

# CS569: Static Analysis and Model Checking for Dynamic Analysis

## Part 2: COMPETITIVE MILESTONE 1

Hafed Alghamdi

Email: [Alghamha@oregonstate.edu](mailto: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 adapted depending on the SUT. Moreover, it is more challenging to generalize the test generator on deferent SUTs. The most easiest and effective approach is Random Testing as it is easy to implement and it provide excellent results most of the time. The initial plan for this project was implementing an adaptive random testing technique. However, it turns out that many of software Testers (students) are going to implement similar approaches. Therefore, the plan was changed to implement a Genetic Algorithm instead of the Adaptive Random Testing algorithm. Genetic algorithms attracted my attention as it applies a simple idea inspired by natural evolution that can be applied using the following general steps:

- 1- Initializing population
- 2- Evaluate population
- 3- While ( condition is not satisfied)
  - a. Selection(population)
  - b. Crossover(population)
  - c. Mutate(population)
  - d. Evalute(population)

The idea is to apply a very simple algorithm that takes only one action randomly at a time like in Hill Climbing algorithm approach. A simple naïve algorithm was tested and revealed many problems including the local maxima problem which requires restarting the entire test and spending time in the wrong place. Therefore, it seemed to be a waste of time as discussed with Professor Alex, since these algorithms are case specific and it becomes more challenging to generalize over multiple SUTs. Although, the idea seemed to be relatively easy, it turns out that it requires a lot of time to adapt in TSTL. Therefore, I decided to stop working on it and implementing Random Tester with a little modification as it will help in the overall competitions and its implementation is going to be relatively easy. There is another idea that will be explored as discussed with professor Alex based on grouping actions depending on a given timeout. This interesting idea will be implemented separately as an extra effort to the class. So, in this class, I will implement a modified version of random tester and if the time helps before the end of the term, I will implement a random tester based on actions grouping.

### ***Random/Sequential Algorithm Based on Random Probability and Good Test Cases***

This 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, using this approach on some of the SUTs revealed that a fault can be detected in a fraction of a second because these actions need to be just executed to produce a fault. Therefore, sequential search has been adapted in this algorithm as a first step to execute TSTL generated STUs only once. The second step in this algorithm 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 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 explains the algorithm in details.

```
# Function To Save The Faults
def saveFaults(elapsedFailure, fault, act, bug, testCase, Algorithm):
    FileName = 'failure'+str(bug)+'test'
    file = open(FileName, 'w+')
    print >> file, Algorithm
    print >> file, elapsedFailure, "Time it takes the tester to discover this fault \n"
    print >> file, fault, "\n"
    print >> file, " Test Case \n"
    i = 0
    # Reading the test case as in randomester.py
    for s in testCase:
        steps = "# STEP " + str(i)
        print >> file, sut.prettyName(s[0]).ljust(80-len(steps), ' '), steps
        i += 1
    file.close()
    print fault

# Sequential algorithm that will traverse over all actions and execute them one by one
for act in sut.enabled():
    seq = sut.safely(act)
    if (not seq) and (FaultsEnabled == 1):
        Sequential = "Discovered By Sequential Algorithm"
        elapsedFailure = time.time() - startTime
        bugs += 1
        print "FOUND A FAILURE"
        sut.prettyPrintTest(sut.test())
        test = sut.test()
        Fault = sut.failure()
        saveFaults(elapsedFailure, Fault, act, bugs, test, Sequential)
        sut.restart()
        # Print the new discovered branches
    if (len(sut.newBranches()) > 0) and (RunningEnabled == 1):
        print "ACTION:", act[0]
        elapsed1 = time.time() - startTime
        for b in sut.newBranches():
            print elapsed1, len(sut.allBranches()), "New branch", b

rgen = random.Random(seeds)

# RandomTester based on randomly selected propability
while (time.time() - startTime <= timeout):
    # This will work only Memory input is set. It is good for finding combination luck faults
    if (len(goodTests) > 0) and (rgen.random() < 0.5):
        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):
    if (time.time() - startTime >= timeout):
        break
    action = sut.randomEnabled(rgen)
    r = sut.safely(action)
    # Start saving discovered fault on Disk
    if (not r) and (FaultsEnabled == 1):
        RandomAlgorithm = "Discovered By Random Algorithm"
        elapsedFailure = time.time() - startTime
        bugs += 1
        print "FOUND A FAILURE"
        sut.prettyPrintTest(sut.test())
        test = sut.test()
        Fault = sut.failure()
        # Saving discovered fault on Disk
        saveFaults(elapsedFailure, Fault, action, bugs, test, RandomAlgorithm)
        # Rest the system state
        sut.restart()
    if (time.time() - startTime >= timeout):
        break
    # Print the new discovered branches
    if (len(sut.newBranches()) > 0) and (RunningEnabled == 1):
        print "ACTION:", action[0]
        elapsed1 = time.time() - startTime
        for b in sut.newBranches():
            print elapsed1, len(sut.allBranches()), "New branch", b
    if (time.time() - startTime >= timeout):
        break
    # When getting new branches, save the test case into goodTest list to be executed based on random
    propability
    if ((length != 0) and (len(sut.newBranches()) > 0)):
        goodTests.append((sut.currBranches(), sut.state()))
        goodTests = sorted(goodTests, reverse=True)[:length]
    # Cleanup goodTest list based on the length of the goodTests
    elif (length != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= length):
        RandomMembersSelection = random.sample(goodTests, int(float((len(goodTests))*0.20)))
        for x in RandomMembersSelection:
            goodTests.remove(x)
```

Figure 1

### ***How to Use This Algorithm:***

This algorithm can be used as specified in project part 2 requirements. When searching for combination lock faults, it is recommended to set the MEMORY/WIDTH to some value. If you want to use the algorithm for any other faults, you can use the default values as commented in the code. Here are some examples:

- 1- Combination Lock Faults like in avl SUT:  
`python tester1.py 60 0 100 100 1 1 1`
- 2- For any other test cases, the following can be used as an example:  
`python tester1.py 60 0 100 0 1 1 1 # To use the normal Random Tester`  
**OR**  
`python tester1.py 60 10 100 0 1 1 1 # To change the number of seed`

You can use as *Figure 2* as a reference to what input you must use.

```
# Terminate the program with time
# You can use 60 as a default Value
timeout = int(sys.argv[1])

# Determines the random seed for testing. This should be assigned 0 when using the
MEMORY/WIDTH
# You can use 12 as a default Value
seeds = int(sys.argv[2])

# TEST LENGTH or Depth
# You can use 100 as a default Value
depth = int(sys.argv[3])

# MEMORY or Width, the number of "good" tests to store
# You can use a 100 as a default Value when testing combination lock faults
length = int(sys.argv[4])

# Enable/Disable Faults
# You can use 1 as a default Value
FaultsEnabled = int(sys.argv[5])

# Enable/Disable Coverage
# You can use 1 as a default Value
CoverageEnabled = int(sys.argv[6])

# Enable/Disable Running
# You can use 1 as a default Value
RunningEnabled = int(sys.argv[7])
```

*Figure 2*