CS 569
Final Report
06/06/2016
Hao Wang (wanghao@oregonstate.edu)

Implementation of Feedback-directed Random Test in TSTL

1. Background

Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method of test can be applied to virtually every level of software testing, unit, integration, system and acceptance. It typically comprises most if not all higher level testing, but can also dominate unit testing as well[1]. For Black-box testing, testers do not need to know the process of implementation, or focus on source code. The high generality of method and low requirements of testers lead to a high concurrency and collaboration of the whole project process, which is the core idea of software engineering.

Random testing is a black-box software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is pass or fail[2]. As an effective method of black-box testing, it is a complement of white-box testing to garantee the completeness of the test process. However, random testing is not a good enough method sometimes since its inherent shortage. Currently, there is a number of modifications that work with the shortage of random testing.

2. Algorithm and Implementation

I get the idea of algorithm from the paper "Feedback-directed random test generation"[3], and the code skeleton from the given randomtester.py and newCover.py[4]. My algorithm is a modification of random testing. Instead of just creating inputs randomly, this algorithm builds 3 method sequences to store the methods. These are error sequence, non-error sequence, and new sequence. Error sequence stores the cases that have been tested right. Non-error sequence stores the cases that have been tested wrong. New sequence stores the cases that are new random generated. Afterwards, do the following steps:

1. Randomly generate a test case and put it in the new sequence;
2. Check whether this case is in error or non-error sequence, if yes, go to step 1, otherwise go to step 3;
3. Test the case and put it in the error or non-error sequence based on its result;
4. Go to step 1 unless the time is out.

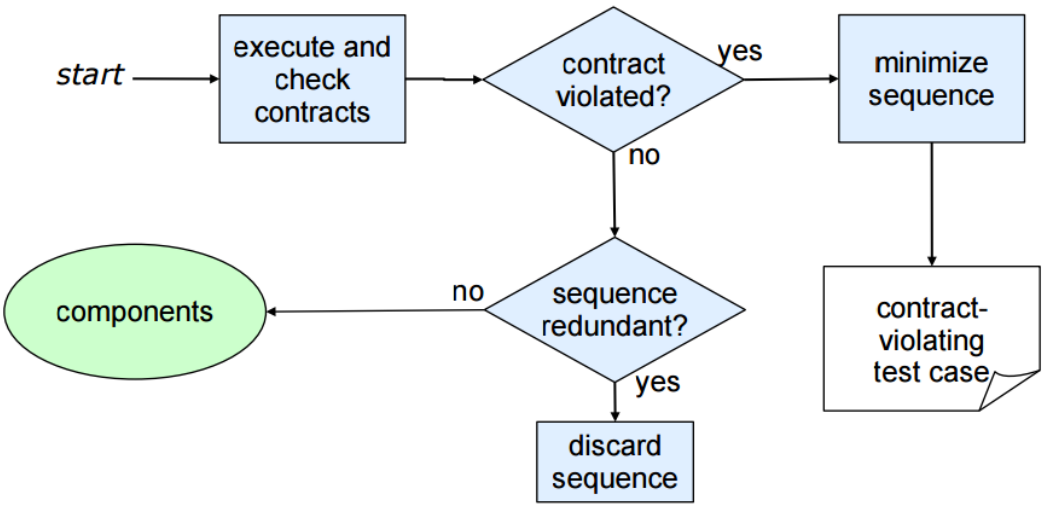
It is good to show a flow chart for a clear understanding as follows,

Comparing with the normal random testing, there are at least 2 advantage of this modification. First, it saves time since it stores the result of tested cases, so it will not

redundantly repeat former tested cases. Second, it will generate a higher coverage of all possible cases in a limited time, because random test would most probably repeat same cases multiple times.

3. What I have done and havn't

In the original paper [3], the whole algorithm can be simply described as the following flow chart:

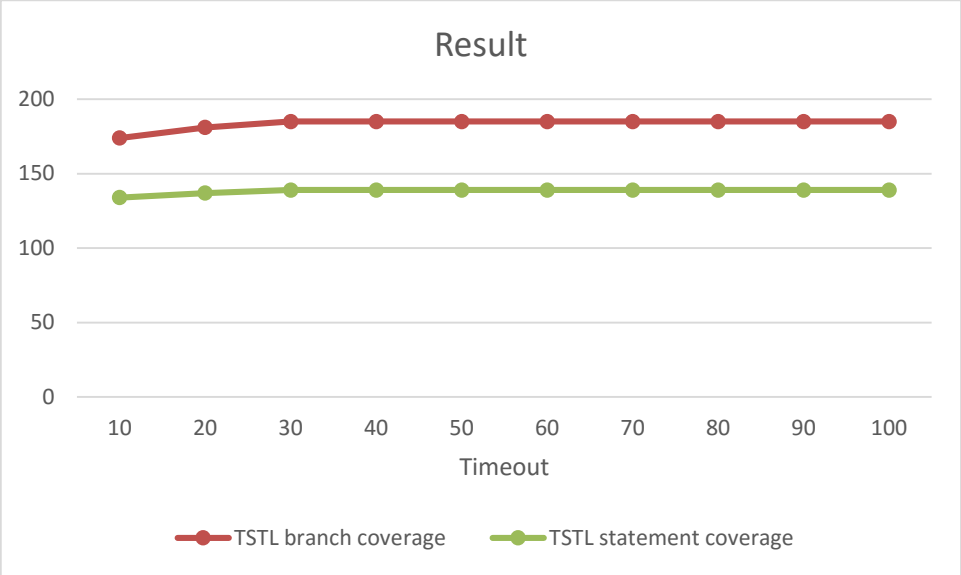


Graph 1: the process of an entire feedback-directed algorithm

I have completed almost all steps but discard the redundant sequence. The feedback-directed algorithm is nearly 85% completed. I think the left part is the most difficult part that could not be implemented easily in TSTL. I've tried multiple times since Milestone2, but it either returns error, or doesn't show any improvement comparing to the algorithm that doesn't have that part. Therefore, I just leave it to be my future research.

4. Result and Analysis

The coverage of my finaltester.py is shown below



Graph 2: coverage vs timeout

From the graph 2, we can see the feedback-directed random testing starts at a very high level of coverage when the timeout is very low, and maintains in a certain number while timeout is increasing. I think the maximum coverage is the limitation of the random testing and all random testing modification. Feedback-directed algorithm only same the testing time, but cannot increase the maximum performance of random testing.

However, pure random testing cannot generate such a high level coverage in a low timeout. It is because pure random testing does not record the previous test case. In this case, some testing might run multiple times but some others might not run. It is why testers cannot obtain a high coverage in a low timeout. On the contrary, feedback-directed algorithm stores the previous test case. It makes sure that every test case runs only one time. Thus the algorithm does more work than pure random testing. It is why feedback-directed algorithm starts at a very high coverage when the timeout is very low.

To compare with other algorithms (e.g. Swarm), my algorithm still has the advantage of high coverage in low timeout. But some algorithms generate a higher maximum coverage than mine. In practice, testers can choose the proper algorithm for use.

5. mytester.py

To compare with the original finaltester.py, I add 3 parameters: maximum error sequence element number, maximum non-error sequence element number and maximum new sequence element number. In the original finaltester.py, the 3 sequence element number could be unlimited if the test case number is unlimited. But in mytester.py, I add a limit of the 3 sequence element number in order to save the resource when the test case number is too huge. The main idea is when the sequence is beyond the maximum number, the first stored test case will be dropped. Testers can use this to use some more time for saving space. It could be useful in some specific cases.

To run mytester.py, just add 3 more parameters than to run the finaltester.py, for

example:

python finaltester.py 20 1 100 1 0 1 1

python mytester.py 20 1 100 1 0 1 1 100 100 100

Notice that the 3 parameters should not be too small, or the algorithm would run no other than the pure random testing.

Reference

[1]. S. Debnath. *"Mastering PowerCLI"*. Packt Publishing Ltd. 2005. pp.35.

[2]. R. Hamlet (1994). *"Random Testing"*. In J. J. Marciniak. *"Encyclopedia of Software Engineering"*. John Wiley and Sons. 2013.

[3]. C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball. *"Feedback-directed random test generation"*. In *"Proceedings of the 29th International Conference on Software Engineering"*. 2007. pp.75-84.

[4]. "https://github.com/agroce/cs569sp16"