CS 569 COMPETITIVE MILESTONE 2

Xuhao Zhou

May 18, 2016

1. Introduction

In automated test generation, a test harness defines the set of valid tests for a system, and usually also defines a set of correctness properties. Automated test generation and model checking are major difficulties in writing test harnesses. Groce et al. [1] presents TSTL, the Template Scripting Testing Language, a domain-specific language (DSL) for writing test harnesses. TSTL compiles harness definitions into a graph-based interface for testing, making generic test generation and manipulation tools that apply to any SUT possible. Automated generation of tests relies on the construction of test harnesses. A test harness defines the set of valid tests (and, usually, a set of correctness properties for those tests) for the Software Under Test (SUT). A TSTL harness defines a template for action definition, and the compiler instantiates the template exhaustively. TSTL enables a declarative style of test harness development, focuses on defining the actions in valid tests, produces a SUT interface and makes it possible for users to easily apply different test generation algorithms to the same system without much effort. There are two concepts we need to understand: "First, at any state of the system, the only actions that are enabled are those that do not use any non-initialized pool values. Second, a value that has been initialized cannot be initialized until after at least one action that uses it has been executed. [1]" According to [2], "in some circumstances, random testing methods are more practical than any alternative, because information is lacking to make reasonable systematic test-point choices." Therefore, this project implements a tester to generate improved random test cases using the TSTL test harness.

2. Algorithm

In order to improve the performance of the regular random testing method, the tester adapts the idea from the class and [3] to implement an efficient test case generation algorithm. The general idea is to expand the coverage of the test cases and avoids random testing's pitfalls. Our algorithm is based on the GenerateSequences algorithm and the algorithm from newCover.py. Below is the GenerateSequences algorithm from [3].

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

```
errorSeqs ← {} // Their execution violates a contract.
nonErrorSeqs ← {} // Their execution violates no contract.
while timeLimit not reached do
    // Create new sequence.
    m(T1 . . . Tk) ← randomPublicMethod(classes)
    ⟨seqs, vals⟩ ← randomSeqsAndVals(nonErrorSeqs, T1 . . . Tk )
    newSeq ← extend(m, seqs, vals)
    // Discard duplicates.
    if newSeq ∈ nonErrorSeqs ∪ errorSeqs then
        continue
    end if
    // Execute new sequence and check contracts.
    ⟨o, violated⟩ ← execute(newSeq , contracts)
    // Classify new sequence and outputs.
    if violated = true then
        errorSeqs ← errorSeqs ∪ {newSeq }
    else
        nonErrorSeqs ← nonErrorSeqs ∪ {newSeq }
        setExtensibleFlags (newSeq , filters, o) // Apply filters.
    end if
end while
return ⟨nonErrorSeqs, errorSeqs⟩
```

3. About Code

There are some arguments can be read from the command line.

<timeout>: time in seconds can be used for testing.

<seed>: seed for Python Random.random object used for random number generation in the code.

<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 tester should check for faults in the SUT; if true, the tester should save a test case for each discovered failure (terminating the test that generated it), in the current directory, as failure1.test failure2.test, etc.

<coverage>: either 0 or 1 depending on whether a final coverage report should be produced, using TSTL's internalReport() function. The tester is allowed to assume that coverage is being collected to guide tests.

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

Reference

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

[2] Hamlet, Dick. "When only random testing will do." Proceedings of the 1st international workshop on Random testing. ACM, 2006.

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