CS 569 spring 2016 – Project

Final Report

Qi Wang

Onid: wangq5

Students ID: 932-439-151

June 6, 2016

Static Analysis and Model Checking for Dynamic

Section 1: Background

Random testing is a very good testing method to test programs. Since the random testing method is a black-box software testing technique[1], so the program uses the randomly generated inputs. It also called "fuzzing"[2]. Random testing is effective in industry testing development. In the most complex systems, the random testing could produce some inputs that are would not think to try for users. For the most programs, this testing approach is useful, but there are still some programs are not always found bugs when using the random testing, because this method could not make sure every possible inputs would be tested. So it generates and uses some meaningless cases to test the program in the limited testing, which will make the testing efficiency lower. What is more, some branches of the program are hard to run and others are not. So maybe in first half of the whole running time, some branches of the program are being tested for many times, however, the others are not. As the testing keeps doing, the situation might still as the same as the first half time of testing. So due to the random testing method, some branches might only be executed for once or twice, the others are tested for too many times. So the testing result might be not so efficient. Therefore, I design and implement a testing generation method based on the random test and the SUT algorithm to improve the random testing method.
So in the following sections, I will firstly introduce the algorithm and present some explanations of it. Then I will present the comparison between the former version and the final version of the tester.

Section 2: Algorithm and Implementation

On my novel algorithm of this project, I use the following steps to design my algorithm. First of all, set the parsing parameters of the program: timeout, seed, depth, width, faults, coverage, and running. So when running the program on the command line, use these parameters in the command line. Then initialize sut.sut(), random.Random(), time.time() of the program. Next is set the number of depth, savecoverage test. After

generating the random statement by calling the randomEnabled(), use this function to randomly find the enabled actions, and then check if they are safe actions. It collects the random statements. Then calculate the data coverage, to find the coverage of the statement is lower than the threshold of the coverage. Then storing the collected statements, if bugs are found, save the failures in the TSTL API. At last, printing out all found bugs and total runtime. In other words, it means the algorithm of the project aims to find the maximal coverage on the SUT(software, under test), which will make the test generation efficiency for finding bugs. So it focuses on the branch coverage, after saving the statements of the current test, find the least coverage in the current branches. Then running the random test and if there is any failures the tester will save it in a file named failurei.test in the directory. In the algorithm, there is a new list called population, which is used to store the tuple of the tests and statements. The list will be applied mutation function and crossover function. The population is the initial guess, for some SUTs the population is good, there will not many changes on the coverage after the mutation and crossover function. The crossover function can exchanges two tests and combine them to one test and return it. Also, in the inside part of crossover function, it still needs to check the faults. The mutation function merely takes a test, and changes it to get a new test. After these operations, it will append the results back to the list and doing the next iteration.

When running the testing generation, users can just run in the command line with parameters. First of all, put the sut.py file and finaltester.py file in the same directory. To run the test generation on the command line, use seven parameters in the command. The first number denotes the timeout that the testing generation will stop when the program runs this time. The second denotes seed, which is for python random.random object used for random number generation in the code. The third parameter denotes maximum length of the test generated. The next represents the maximum memory that is basically a search width. The next three parameters can be set in 0 or 1. The last third parameter means

the test generation will not check for faults in the SUT. So when testing the file should set it to 1 to find bug. The following two parameters mean that the final coverage report will be produced and also the branch coverage will be produced.

Section 3: Result

When testing the program, the SUT that I used to test is mostly AVL tree. We have used the AVL tree for a long time and we are familiar with this program, and result is not difficult to understand. In the first version of the tester file, after running the program in 40 seconds it cannot find any bugs and there are total 180 branches and statements. Total TSTL BRANCH COUNT is 180, total TSTL STATEMENT COUNT is 135, and 0 FAILED, 107500 TOTAL ACTION, and 40.0245230198 TOTAL RUNTIME. The result is shown as the following screenshot:

```
TSTL BRANCH COUNT: 180
TSTL STATEMENT COUNT: 135
0 FAILED
180 BRANCH
107500 TOTAL ACTIONS
40.0245230198 TOTAL RUNTIME
```

Then after revising the calculate method and the tester can save bugs as a separated files the tester could find failures and increase the number of branch and statements. The branches that my novel algorithm covers are slightly improved. Total TSTL BRANCH COUNT is 191, total TSTL STATEMENT COUNT is 141, there are 4 FAILED found. There are 7252 TOTAL ACTIONS, and 40.0069658756 TOTAL RUNTIME. The result is shown as the following image:

```
TSTL BRANCH COUNT: 191
TSTL STATEMENT COUNT: 141
4 FAILED
191 BRANCH
7252 TOTAL ACTIONS
40.0069658756 TOTAL RUNTIME
```

When setting the fault to 1, it could find failures of the program, and it will create the failure files in the current directory and the content is shown in the following:

```
self.p_avl[0] = avl.AVLTree()
self.p_val[0] = 1
self.p_avl[0].insert(self.p_val[0])
self.p_val[0] = 2
self.p_avl[0].insert(self.p_val[0])
self.p_val[3] = 3
self.p_avl[0].insert(self.p_val[3])
self.p_val[0] = 4
self.p_avl[0].insert(self.p_val[0])
```

I made a comparison between my novel algorithm and the random test generation. I set the same configuration: the timeout is 40, the seed is 1, the depth is 1000, the width is 100, the faults is 1, the running is 1, and the coverage is 1. After running the both programs, the result is shown as the following form. It shows that my novel algorithm can cover more branches and statements than the random test generation. It also could find some bugs in the limited time, and the other one is not.

	Random test	finaltester
branches	180	191
statements	135	141
actions	8949	11000
Running time	40	40.009577
bugs	0	7

Reference

[1] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. *Software Engineering,* 2007. ICSE 2007. 29th International Conference on, pages 621-631. IEEE, 2007.

[2] W.Howden. Systems Testing and Statistical Test Data Coverage. COMPSAC ’ 97, Washington, DC, August 1997. P.500-504.