

Improved Random Test Generation using TSTL APIs

Thang Hoang

May 3, 2016

1 Project Description

1.1 Motivation

Random testing has been intensively adopted to detect potential failures in software due to its simplicity, efficiency and ease of implementation [2] compared with other complicated testing techniques [1]. Despite its surprisingly effective production of error-revealing test cases, random testing sometimes might generate a numerous of test cases which are unreachable, illegal or duplicated with others, and therefore limit its effectiveness. Specifically, based on my observation, implementing purely random testing (e.g., Breadth first search random testing) using TSTL APIs to test Python software frequently generates irrelevant actions that cannot cover every single lines of the code in the program. Meanwhile, such unreachable lines may contains potential bugs. Random testing cannot cover these lines as test cases are generated randomly and independent to each other.

1.2 Contribution

I propose a novel test generation algorithm using TSTL APIs. My approach is an improvement of random testing, in which subsequent random actions are selected regarding to SUT state being observed to cover and explore new statements as many as possible.

Main idea. The main objective of my strategy is that given a budget of time t and a depth value d representing the length of random actions being performed, *the frequency distribution of all statements in the program being scanned should be uniform*. Main intuition of my test generation algorithm is as follows:

First, I spend t/k time from the total time budget to perform purely random testing in phase 1 to scan preliminary statements being easily covered. d consecutive random accesses are performed in this case to explore new statements. For each time when new statements being found, I store its information as well as the state that can generate to it in two different arrays. After d consecutive random accesses, I measure the scanning frequency of each statement being covered.

Next, I calculate mean μ and standard deviation σ of frequency of all statements to determine statement outliers as:

$$\begin{aligned}\mu &= \sum_{i=1}^N f_i \\ \sigma &= \sqrt{\frac{\sum_{i=1}^N (f_i - \mu)^2}{N - 1}}\end{aligned}\tag{1}$$

Algorithm 1 Improved random testing using TSTL APIs

Phase 1. Random exploring

```
1: while TIME_BUDGET_FOR_PHASE_1 do
2:   for  $depth = 1$  to  $d$  do
3:     random_action()
4:     if new statements  $\vec{o}_i$  are found then
5:       store sut.state()  $s_i$  in  $\vec{s}$ 
6:       store  $\vec{o}_i$ 
7:     end if
8:   end for
9:   Calculate frequency  $f_i$  of each statement  $s_i$  being found
10: end while
    frequency analysis
11:  $\mu = \sum_{i=1}^N f_i, \sigma = \sqrt{\frac{\sum_{i=1}^N (f_i - \mu)^2}{N-1}}$   $\triangleright$  mean and standard deviation
12:  $\tau = \mu - c\sigma$   $\triangleright c = \frac{2}{3}$ 
13: Select states  $s_i$  that generate statements whose frequency less than  $\tau$ :  $\vec{s}' = \{s_i\}$ 
    Phase 2: Exploit more important states
14: while TIME_BUDGET_FOR_PHASE_2 do
15:   for each  $s_i$  in  $\vec{s}'$  do
16:     Assign  $t'$  time budget to exploit this state.
17:     while  $t' > 0$  do
18:       for  $depth = 1$  to  $d$  do
19:         random_action()
20:         if new statements  $\vec{o}_j$  are found then
21:           insert sut.state()  $s_j$  to the position right after  $s_i$  in  $\vec{s}'$ 
22:         end if
23:       end for
24:     end while
25:   end for
26: end while
```

```

def insert(self, key):
    tree = self.node
    if (key == 1) and (self.evilstate == 0):
        self.evilstate = 1
    elif (key == 2) and (self.evilstate == 1):
        self.evilstate = 2
    elif (key == 3) and (self.evilstate == 2):
        self.evilstate = 3
    elif (key == 4) and (self.evilstate == 3):
        self.evilstate = 4
    elif (key == 5) and (self.evilstate == 4):
        key = key / 0
        self.evilstate = 5
    else:
        self.evilstate = 0

```

where f_i is the frequency of statement s_i being covered.

I denote statement outliers as \vec{o} based on a threshold τ which is defined as:

$$\tau = \mu - c\sigma \quad (2)$$

So, $\vec{o} = \{s_i | f_i \leq \tau\}$ is a collection of statements whose frequency is less than or equal to τ .

After determining \vec{o} , I select states of SUT that explored such statements in the first phase ordered by least coverage statement. Finally, I perform random testing of these states attempting to explore new statements. I assign time budget for each state regarding to the statement that it covers in the first phase. For instance, states that explore least statement will be assigned most time budget and so on. For each state being exploited, I also call d successive random actions and continuously measure the scanning frequency of all statements as in the first phase. If a new statement is being found, the state that explores this statement will be inserted to the array so that it will be exploited in the next loop. Our intuition behind this is that, it is more important to exploit the state that explore new statements, but still remain time budget for other less important states. This differs from `coverTest.py` in which, only the state of the least statement is exploited. Algorithm 1 presents my idea in detail.

I test my algorithm on an AVL tree program with bugs located at some positions. For example, It can find a bug located at the fifth-level of a if statement as in Figure 1 within 30 seconds allocated and in Phase 2 with the parameters as “tester1.py 30 1 100 20 1 1 1”. The number of coverage is 194 and number of statements is around 132, reported by `internalReport()` function.

References

- [1] Alex Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 621–631. IEEE, 2007.
- [2] Richard Hamlet. Random testing. *Encyclopedia of software Engineering*, 1994.