Implementation of Feedback-directed Test Generation on TSTL

Zixuan Zhao
Oregon State University

Introduction

1.1 Purpose

This report describes the project, implementation of Feedback-directed Test Generation on TSTL. TSTL is a test tool for python. TSTL provides a random test generator randomly producing test cases and executing these test cases. This project aims to find more potentials of TSTL by implementing various test generator algorithms. In this project, it concerns on applying Feedback-directed Test Generation algorithm on TSTL and evaluating this implementation.

1.2 TSTL

TSTL is a test tool for python. It has its own characteristics. TSTL mainly work with these elements: pool, actions, and properties.

- Pool: Define the value and available number of the value used in test.
- Action: Define the behaviors to test, such as initializing variables in pool, or calling functions of test object, like insert() function in avl.py[1].
- Properties: Define the rules that the test needs to obey.

Specifically, action is the key to TSTL itself since it is the component of test case. After defined those elements, TSTL can generate a sut.py file providing functions for selecting actions and executing them. User can import this sut.py as module and write their own test generator.

1.3 Feedback-directed Test Generation Algorithm

The mechanism of Feedback-directed Test Generation algorithm is building up test cases by incrementally randomly selecting functions and finding related arguments existed in previous test cases. Once we create a test case, a sequence of functions, we can execute it and judge whether it is failed, redundant, or good to consistently increase sequence of functions.

Algorithm Design

2.1 Idea

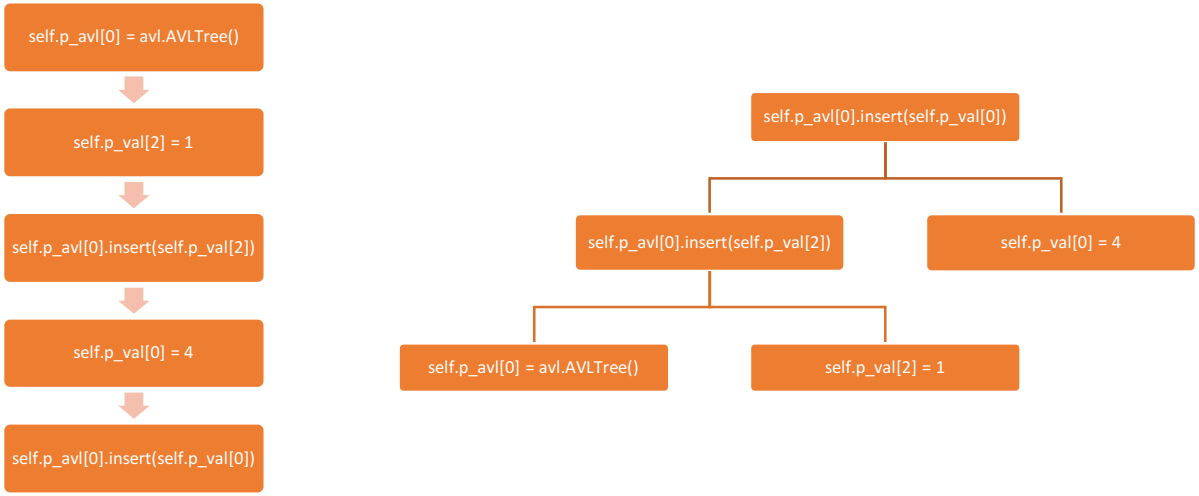
The actions can be classified to two types: one is for initializing the value for a variable in pool, I call it variable-action; the other one is called function-action. Function-action usually is related with testing function, and functions may need to work with object and argument. Thus, before executing all the corresponding variable-actions for a function-action, we cannot select this function-action.

That is the interesting part for TSTL. In this case, we can say the mechanism of TSTL is to select arguments first then the available function call. This is different with the technique of Feedback-directed Test Generation.

To adapt to the system of TSTL, the idea of implementing Feedback-directed Test Generation algorithm on TSTL is as following:

1. Randomly pick up one action and execute it.
2. If it is not failed and follows the property:
    a. If there is enabled function-action, select one of function-actions and execute it.
    b. Otherwise, randomly pick up one action and execute it.
3. Repeat step 2.

[1] This file can be find https://github.com/agroce/tstl/tree/master/examples/AVL



(a) *Feedback Test Generator under TSTL* *(b) Original Feedback Test Generator*

Figure 1 The Process of Both Algorithms for the same . (a) is the implementation of Feedback Test Generator algorithm on TSTL. It just sequentially run the available test one by one. (b) is the original Feedback Test Generator. It picks up the function first, and find the proper argument for this action.

2.2 Implementation

By following the idea in 2.1, I have this pseudocode:

```
1   Initialize passPool = [sut.state()], failedPool = []
2   While not TIMEOUT:
3       Restart a new test
4       Randomly select one sequence from passPool
5       Recovery the test of this sequence
6       Randomly set the increased sequence length as num
7       Select one action by sut.randomeEnabled() and save to variable action
8       While total new selected actions' number =< num:
9           Execute action
10          If there is error or not following property
11              Save current sut.state() to failedPool
12          If there is enabled function-actions
13              Randomly select one from these enabled function-actions and save to action
14          Otherwise
15              Select one action by sut.randomeEnabled() and save to variable action
16          If no error and following the property
17              Save current sut.state() to passPool
```

In this implementation, passPool is used to store the sequence of actions that does not have errors. On the contrast, failedPool will store the sequence of actions failed the test. These two pools save the state of the test so that I can easily recovery to the test of selected sequence.

Specifically, from line12 to 13, this somehow reduce the redundant actions. When the variable-actions are ready, program immediately check whether there are available function-actions and select one. It avoids program consistently generate repeated variable-actions.

2.3 Parsed Arguments

There are some parsed arguments for generating test cases. Argument timeout, depth and length may be helpful for uses to get better test cases.

Table 1 Parsed Argument Information

Argument Name	Description	Note
Timeout	Timeout in seconds (default = 60).	
Seed	Random seed (default = 10).	
Depth	Maximum length of a sequence (default = 100).	
Width		Not used
Length	Maximum increased length of a sequence (default = 20. No more than Depth is better).	Only in mytester.py
Faults	Store failure information (default = True).	
Coverage	Produce a coverage report at the end (default = True).	

Evaluation

In this section, I would like to evaluate the implementation itself. Also, I will compare it with randomtester.

The following tests are under this environment:

Operating System: OS X 10.11.4

Processor: 2.7 GHz Intel Core i5

Memory: 8 GB 1867 MHz DDR3

Language: Python 2.7 (OS default)

Test Object: avl.py, avl.tstl

All data are tested by mytester.py, since it contains more flexible parsed arguments.

3.1 Arguments Effect

Since there are three parsed arguments may affect the performance of this implementation. I would like to observe their relationships.

1. Timeout VS Length

   Range of Timeout: 10, 30, 60, 120, 200, 300

   Range of Length: 10, 20, 50, 100, 200

   By input above values one by one, I collect the following data. In the testing, most of data are very stable.

Table 2 The Information for Timeout VS Length

Branch Count						Statement Count					
	10	20	50	100	200		10	20	50	100	200
10	185	185	185	187	181	10	139	139	139	140	137
30	187	185	185	187	185	30	140	139	139	140	139
60	187	185	187	185	187	60	140	139	140	139	140
120	187	187	191	187	185	120	140	140	141	140	139
200	187	191	187	191	187	200	140	141	140	141	140
300	189	187	187	185	185	300	141	140	140	139	139

As we can see that with the increasing of timeout, the counts are all increased. However, for length, when it reaches to some point, the counts start to reduce. By column Length = 200, that when the length is too large, it affects a lot to the testing.

Specifically, for the (Branch/Statement) = (191/141) case, it means program find the error.

2. Timeout VS Depth
   Range of Timeout: 30, 60, 120, 200
   Range of Depth: 50, 100, 300

Table 3 The Information for Timeout VS Depth

Branch count				Statement Count			
	50	100	300		50	100	300
30	185	187	185	30	139	140	139
60	187	187	187	60	140	140	140
120	187	187	187	120	140	140	140
200	185	187	189	200	139	140	141

From this table, we can see that under most of cases, the depth does not affect testing a lot. Meanwhile, with a large number of depth, the result could be better.

3.2 Randomtester VS. Mytester

Timeout is the only same argument in both program. I will compare them with different timeout.

Table 3 The Information for Randomtester VS Mytester

	Timeout	10	30	60	120	200	300
Random Tester	Branch Count	187	187	187	187	187	187
	Stmt Count	140	140	140	140	140	140
Mytester	Branch Count	185	185	185	187	191	187
	Stmt Count	139	139	139	140	141	140

Random Tester works stable but it cannot find any bugs. For mytester, when Timeout=120, it found out one bug.

Summary

In my opinion, this implementation reaches my expectation. By Evaluation Section, we can see that this implementation of Feedback Test Generation can find the error. However, it may take longer time to find bugs. At beginning of the test, test generator frequently picks up short sequence and adds a few of new actions to the sequence. It cannot generate very long sequence in short time unless add a huge number of new actions to the sequence. In this case, it is better to test longer time to ensure whether there is error.

What's more, it is hard to determine the value for parsed arguments. I would recommend users to try more combinations of values so that testing can produce better test case.

Reference

Carlos Pacheco , Shuvendu K. Lahiri , Michael D. Ernst , Thomas Ball, Feedback-Directed Random Test Generation, Proceedings of the 29th international conference on Software Engineering, p.75-84, May 20-26, 2007