Course: CS569
Name: Yanshen Wang
Date: 06/06/16
Final Report

Random Test Generation in TSTL

1 Introduction

Right now, random testing has been widely used for detecting bugs from all kinds of different software applications in case of its ease of implications. However, some theoretical experts argue that even through the random testing is very convenient to use, in the practical works it is not very efficient as people imagined. For example, some interesting and specific tests cannot be generated randomly, and moreover the random test generation will get a lower level of code coverage than the systemic testing in case of the modeling checking, symbolic execution, chaining and so on. Therefore, the Feedback – directed random test is a good technique to improve the random generation because this method can provide some feedbacks for the created inputs to check out whether or not they are suitable for the latter function calls.

2 Algorithm

GenerateSequences(*classes*, *contracts*, *filters*, *timeLimit*)

```
1  errorSeqs ← {} // Their execution violates a contract.
2  nonErrorSeqs ← {} // Their execution violates no contract.
3  while timeLimit not reached do
4     // Create new sequence.
5     m(T_1 ... T_k) ← randomPublicMethod(classes)
6     ⟨seqs, vals⟩ ← randomSeqsAndVals(nonErrorSeqs, T_1 ... T_k)
7     newSeq ← extend(m, seqs, vals)
8     // Discard duplicates.
9     if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
10       continue
11    end if
12    // Execute new sequence and check contracts.
13    ⟨o⃗, violated⟩ ← execute(newSeq, contracts)
14    // Classify new sequence and outputs.
15    if violated = true then
16       errorSeqs ← errorSeqs ∪ {newSeq}
17    else
18       nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
19       setExtensibleFlags(newSeq, filters, o⃗)  // Apply filters.
20    end if
21 end while
22 return ⟨nonErrorSeqs, errorSeqs⟩
```

According to the article "Feedback-directed Random Test Generation", authors give an example to show how this algorithm works to improve the quality of the random test generation. From the first line, we can see that the Generate Sequences function includes four inputs include a list of class creating the sequence, a list of contracts, a list of filters and the time out. Among those four parameters, the contracts and filters can be generated by the RANDOOP, which is a framework used to automatically generate unit test cases, so I think in my project I need to use the PYUNIT (it is similar with RANDOOP, but used in the python) to instead of it. In this algorithm, each sequence has associated with the Boolean vector since each value has a Boolean flag, and those flags indicate to show whether or not this value is legal to create a new sequence. There are two kinds of sequences: the *error Sequences* get the list which violates the contract, and the *none-error Sequences* get the list which

does not violate the contract. Then there is a loop with a time limit, and in this loop at the beginning we need to select a method *m* from the function *Random-Public-Method (classes)*, and then we need to use the lists of the sequence and the lists of value to recall this operator. In addition, the helper function *random-Sequences-And-Values* can incrementally generate a list of sequences and a list of values, and put them into the <*seqs*, *vals*>. After we implement the operator extend which used the *m*, *seqs*, and *vals* as the inputs, we can generate the new sequence. Next, we can use the function *execute (newSeq, contracts)* to check whether or not each sequence violates the contracts. The result of *execute* return a pair looks like this <*o*, *violated*>, and the *violated* is a Boolean flag which indicates that if at least one contract has been violated, the sequence will be added to the *error Sequences*, otherwise it will be added to the *none-error Sequences*. Finally, in *none-error Sequences*, we will use the *filter* function to check which values of sequence can be used as the parameter in the new function calls.

3 Command line arguments

In this project, when we implement the program, the command line has 7 input parameters which including the timeout, seed, depth, width, faults, coverage and running. The following context is the explanations of those parameters:

<*timeout*>: this parameter is used for restricting the time of the testing generation, so if the running time reaches to the number of time limits, the program will be stopped automatically.

<*seed*>: this parameter is the number of Python *Random.random* object used for random number generation.

<*depth*>: this parameter is the maximum length of a test generated.

<*width*>: this parameter is the maximum width of a test generated.

<*faults*>: this parameter will be either 0 or 1 depending on whether the tester should check for faults in the SUT. If true, it should save a test case for each discovered failure; if false, the test generation does not find the bug in the SUT.

<*coverage*>: this parameter will be either 0 or 1 depending on whether a final coverage report should be produced, using TSTL's *internalReport()* function. If true, it will print the TSTL internal coverage report; if false, it will not print the report.

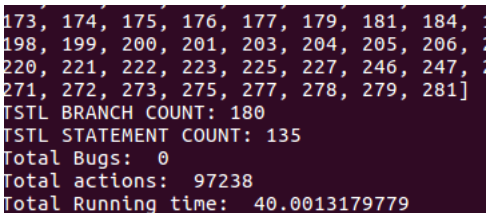
<*running*>: this parameter will be either 0 or 1 depending on whether running info on branch coverage should be produced. If true, it will print the above information; if false, it will not do anything.

4 Improvement

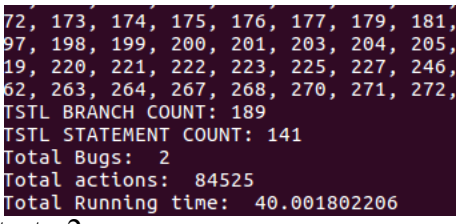
In this project, I changed the insert method of new state position, which makes the code looks more efficient. For example, in my previous code, it cannot find any bugs in 40s.

However in my new code, it can find 2 bugs, which can prove the new code is more efficient than the old one.

My test command looks like this:
python tester2.py 40 1 100 1 1 1 1



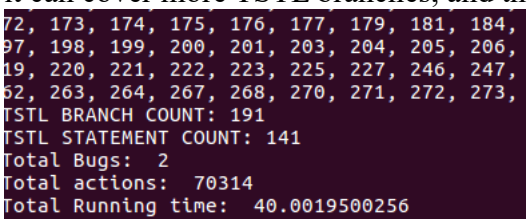
tester1.py



tester2.py

5 Final work

In my final project, I optimize a little bit about the structure of my program, so right now it can cover more TSTL branches, and the result will be displayed in the following image:



6 Summary

In this class I am very happy to learn the TSTL language, and I think this is a very convenient and useful tool to test other programs. Even through in the whole process of this project, there are lots of crashes, or bugs happened when I was writing my code, at the end of this term, I successfully utilize the python language to write a program which can more efficiently generate random testers. Personally, I think this class help me better understand the TSTL language, and how to write a random tester file using TSTL, so I believe this kind of testing language must be welcomed warmly by more program testers.

References

[1] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. IEEE TSE, 16(12):1402–1411, Dec. 1990.

[2] J. W. Duran and S. C. Ntafos. An evaluation of random testing. IEEE TSE, 10(4):438–444, July 1984.

[3] Pacheco, Carlos, et al. "Feedback-directed random test generation." Software Engineering, 2007. ICSE 2007. 29th International Conference on. IEEE, 2007.

[4] R. Ferguson and B. Korel. The chaining approach for software test data generation. ACM TOSEM, 5(1):63–86, Jan. 1996.