Final report CS 569

Feedback-Directed Random Testing

Muhammad Bangash

INTRODUCTION:

When I presented the project proposal, I decided to go with Kernel Density Adaptive Random testing but after seeing its complexity I decided to go with Dynamic Analysis using Model checking and Static Analysis mentioned in "Feedback-directed Random Test Generation", which is raised by Pacheco, Carlos; Shuvendu K. Lahiri; Michael D. Ernst; and Thomsa Ball. The writers have found a new way to progress the efficiency of random testing and they have mentioned this in their paper. The techniques presented by the authors, they are improving random test generation by utilizing the response found from the input test cases which the generator has created before. I have tried to match what authors have tried to explain in their research.
Feedback-directed random test generation is a widely used technique to generate random method sequences. It leverages feedback to guide generation. However, the validity of feedback guidance has not been challenged yet. In practice I have tried to investigate the characteristics of feedback-directed random test generation and used a method that exploits the obtained knowledge that excessive feedback limits the diversity of tests. It shows that the feedback loop of feedback-directed generation algorithm is a positive feedback loop and amplifies the bias that emerges in the candidate value pool. This over-directs the generation and limits the diversity of generated tests. Thus, limiting the amount of feedback can improve diversity and effectiveness of generated tests. Second, used a method named feedback-controlled random test generation, which aggressively controls the feedback in order to promote diversity of generated tests.

ALGORITHM EXPLANATION:

- ### Tester1.py:

The algorithm I have used has three arrangements. 1. Non-error sequence, 2. Error sequence, and 3. new sequence. These error sequence and non-error sequence are being used to accumulate the cases that have been tested. The new sequence in algorithm is used to store the new case that generator creates. Starting from the beginning, the generator will create a test case and put it into new sequence. Secondly after checking the new test case, stored in the new sequence, has been tested before. When testing the new case has been tested, create a new case. Thirdly, now executing the new sequence and check the feedback. When the feedback has no error then store the new sequence into non-error sequence. If there is error in feedback, then store the new sequence into error sequence. When testing is performed and the feedback has errors then it must store the new sequence into the error sequence. My algorithm after taking references from different researches will overcome non-error-sense test cases. While going through the research the writers have

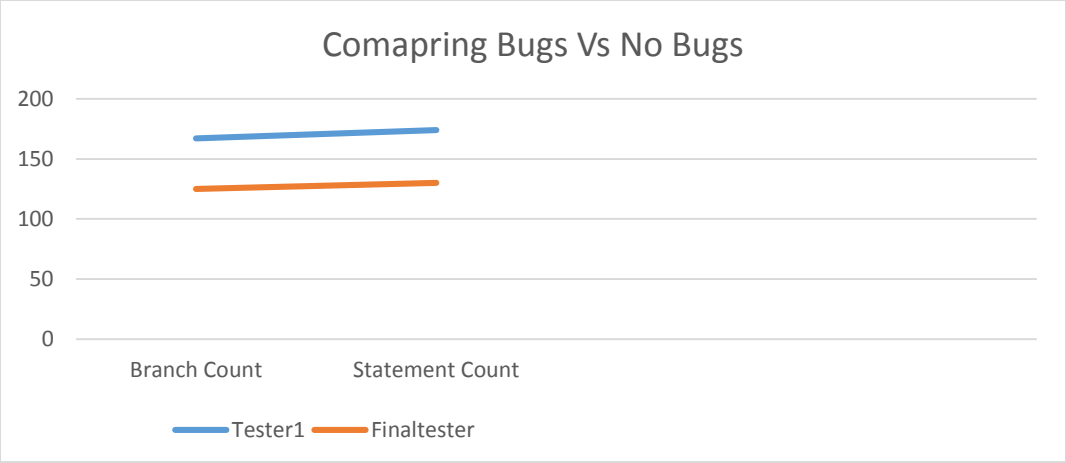
implemented an algorithm, it's a unit test generator preferably works over java. Its function is that it will automatically create unit test in a format known as j-unit. My Feedback-Random tester works in the same way and my random tester deals with SUTs providing higher efficiency as compare to others. My previous i.e. tester1 was the basic implementation it met all the necessary requirements. In my first tester (tester1.py) I had defined a function of parse_argthis function added command line arguments and sets input value using sys.argv to input into the parameter. I also implemented a function of make_config, this uses pargs and parser which works almost same as random tester. This function defines that it will return a dictionary after calling config.function. I did used another function check_action, this will get a random function using sut.randomEnabled and will also check if the action is correctly performed or not. It will give the required output if command line runs 1and if not the there is a bug and will output a failure massage. If everything works fine and line fault is 1 then is saves different failure messages into different file. When my algorithm generates a new statement, the algorithm will check whether if the statements have been tested. If statements have been tested, then generate a new statement. My tester1.py allow user give a time how long will the tester run and how many branch count and statement count it covers.

- **Tester2.py & finaltester.py:**

When I started my tester2.py, I focused on the aim to make my tester efficient to test bug and make a report file to address the bug and report it but I failed to do that. First of all, I have made some definitions for making configurations, to parse the arguments and to check the actions. The code looks simplified when the program is divided by definitions. Unlike when everything is mixed. Therefore, this is a good practice in programming. I have used the time library to measure the time elapsed. The sut is restarted after that and is backtracked. The for loop runs from value 0 to the depth and width. If the check action is not recorded in ok then sut.currStatements is accessed. After inserting the states, the sut.internal report is used to show the branch and statement coverage. My finaltester.py is an actual version of my code which I was trying to achieve.

RESULTS & IMPROVEMENTS:

My test failure was not detected, with that the test file was not created. Seeing the issues and code being crashed at a certain point. I have now resolved the issue by making some changes to my code. Test failure is now being detected and file is now created. So I changed the behavior of code and tried finding the bug and making the output failure report for it. For finaltester.py it worked pretty well results are shown below.



arsalan@ubuntu:~/Desktop/cs569$ python2.7 finaltester.py 30 1 100 1 0 1 1

With Bug:

TSTL BRANCH COUNT: 174

TSTL STATEMENT COUNT: 130

Running time: 30.0011839867

Bugs Found: 1

No. of actions: 79680

No Bug:

TSTL BRANCH COUNT: 167

TSTL STATEMENT COUNT: 125

Running time: 30.0011250973

Bugs Found: 0

No. of actions: 77530

TSTL INTERNAL COVERAGE REPORT:

/home/arsalan/Desktop/cs569/avl.py ARCS: 174 [(-1, 6), (-1, 11), (-1, 16), (-1, 31), (-1, 35), (-1, 44), (-1, 55), (-1, 70), (-1, 85), (-1, 111), (-1, 136), (-1, 148), (-1, 159), (-1, 171), (-1, 184), (-1, 233), (-1, 245), (-1, 254), (6, -5), (11, 12), (12, 13), (13, -10), (16, 17), (17, -15), (31, -30), (35, 36), (36, 37), (37, 39), (39, -34), (44, 45), (45, -43), (55, -54), (70, 72), (72, 73), (72, 75), (73, -69), (75, 76), (75, 78), (76, -69), (78, 79), (78, 80), (79, -69), (80, 81), (81, -69), (85, 87), (87, 89), (89, 90), (89, 95), (90, 91), (91, 92), (92, 93), (93, 104), (95, 96), (95, 98), (96, 104), (98, 99), (98, 102), (99, 104), (102, 104), (104, -84), (111, 112), (112, 113), (113, -106), (113, 114), (114, 115), (114, 123), (115, 116), (115, 119), (116, 117), (117, 118), (118, 119), (119, 120),

(120, 121), (121, 123), (123, 113), (123, 124), (124, 125), (124, 128), (125, 126), (126, 127), (127, 128), (128, 129), (129, 130), (130, 113), (136, 137), (137, 138), (138, 139), (139, 141), (141, 142), (142, 143), (143, -134), (148, 149), (149, 150), (150, 151), (151, 153), (153, 154), (154, 155), (155, -146), (159, 160), (159, 168), (160, 161), (160, 166), (161, 162), (162, 163), (163, 164), (164, 166), (166, -158), (168, -158), (171, 172), (171, 180), (172, 173), (172, 178), (173, 174), (174, 175), (175, 176), (176, 178), (178, -170), (180, -170), (184, 185), (184, 214), (185, 186), (185, 207), (186, 187), (187, 188), (187, 190), (188, 205), (190, 191), (190, 192), (191, 205), (192, 193), (192, 197), (193, 205), (197, 198), (198, 199), (199, 200), (200, 203), (203, 205), (205, 206), (206, -182), (207, 208), (207, 209), (208, 212), (209, 210), (210, 212), (212, -182), (214, -182), (233, 234), (234, 236), (236, 237), (237, 238), (238, 239), (238, 241), (239, -229), (241, 236), (245, 246), (245, 249), (246, -244), (249, 250), (250, 251), (251, -244), (254, 255), (254, 257), (255, -253), (257, 258), (258, 259), (259, 260), (259, 262), (260, 259), (262, 264), (264, 265), (265, 266), (265, 268), (266, 265), (268, -253)]

Mytester:

I have put an experiment in my code and I tried to generate a random number in the limit of 0 till action count. Then in result I got actual count and for complex count I divided it further by number of bugs found.

Result:

TSTL BRANCH COUNT: 167

TSTL STATEMENT COUNT: 125

####### mytester.py ######

Running time: 30.0013329983

Bugs Found: 0

No. of actions: 76164

Random count is 5525

Actual count 3

Complex count 0

EVALUATION:

To evaluate y finaltester.py, I tried to find the unknown bugs through my tester but it is only capable of finding known bugs only which I think is a drawback with my tester. Although with comparison to my previous tester code my finaltester is capable of finding known bugs and report it.

REFERENCES:

[1]. P. S. Loo and W. K. Tsai. “Random testing revisited”. Information and Software Technology, Vol.30. No 7, pp. 402-417, 1988.

[2]. T. Y. Chen, H. Leung, and I. K. Mak, “Adaptive random testing” in Proceedings of the 9th Asian Computing Science Conference, LNCS 3321, pp. 320-329, 2004.

[3]. T.Y. Chen, F.-C. Kuo, R.G. Merkel, and T.H. Tse, “Adaptive random testing: the ART of test case diversity,” Journal of Systems and Software, vol. 83, no. 1, pp. 60–66, 2010.

[4]. Pacheco, Carlos; Shuvendu K. Lahiri; Michael D. Ernst; Thomas Ball (May 2007)."Feedback-directed random test generation". ICSE '07: Proceedings of the 29th International Conference on Software Engineering (IEEE Computer Society): 75–84.