CS569: Static Analysis and Model Checking for Dynamic Analysis
Part 2: COMPETITIVE MILESTONE 1

Hafed Alghamdi
Email: Alghamha@oregonstate.edu

Introduction:

Generating test cases for a software under test is not an easy task as there are many approaches and techniques that can be adapted. The most easiest and effective approach is Random Testing as it is easy to implement and it provide excellent results. The initial plan was implementing an adaptive random testing technique. However, it turns out that many of my colleagues were going to implement similar approaches. Therefore, the plan has changed to implement genetic algorithm instead. Genetic algorithms attracted my attention as it applies a simple idea inspired by natural evolution which 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 approach that it takes only one action randomly at a time like in Hill Climbing algorithm approach. This approach may have many problems including local maxima problem which requires restarting the test, but it seems to be interesting to keep working on this idea separately as an extra effort to the class. Although, the idea seemed to be easy, it turns out that it requires a lot of time to adapt in TSTL. Therefore, I will continue working on it to learn more about genetic algorithms and implementing Random Tester with a little modification as it will help in the competitions and it is easy to implement. 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 the simple genetic algorithm approach.

Random Sequential based on random probability Algorithm

This simple naïve algorithm combine both sequential and random tester techniques to search for faults as quickly as possible. Since some actions in the Software Under Test (SUT) 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 raise an error. Therefore, sequential search has been adapted to this TSTL Test Generator. This is the first step. The second step in this algorithm is Random Testing based on selecting Random probability less than 50%. It is a normal implementation of a random tester but with saving a the good test cases in a list to be replayed back based on the random selection of a probability less than 0.5. Testing this on the modified avl SUT file reveals that triggering the combination lock problems is sometimes faster than randomTester.py. *Figure 1*: Shows the algorithm in details

```
# gloable variables initilization
sut = sut.sut()
sut.silenceCoverage()
bugs = 0
goodTests = []
startTime = time.time()

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

# Sequntial 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 (FaultEnabled == 1):
		elapsedFailure = time.time() - startTime
		bugs += 1
		print "FOUND A FAILURE"
		sut.prettyPrintTest(sut.test())
		test = sut.test()
		Fault = sut.failure()
		print "REDUCING"
		REDUCING = sut.reduce(sut.test(),sut.fails, True, True)
		sut.prettyPrintTest(REDUCING)
		saveFaults(elapsedFailure, Fault, act, bugs, REDUCING)
		sut.restart()

# RandomTester based on randomly selcted propability
while (time.time() - startTime <= TIMEOUT):
	# This will work only Memory input is set. It is good for finding combanition luck faults
	if (len(goodTests) > 0) and (rgen.random() < 0.5):
		sut.backtrack(rgen.choice(goodTests)[1])
	else:
		sut.restart()
	# Based on the depth randonly execute an action
	for s in xrange(0,DEPTH):
		action = sut.randomEnabled(rgen)
		r = sut.safely(action)
		# Start saving discovered fault on Disk
		if (not r) and (FaultsEnabled == 1):
			elapsedFailure = time.time() - startTime
			bugs += 1
			print "FOUND A FAILURE"
			sut.prettyPrintTest(sut.test())
			test = sut.test()
			Fault = sut.failure()
			# Start Reducing the Test Case
			print "REDUCING"
			REDUCING = sut.reduce(sut.test(),sut.fails, True, True)
			sut.prettyPrintTest(REDUCING)
			#Saving discovered fault on Disk
			saveFaults(elapsedFailure, Fault, act, bugs, REDUCING)
			# Rest the system state
			sut.restart()
		# 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
		# When getting new branches, save the test case into goodTest list to be executed based on random 
propability
		if ((MEMORY != 0) and (len(sut.newBranches()) > 0)):
			goodTests.append((sut.currBranches(), sut.state()))
			goodTests = sorted(goodTests, reverse=True)[:MEMORY]
		# Cleanup goodTest list based on the length of the goodTests
		elif (MEMORY != 0) and (len(sut.newBranches()) == 0) and (len(goodTests) >= MEMORY):
			RandomMemebersSelection = random.sample(goodTests,int(float((len(goodTests))*.20)))
			for x in RandomMemebersSelection:
				goodTests.remove(x)
# Printing Report
print elapsed, "Total Running Time"
print bugs, " Bugs Found"
if CoverageEnabled == 1:
	CoverageFileName = 'coverage.out'
	sut.report(CoverageFileName)
	print "Coverage Report is Saved on Disk"
	print len(sut.allBranches()),"BRANCHES COVERED"
	print len(sut.allStatements()),"STATEMENTS COVERED"
```

Figure 1

Using this Algorithm:

This algorithm can be used as specified in project part 2 requirements. When applying this on to find a combination lock faults, it is recommended to set the MEMORY/WIDTH to some value. If you want to just use the algorithm for any other faults you can just use the default values. 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 number of seed

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

```
# Terminate the program with time
TIMEOUT = int(sys.argv[1])

# Determines the random seed for testing
SEEDS = int(sys.argv[2])

# TEST_LENGTH or Depth
DEPTH = int(sys.argv[3])

# MEMORY or Width, the number of "good" tests to store
MEMORY = int(sys.argv[4])

# Enable/Disable Faults
FaultsEnabled = int(sys.argv[5])

# Enable/Disable Coverage
CoverageEnabled = int(sys.argv[6])

# Enable/Disable Running
RunningEnabled = int(sys.argv[7])
```
Figure 2