Project Report

Xuan he

Current Work

So far, I have implemented a basic random tester for testing the sut file created by TSTL. In the terster.py, firstly I receive six arguments from the command line using the command “sys.argv[i]”. The arguments are follows:

Timeout : Pointing out the max running time of the test in seconds.

Seed : Pointing out the seed for Python Random.random object used for random number generation in the code

Depth: Pointing out the maximum length of a test generated by your algorithm

Width: Pointing out maximum memory/BFS queue as a search width

Faults: Either 0 or 1 depending on whether your tester should check

for faults in the SUT; if true, I will save a test case for the discovered failure in the current directory, as failure1.test failure2.test, etc.

Coverage: Either 0 or 1 depending on whether a final coverage report should be produced.

Running: Either 0 or 1 depending on whether running info on branch

coverage should be produced.

The program uses the function “sut.randomEnabled(rgen)” to generate the random actions, and checks whether the action is correct by the function “sut.safely(action)”. When the bug is found, firstly it gets the reducing steps of the test by using the function “sut.reduce(sut.test(), sut.fails, True, True)”, then rewrites the function “sut.prettyPrintTest(R)” in order to record the faults case to the file by using file IO system and the function “sut.prettyName(s).ljust(80 - len(steps),' ')”. After that, it breaks out the loop and terminates the test. If Coverage is set, it will call the function “sut.internalReport()” to show the coverage information. If Running is set, it will call the

function “sut.newBranches()” and show the branches information.

Next Work

I already implement the base algorithm of the randomtester. Next, I will find some functions to improve the current random test algorithm. I will focuse on the EvoSuite. It is an automatic test suite generation for Java. It introduces a lot of algorithms for testing.

BST is based on heuristics that require frequent test execution, and can therefore become very inefficient if test execution time is high. On the other hand, DSE can be very efficient when applied on problems which the underlying constraint solver is able to address, yet its scope is much more limited. Despite tremendous recent progress, DSE may still struggle with floating point arithmetics, string datatypes, mixed constraints and sequences of function calls. Their approach of combining SBST and DSE is based on the ob- servation that DSE can be seen as a special case of *local* search. A local search algorithm explores the neighborhood of a candidate solution during the search, whereas a *global* search algorithm (e.g., a genetic algorithm) uses a population-based approach to explore larger parts of the search space.

The algorithm they describe is very interesting. In the future, I will do more research about that algorithm. Then, I will try to find something about that which can improve my random test algorithm. I will try my best to do that. If it is infeasible, I will try find other information to improve my algorithm as soon as possible.