**Project Report**

**Xuan He**

**Current Work**

So far, I have implemented an improvement random tester for testing the sut file created by TSTL, compared with tester1.py.

In tester2.py, the program splits to 2 phases.

In first phase, 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)”. Then it counts and records the current statement in the array “coverageCount” from the set “sut.currStatements()”. If the set “sut.newStatements()” is not empty, it puts the current test to the fullpool. 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 terminates the test. If Running is set, it will call the function “sut.newBranches()” and show the branches information. The running time of the first phase is 1/5 of the limted time “Timeout”.

In second phase, the program build an activePool that stores the statements whose coverage is less than the mean coverage among the fullPool. Then 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)”. After that it will execute the function “sut.replay(rgen.choice(activePool)[0])” randomly, which will cover one of the statement in the activePool. Then, it will continue do the test as same as the first phase.

At last, if Coverage is set, it will call the function “sut.internalReport()” to show the coverage information.

**Next Work**

I already implement the base improvement algorithm of the randomtester. Next, I will continue to find more 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.