Improved Random Test Generation using TSTL APIs

Thang Hoang

May 18, 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 $t/4$ 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 determine k statements which is least covered as $\vec{o}$ based on a threshold τ which is defined as:

$$\tau = f_l \tag{1}$$

where f_l is the k-least frequency. So, $\vec{o} = \{s_i | f_i \leq \tau\}$ is a collection of k 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

Algorithm 1 Improved random testing using TSTL APIs

Phase 1. Random exploring

while `TIME_BUDGET_FOR_PHASE_1` **do**
  for *depth* = 1 to d **do**
    random_action()
    if new statements $\vec{o_i}$ are found **then**
      store sut.state() s_i in $\vec{s}$
      store $\vec{o_i}$
    end if
  end for
  Calculate frequency f_i of each statement s_i being found
end while
Sort the frequency with ascending order, resulting in $\mathbf{f'}$
$\tau = f'[k]$
Select states s_i that generate statements whose frequency less than τ: $\vec{s'} = \{s_i\}$

Phase 2: Exploit more important states

while `TIME_BUDGET_FOR_PHASE_2` **do**
  for each s_i in $\vec{s'}$ **do**
    Assign t' time budget to exploit this state.
    while $t' > 0$ **do**
      for *depth* = 1 to d **do**
        random_action()
        if new statements $\vec{o_j}$ are found **then**
          Replace s_i in $\vec{s'}$ by sut.state()
          sut.replay(s_i)
          depth = 1
        end if
      end for
    end while
  end for
end while

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 replicated the current and d new successive random actions will be assigned to this state.. 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 improved algorithm on an AVL tree program without bugs to determine the maximum number statements and branches being covered. It show a little bit improvement over the previous version in that the improved algorithm can cover 185 branches with 138 statements, compared with 183 and 137 from the previous version.

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.