CS 569 Final Report

Improved Random Test Generation in TSTL APIs

Yu-Chun Tseng

June 6, 2016

1. Background

Random testing is a black-box software testing, also known as monkey testing, where programs are tested by generating random and independent inputs. Results of the output are compared against software criterions to verify that the test output is success or failure. In case of lacking of criterions, the expectations of the language are used that means if an expectation appears during test execution then it shows there is a bug in the program. However, random testing only finds basic bugs (e.g. Null pointer dereferencing) and is as precise as the criterion and criterions are basically imprecise. Moreover, according to the in-class experiment, we can find that utilizing pure random testing, such as BFS, in TSTL to test Python programs often causes unrelated operations that cannot lay over every code line of the programs. Such portion of the code that we cannot cover could have bugs.

2. Approach

The basic idea is that an object-oriented unit consists of a sequence of method calls that set up state, such as creating and mutating objects, and an assertion about the result of the final call. I define an extension operation that takes 0 or more sequences and produce a new sequence. Extension is the essential operation of my algorithm. The extension operation builds a new sequence by concatenating its input sequences and appending a method call at the end. Formally, the operator *extend(m, seqs. vals)* takes 3 inputs:

- **m** is a method with formal parameters of type $T_1, \ldots ,T_k$.
- **seqs** is a list of sequence.
- **vals** is a list of values v_1: T_1, …, v_k: T_k. Each value is a primitive value, or it is the return value $s.i$ of the i-th method call for a sequence in *seqs*.

The result of *extend(m, seqs. vals)* is a new sequence that is the concatenation of the input sequences *seqs* in the order that they appear, followed by the method call m($v_1, \ldots , v_k$).

3. Algorithm

The algorithm is as follows.

```
GenerateSequences(class, contracts, filters, timeLimit)
errorSeqs ← {} // Their execution violates a contract.
nonErrorSeqs ← {} // Their execution violates no contract.
While timeLimit not reached do
    // Create new sequence.
    m( $T_1$ ... $T_k$ ) ← randomPublicMethod(classes)
    <seqs, vals> ← randomSeqsAndVals(nonErrorSeqs,  $T_1$ ... $T_k$ )
    newSeq ← extend(m, seqs, vals)
    // Discard duplicates.
    if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
        continue
    end if
    // Execute new sequence and check contracts.
    < $\underset{o}{\rightarrow}$, violated> ← execute(newSeq, contracts)
    // Classify new sequence and outputs
    if violated = true then
```

```
        errorSeqs ← errorSeqs ∪ {newSeq}
    else
        nonErrorSeqs ← nonErrorSeqs ∪ {newSeq}
        setExtensibleFlags(newSeq, filters, <$\vec{o}$>) // Apply filters.
    end if
end while
return <nonErrorSeqs, errorSeqs>
```

This algorithm builds sequences, starting from an empty set of sequences. As soon as a sequence is constructed, it is executed to make sure that it creates non-redundant and legal objects.

A sequence has an related Boolean vector: every value *s.i* has a Boolean flag *s.i.extensible* that points out whether the given value may be used as an input to a new method call.

Sequence creation first picks a method *m(* T_1 ... T_k *)* at random among the public methods of classes. Then it tries to apply the extension operator to *m*. At every step, it adds a value to v*als*, and potentially also a sequence to *seqs*. The sequence *newSeq* is the result of applying the extension operator to *m*, *seqs*, and *vals*. If *newSeq* is equal to a sequence in *nonErrorSeqs* or *errorSeqs*, the algorithm will try again to create a new sequence.

If the algorithm has created a new sequence. The helper function *execute(newSeq, contracts)* runs each method call in the sequence and checks the contracts after each call.

The output of *execute* is the pair <$\vec{o}$, *violated*> consisting of the runtime values created during the execution of the sequence, and a Boolean flag *violated*. The flag is set to true if at least one contract was violated during execution. A sequence that leads to contract violation is added to the set *errorSeqs*. In other words, if the sequence leads to no contract violation, it would be added to *nonErrorSeqs*.

4. Usage

There are 7 arguments required sequentially in the experiment.

<***timeout***>: time in seconds for testing

<***seed***>: seed for Python Random.random object used for random number generation in the code, if it is stochastic

<***depth***>: maximum length of a test generated by the algorithm

<***width***>: maximum memory/BFS queue/other parameter that is basically a search width

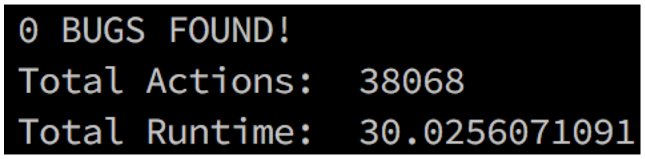
<***faults***>: either 0 or 1 depending on whether the tester should check for faults in the SUTs; if true, save a test case for each discovered failure in the current directory

<***coverage***>: either 0 or 1 depending on whether a final coverage report should be produced, using TSTL's internalReport() function.

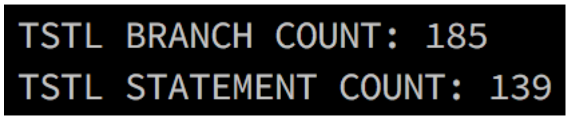
<***running***>: either 0 or 1 depending on whether running info on branch coverage should be produced

5. Experiment Result

Basically, I adopt the same command "python tester.py 30 1 100 0 1 1 1" to test my code. In tester1, my code is crushed, there is no result to compare. In tester2, the result is as follows.

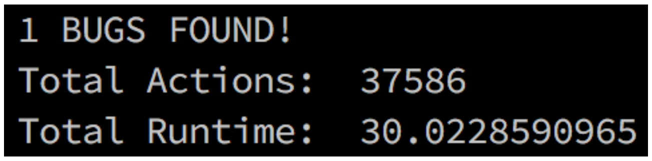


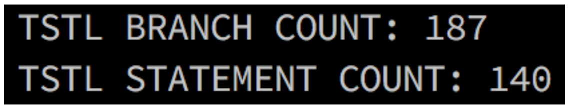
```
0 BUGS FOUND!
Total Actions:  38068
Total Runtime:  30.0256071091
```



```
TSTL BRANCH COUNT: 185
TSTL STATEMENT COUNT: 139
```

In finaltester, the result is as follows.


```
1 BUGS FOUND!
Total Actions:  37586
Total Runtime:  30.0228590965
```


```
TSTL BRANCH COUNT: 187
TSTL STATEMENT COUNT: 140
```

In finaltester, I just modified a small portion of the code that makes the code work more efficiently.

6. Conclusion

This is my first time working on TSTL APIs. Even though it is designed for the particular programming language – python, it is still a powerful and useful tool for software testing purpose. It shows every detail in software testing by using my designed algorithm. It helps me to improve my algorithm and gives me feedback immediately. If TSTL APIs can be adopted by more programming languages, it will dominate software testing.

7. References

[1]. Groce, Alex, el al. "TSTL: a language and tool for testing." Proceedings of 2015 International Symposium on Software Testing and Analysis. ACM, 2015.

[2]. Richard Hamlet. Random Testing. Encyclopedia of Software Engineering, 1994

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