Project Report 2

Zheng Zhou
Student ID: 932463162

May 17, 2016

1 Introduction

This report is for reporting the **second** version of my test generation algorithm. In my last proposal, I was intending to implement Adaptive Random Tester using TSTL. Adaptive Random Testing algorithm is quite effective and efficient for finding failures based on random testing. Nevertheless, I felt that it is hard to implement and improve Adaptive Random Testing using TSTL API. Thus I plan to consider another simple but also effective algorithm for this project. My brand new algorithm in first phase will base on breath first search which has been proven as a slow but comprehensive algorithm. Due to the slowness of BFS, some of the bugs in SUTs have to take hours even days to explore. What I am trying to improve is instead of going through all nodes in one level, I set a time limitation for each level (depth) to explore deeper with less time required in the state tree. This is going to be first phase of my algorithm. In first phase I will collect the coverage data and statement information for using in next phase. The second phase is going to be different from what I introduced in first version. Due to we have to satisfy the timeout parameter, I will not have chance to perform second phase that introduced in initial version with enough amount of time. Thus instead of using a separate phase to explore information, I was trying to collect coverage information after each depth done. In the meantime, the main loop will insert the state of new statement into the beginning of state queue and trying to track it again for more new things. In addition, I also used population mutation that introduced in class. I record the population and mutate it after each depth done. In this way I expected that I can collect more new statements information.

2 Implementation

For this Second version, what I changed in code is below the depth loop and in the middle of action loop. I added three variables which are coverageCount, leastCovered, population respectively. coverageCount is used to record the coverage information for current context. This will be used to compute the leastCovered statement which we mostly care about. After filling out coverageCount and get the least covered statement, program will use it to add the state of least covered statement or new covered statement (which is obviously least covered) into state queue. Thus next iteration will take these additional states into account. Besides this, population list is collected at the end of state loop. When each state is done, population will add the test that program have done and current branch number as a tuple in order to mutate. Program then randomly choose a population in list and mutate it. After mutate population, Program will append the mutated test.

3 Future Plan

The second version got sightly better coverage compared with first version with the parameter value that instructor provided. But it still does not work well on finding combination locks. And the coverage is comparatively lower than random tester as before. I think the possible reason is my algorithm can not find enough interesting information in small amount of time and relatively narrow width. In the final version I will try my best to explore as more coverage as possible to improve my algorithm.