CS569: Test generator Final Report

Instructor: Prof. Alex Groce
Oregon State University
Tien-Lung, Chang
Email: changti@onid.oregonstate.edu

Introduction

This is a document for my test generator (you can see the full detail in finaltester.py). At the beginning of the project, I was consider doing the Adaptive Random Test. However, it still have some difficulty to implement it. Thus, I decided to obtain a little concept from the Adaptive Random Test to my new test generator which is a Random test generator. The basic concept of the this test generator is to randomly generate the test and capture the failure test. In this project I will implement a test generator base on the information cover in class. The program mainly do three things. First, generate a random test and let the action be a random action. Second, check whether the action is safety or not. If the action is not safety, we will determine it as a failure case and add it to our fail case list. If it is a safe case we will check whether the statement is valid or not. We will set a time limitation for the test, so that each test will have a limitation of explore how deep the test will go. In this test generation I only perform one phase in it. If I implement two phase to the test generator, it might cause a timeout problem. The output can be category into three parts, the new statement discovered, the failure test case that we capture and the final result.

Implementation

The program has three mainly function in it, collectCoverage(), failure() and newStatement(). The collectCoverage() will collect the coverage when the test is generating. The purpose for this function is to collect all the coverage to guide the test. In the collectCoverage() function, there's a global variable called coverageCount. This variable is use to count the current coverage information for the context. At the end of the test we will print out the total cover mean and cover sum, which is associated with the coverage count. The second function is failure(). This function will capture the bugs and collect the information of that case. The variable bugs is used to count how many bugs we have and we will output the number of bugs count as a total failure case that we capture. After manipulate the failure case, we have to do sut.reduce() and sut.restart() to arrange the process. When we found a failure test case, we will save the test case by using the sut.saveTest() function. The function takes two argument, one is the failure test case, the other one is the smessage of the failure test case. The last function is the newStatement() function. When the action is a safe case, we will check the length of the new statement and see if it is a valid statement. If it is a valid statement, we will add to our list pool. Finally, we will deal with all the coverage that we collect and get the mean coverage number. The mean coverage number is use to count the coverage below mean.

Execute

To execute the program, we should input the 7 variable into the command line. The command will look like this " python2.7 finaltester.py 30 30 100 1 1 1 1 ". The first argument is "Time". This argument will manage the time for test generator in second. We use the function "time.time()" to get the current time and check if it is below our input time. The second argument is "SEED". This argument is a seed for the function random.random() to generate random number. The third argument is "DEPTH". This argument define the maximum depth of how deep the test generator are. The fourth argument is "WIDTH". This argument is similar with the depth argument, but it is use in define the maximum width of how wide the test generator are. The fifth argument is the "FAULT" argument. This argument is a boolean variable, which indicate that the tester would like to check the faults or not by determine 1 or 0. If the user give the argument a integer 1, we will save test case for each discovered failure in the same directory. The sixth argument is "COVERAGE". This argument is also a boolean variable, which indicate that the final coverage report would be generated or not. Although we would collect all the coverage to guide the test, the generation for final report is depend. If the user give a integer 1, we will produce a coverage report by using the function define in TSTL, which is called "sut.internalReport()". The last argument is "RUNNING" argument. This argument is a boolean variable that indicate the running info branch coverage will be produces or not.

Result

The output from the test can be separated into three part. The first part is the new statement discovered. When we found a new statement, we will print out the information of that test and the length of the old test and new test. The second part is when we found a bug or failure test, we will print out the information of that failure test. The last part is the final result part. This output will give a final summary of the over all test. The final result include the mean coverage calculated from the coverage count, a result of how many statement below mean coverage, a total test that we do in this test, a total failure that we capture, a total action that we execute, the exactly run time that we do this time(approximately equal to the time that we gave when we run the program).

Future

In this test generator program, I only perform one phase. However, to capture more failure test, we can improve the program by adding multiple phase into the program. We also can improve the algorithm that we use in the program by doing a more complex operation such as to realize the Adaptive Random Test in the future.

Reference

[1] Alex Groce coverTester.py. Retrived May 5, 2016, from https://github.com/agroce/cs569sp16/blob/master/SUTs/coverTester.py

[2] Alex Groce NewCover.py. Retrived May 5, 2016, from https://github.com/agroce/cs569sp16/blob/master/SUTs/newCover.py

[3] Chen, T., Leung, H., & Mak, I. (n.d.). Adaptive Random Testing. Retrieved April 19, 2016, from http://www.utdallas.edu/~ewong/SYSM-6310/03-Lecture/02-ART-paper-01.pdf

[4] E Godefroid, P. (n.d.). Compositional Dynamic Test Generation. Compositional Dynamic Test Generation. Retrieved April 18, 2016, from http://dl.acm.org/citation.cfm?id=1190226

[5] Zhang, S., Staff, D., Bu, Y., & Ernst, M. D. (n.d.). Combined Static and Dynamic Automated Test Generation. Combined Static and Dynamic Automated Test Generation. Retrieved April 18, 2016, from http://dl.acm.org/citation.cfm?id=2001463