART) through Random String Generation concept. Instead, I chose to further refine the 2-phase coverage approach that we developed in class. The major alterations that I made to the algorithm were to have the phases alternate between each other using time-based heuristics in the case of phase 1, and annealing to a coverage threshold on phase 2.

2 Phase 1: Random Testing

Within Phase 1, my algorithm randomly selects actions from the set of enabled actions as determined by TSTL within the generated SUT file. These actions are added to tests up to the configured depth (option: <DEPTH>). This process is repeated until either the allotted time for Phase 1 has passed or the full algorithm has surpassed the configured timeout (option: <TIMEOUT>). The allotted time for Phase 1 adheres to the following heuristic:

$$timeout < 30 : 2 \times (\frac{timeout}{3})\ seconds$$
$$timeout \geq 30 : (\frac{timeout}{10})\ seconds$$

These heuristics allow the random testing pool that is generated in Phase 1 to gather enough tests to facilitate the filtering and annealing that occurs in Phase 2. Since my algorithm is setup to cycle between Phase 1 and Phase 2 until the allotted time has been reached, the improvements to Phase 1 also include rebuilding a new pool of tests when returning to this phase.

3 Phase 2

Whithin Phase 2, my algorithm filters the pool of tests generated in Phase 1 by pulling the bottom 10% of statements based upon coverage and generating a pool of all tests that contain these statements. Using the threshold determined for the 10% filter, and a coverage set for each of the low-coverage statements, my algorithm randomly selects tests to execute until the mean coverage across the set has surpassed the threshold. This means that tests with potentially interesting properties (i.e. low coverage indicating hard to reach statements) have been revisited to the point that a similar level as all other statements. Phase 2 continues this process until reaching a mean coverage across the set of equal to or greater than threshold, or the full algorithm has surpassed the onfigured timeout (option: <TIMEOUT>). The timing within this phase is variable, but overall respects the timing requirements to within a small margin of overrun.

4 Overall

My algorithm is meant to allow cycles of random testing (Phase 1), exploiting for coverage (Phase 2), and rebuilding all around a mapping of statement coverage across all portions of the program. On each success cycle of Phase 1 and Phase 2, new statements are singled out as being below the 10% threshold and therefore new tests are also pulled into pools to be exploited in Phase 2.

In brief testing of the algorithm, it was unable to locate the combination fault within the sample AVLTree file. However, I believe with an increase beyond 30 seconds it will have a better chance of locating this fault and others. The overhead of setting up, extracting, and annealing statement coverage and threshold values perform better when scaled up.