CS569: Static Analysis and Model Checking for Dynamic Analysis
Part 4: Final

Hafed Alghamdi
Email: Alghamha@oregonstate.edu

Introduction

Generating test cases for a Software Under Test (SUT) is not an easy task as there are many approaches and techniques that can be adopted depending on the SUT and the main objective behind the testing. Moreover, it is more challenging to generalize the test generator on deferent SUTs. The most easiest and effective approach is Random Testing because of its simplicity, fast execution of actions and great capabilities in finding faults. Therefore, in this project there are two algorithms have been implemented. The first algorithm's called "PROP" that combines both sequential and random tester techniques based on randomly selected probability to replay back good test cases those are saved whenever new branches get discovered. The aim of this algorithm is searching for faults as quickly as possible by taking advantage of simple random generator speed. The second algorithm's called "Grouping" that uses the same idea of selecting random probabilities the saved good test cases to be replayed back but instead of using the sequential algorithm we make the algorithm focus on randomly selected group of actions for some time. These algorithms can be used through an easy interface that will be described in "How to Use the Test Generator Algorithm" section.

Random/Sequential Algorithm Based on Replaying Saved Good Test Cases (PROP)

PROP algorithm combines both sequential and random tester techniques to search for faults as quickly as possible. Since some actions in the SUTs could trigger a fault by just executing them with any random values, then it is a good idea to execute all actions sequentially at least once before applying any other techniques. Thus, sequential search has been adapted in this algorithm as a first step to execute TSTL generated SUTs only once. The second step is generating random test cases based on Random probability selection that satisfies 0.5 probability condition. It is a normal implementation of any random tester with the addition of saving good test cases in memory whenever new branches got discovered to be replayed back based on random selection of a probability that is less than 0.5. Testing this on the modified avl SUT file reveals that triggering the combination lock problems is promising. *Figure 1* includes some parts of the code shortened to explain PROP algorithm:

```
startTime = time.time()
# Combined Sequntial and Random tester algorithm
for act in sut.enabled():
        seq = sut.safely(act)

sut.restart()
rgen = random.Random(seeds)
action = None
goodTests = []
# RandomTester based on randomly selcted propability
while (time.time() - startTime <= timeout):
        # Based on propability replayback saved good tests those are saved when new branches
        # got discovered. It is good for finding combanition luck faults
        if (len(goodTests) > 0) and (rgen.random() < Prop):
                sut.backtrack(rgen.choice(goodTests)[1])
                if (time.time() - startTime >= timeout):
                        break
        else:
                sut.restart()
        # Based on the depth randomly execute an action
        for s in xrange(0,depth):
                action = sut.randomEnabled(rgen)
                r = sut.safely(action)
                # When getting new branches, save the test case into goodTest list to be re-executed based on random
propability
                if ((length != 0) and ((len(sut.newBranches()) > 0) or (len(sut.newStatements()) > 0))):
                        goodTests.append((sut.currBranches(), sut.state()))
                        goodTests = sorted(goodTests, reverse=True)[:length]
                # Cleanup goodTest list based on the length of the goodTests
                if (length != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= length):
                        RandomMemebersSelection = random.sample(goodTests,int(float((len(goodTests))*.20)))
                        for x in RandomMemebersSelection:
                                goodTests.remove(x)
```

Figure 1

Random Actions Grouping Algorithm Based on Replaying Saved Good Test Cases

This algorithm is similar to the previous one. However, instead of using sequential algorithm, we are grouping actions randomly and executing them based on 50% of the group length. Whenever, there is some tests cases saved, it will be replayed back based on satisfying random probability. This technique is also promising for the combination lock in AVL and other examples. *Figure 2* includes some parts of the code shortened to explain PROP algorithm:

```
sut.restart()
rgen = random.Random(seeds)
action = None
TimeElapsed = time.time()
# RandomTester based on randomly selcted probability
while (time.time() - startTime <= timeout):
            # Based on propability replayback saved good tests those are saved when
            # new branches got discovered. It is good for finding combination luck faults
            if (len(goodTests) > 0) and (rgen.random() < Prop):
                        sut.backtrack(rgen.choice(goodTests)[1])
                        # Based on the depth execute an action randomly building on the previously executed savedTest
                        for s in xrange(0,depth):
                                    action = sut.randomEnabled(rgen)
                                    r = sut.safely(action)
                                    # When getting new branches, save the test case into goodTest
                                    # list to be re-executed based on random probability
                                    if ((length != 0) and ((len(sut.newBranches()) > 0) or (len(sut.newStatements()) > 0))):
                                                goodTests.append((sut.currBranches(), sut.state()))
                                                goodTests = sorted(goodTests, reverse=True)[:length]
                                    # Cleanup goodTest list based on the length of the goodTests
                                    if (length != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= length):
                                                RandomMemebersSelection = random.sample(goodTests,int(float((len(goodTests))*.20)))
                                                for x in RandomMemebersSelection:
                                                            goodTests.remove(x)
            # This part will randomly selects a group of actions and continue testing them based on a given probability
            else:
                        sut.restart()
                        # Select a group of action
                        groupActions = random.sample(sut.enabled(),int(len(sut.enabled())* Prop))
                        # Calculate the length of groupActions and take some percentage to be the depth
                        for s in xrange(0,int(len(groupActions) * Prop)):
                                    if (time.time() - startTime >= timeout):
                                                break
                                    # Use the actions them one by one
                                    action = groupActions[s]
                                    # Execute The actions one by one
                                    r = sut.safely(action)
                                    # When getting new branches, save the test case into
                                    # goodTest list to be re-executed based on random probability
                                    if ((length != 0) and ((len(sut.newBranches()) > 0) or (len(sut.newStatements()) > 0))):
                                                goodTests.append((sut.currBranches(), sut.state()))
                                                goodTests = sorted(goodTests, reverse=True)[:length]
                                    # Cleanup goodTest list based on the length of the goodTests
                                    if (length != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= length):
                                                RandomMemebersSelection = random.sample(goodTests,int(float((len(goodTests))*.20)))
                                                for x in RandomMemebersSelection:
                                                            goodTests.remove(x)
```

Figure 2

Algorithms Evaluations

Test generator algorithms evaluation is not an easy task as their inputs might differ from each other and some of them are good at things while the other are good at something else. However, because of the time frame in this section we are just trying to evaluate the algorithms by comparing them with existing algorithms such as randomtester.py and SWARM using the default values. The only thing that is going to be changed is time. There are four SUTs we have used to compare these algorithms to get a flavor of how well they are comparing to very well-known algorithms.

1. tictactoe.tstl :
   In this part we enabled the property check option on all the algorithms

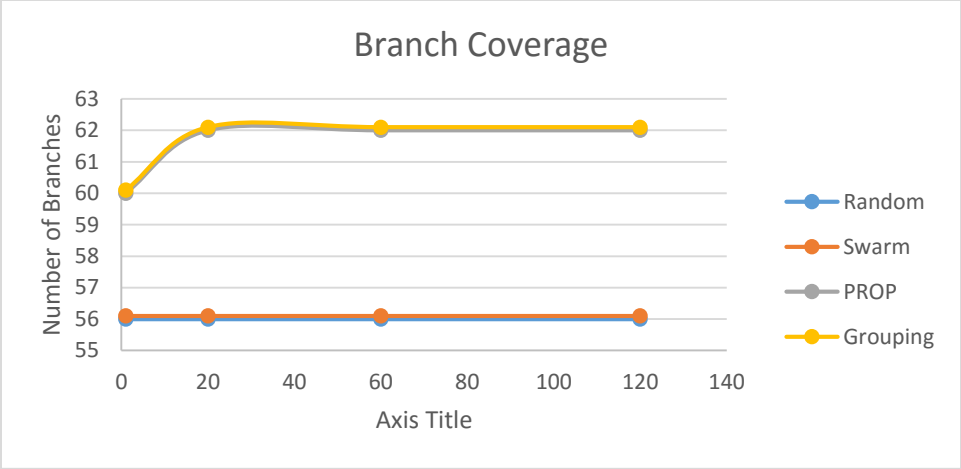


Figure 3

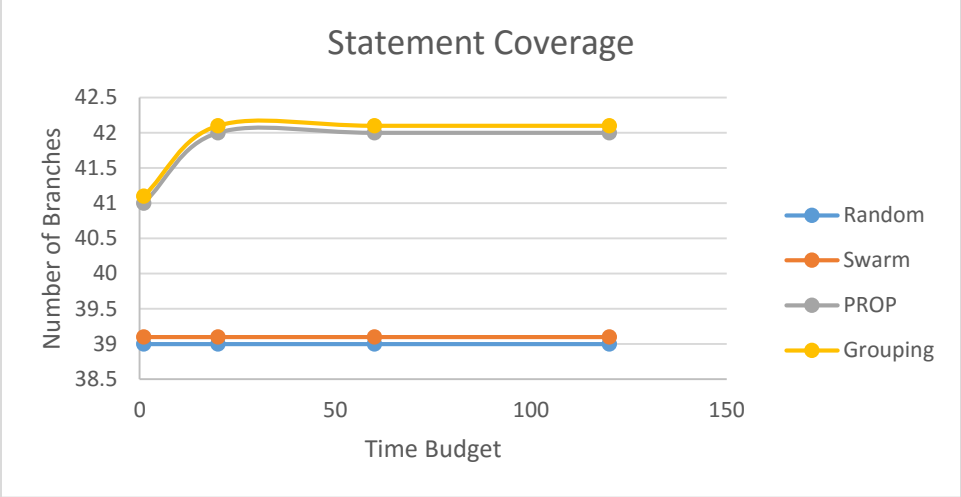


Figure 4

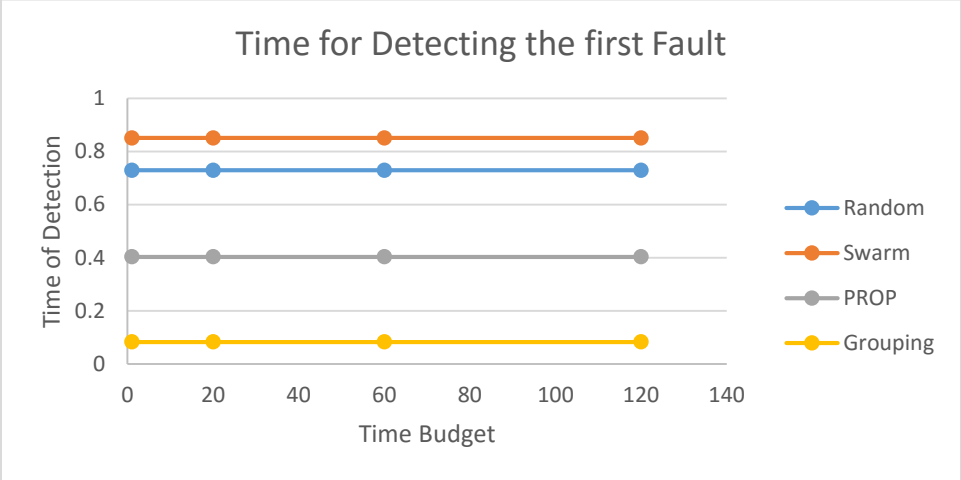


Figure 5

2. stack.tstl:

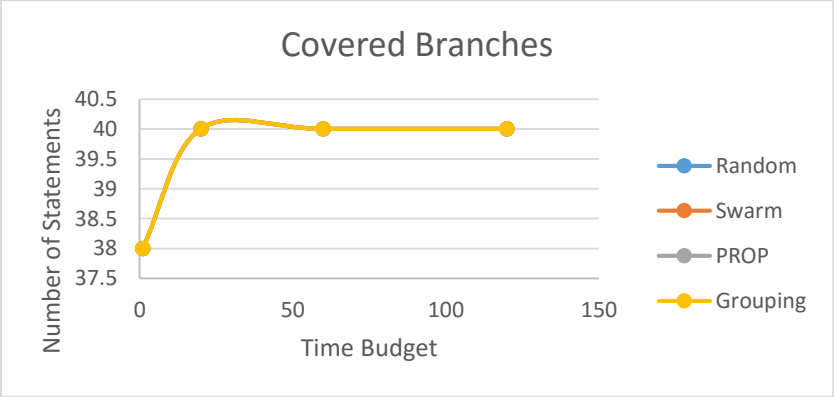


Figure 6

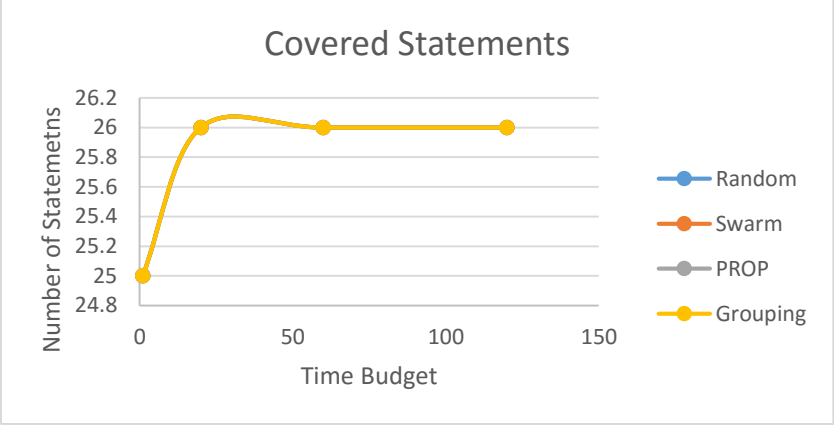


Figure 7

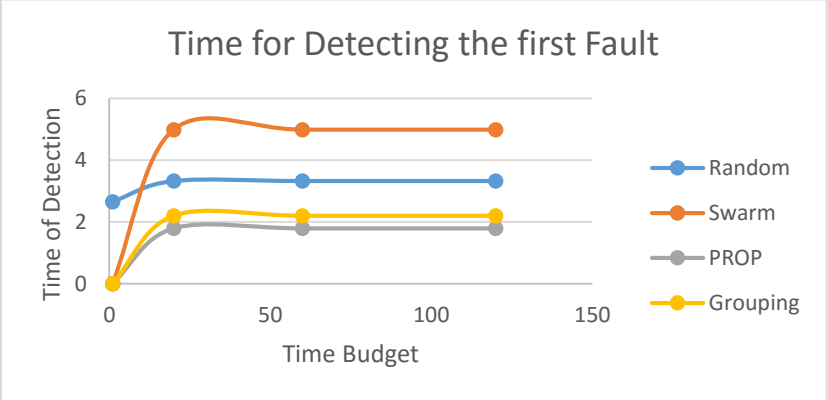


Figure 8

3. avl.tstl :

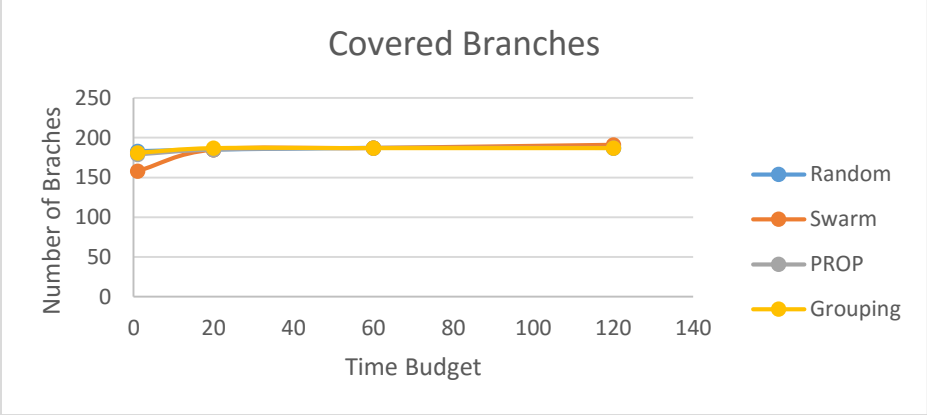


Figure 9

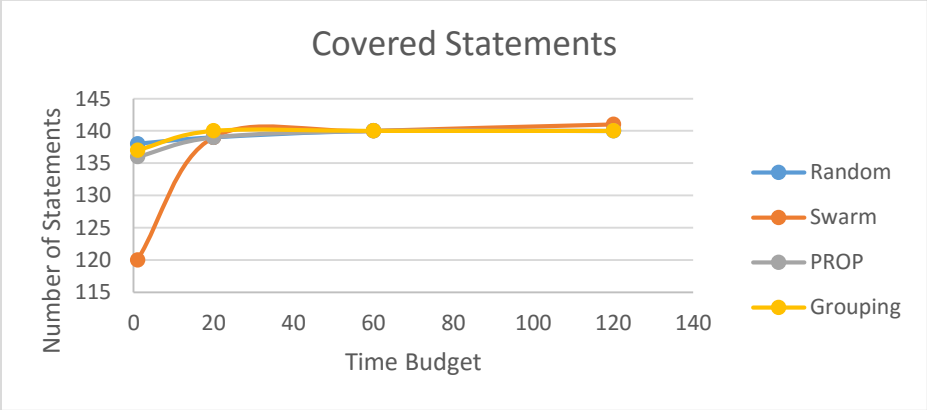


Figure 10

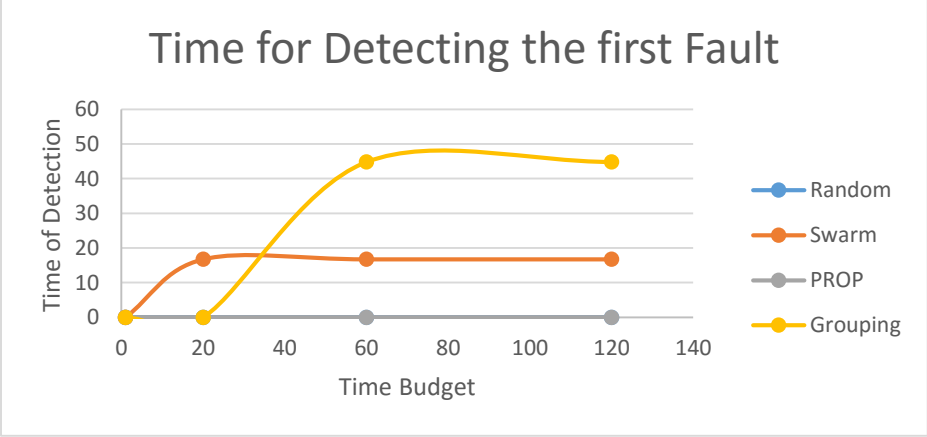
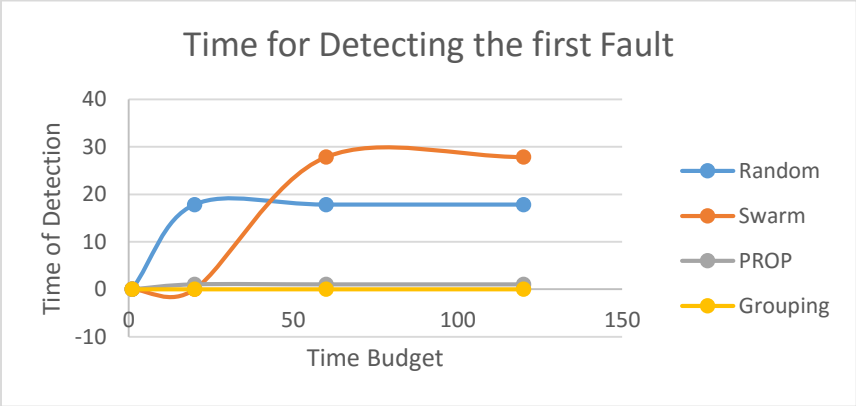


Figure 11

4. numpy.tstl



References:

[1] **"Adaptive Random Testing",**
http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf

[2] **"Lightweight Automated Testing with Adaptation-Based Programming",**
http://www.cs.cmu.edu/~agroce/issre12.pdf

[3] **"Adaptive Random Testing: the ART of Test Case Diversity",**
https://pdfs.semanticscholar.org/11c1/87d3cd8394f4d13523b97d0a40cbdced1691.pdf

[4] **"New Strategies for Automated Random Testing",**
http://etheses.whiterose.ac.uk/7981/1/ociamthesismain.pdf

[5] **"TSTL: The Template Scripting Testing Language",**
https://github.com/agroce/cs569sp16/blob/master/tstl.pdf

[6] **"TSTL: The Template Scripting Testing Language",**
https://github.com/agroce/tstl

1-