Bowen Yu

932-439-028

6/6/2016

Final Report

Introduction

The random testing method is a black-box software testing technique, so the program uses the randomly generated inputs. It actually has many advantages. 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 I will implement a testing generation that could try to use fewer test cases to detect the software faults in TSTL. I will use Adaptive Random Testing, which is an enhanced form of random testing. Adaptive random testing seeks to distribute test cases more evenly within the input space. It is based on the intuition that for non-point types of failure patterns, an even spread of test cases is more likely to detect failures using fewer test cases than ordinary random testing. Experiments are performed using published programs. Results show that adaptive random testing does outperform ordinary random testing significantly.

Explanation

They main idea of this project is to improve tester1.py. I will implement a testing generator to do that. For this project, I firstly generate the random test, and I make some random action and check if it is safe. When it is safe, this part is the bellowed mean, so I will test that part again. The function will add some new command line to test. It also set

input to the parameters. Then testing generator will find the bugs and print them out.

Run the testing generation:

The user could run the test generation in the command line and write their only parameter, like timeout, seed, depth, width, faults, coverage and running. To run the test generation on the command line, use the following code: python tester1.py 30 1 100 1 0 1 1. The first number indicates when the program runs this time, the testing generation will stop. The second is seed, it is used for generating a random number for python random object. The third parameter represents the 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. The following two parameters mean that the final coverage report will be produced and also the branch coverage will be produced.

Then initialize the sut.sut(), random.Random(), time.time() of the program. Next is set the number of depth. After calling randomEnabled(),Using this function random find support operations, and check that they are safe action. It collects random statement. Then calculate the data reported, found the statement coverage is below the threshold reported. Then stored statements collected, if errors are found, save failed TSTL API. Finally, print out the bug and total run time of all discovered. It focuses on the branch coverage, save the current statement after tests found to cover at least in the current branch. Then run a random test, if there is any failure of testers will be saved in a file named failure i of. Testing in the directory.

When running the test generates, users can run the command-line arguments. First, sut. py file and final tester. py file in the same directory. At the command line to run test generation, use the command seven parameters. The first number indicates the test generation timeout will stop running. The second represents the seed, python random. Random objects used for random number generation code. The third parameter indicates the

maximum length of the test generation. The next represents the maximum memory is basically a search width. The next three parameters can be set to 0 or 1. Finally, the third parameter is the test generation SUT does not check for errors. When the test file should be set to 1 to find the bug. The following two parameters means that the final report will generate coverage and branch coverage will be generated.

Result

In the tester2.py, I add the function to accomplish the goal that the failures can be save in separate files. I use command: `python tester1.py 40 1 100 1 1 1 1` to test it. After running the program in 40 seconds it finds 6 bugs and there are total 191 branches and 140 statements. Total actions are 6722, and total runtime is `42.0286278725s.`

```
TSTL BRANCH COUNT: 191
TSTL STATEMENT COUNT: 141
6 Failures have been found
TOTAL ACTIONS 6722
TOTAL RUNTIME 42.0286278725
```

For the second version of my code, which is tester2.py. I changed the calculate method and add functions in order to make sure 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. I use command: `python tester2.py 40 1 100 1 1 1 1` to test it. The result is similar. After running the program in 40 seconds it finds no bugs and there are total 178 branches and 134 statements. Total actions are `107900`, and total runtime is `42. 40.0346910954s.`

```
TSTL BRANCH COUNT: 178
TSTL STATEMENT COUNT: 134
0 Failures have been found
TOTAL ACTIONS 107900
TOTAL RUNTIME 40.0346910954
```

It could find failures of the program and it will create the failure files in the current directory. I compared between my 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 fault is 1, the running is 1, and the coverage is 1. After running the both programs, 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.

References:

[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] T.Y. Chen, F.C. Kuo, H. Liu, and W.E. Wong. Code coverage of adaptive random testing. IEEE Transactions on Reliability, 62(1):226–237, March 2013. http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6449335&url=http%3 A%2F%2Fieeexplore.ieee.org%2Fiel5%2F24%2F6471782%2F06449335.pdf% 3Farnumber%3D6449335

[3] T.Y. Chen, H. Leung, and I.K. Mak. Adaptive random testing. http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.p df

[4] T.Y. Chen, F. Kuo, R. Merkel, and T.H.Tse. Adaptive random testing: The art of test case diversity. Journal of Systems and Software, 83(1):60–66, 2010.