# CS 569: Static Analysis and Model Checking for Dynamic Analysis

## Introduction:

Random testing has proved to be a simple yet effective approach for software testing. Although there are other algorithms such as Adaptive Random Testing and Mutation algorithms, random testing requires far less fine tuning compared to the other methods. Random testing has served has a basis for many algorithms and variations of random testing have proved to be very efficient. My proposed algorithm for this class also uses random testing as the basis and some additional modifications to make it more efficient for certain cases.
The proposed algorithm is a random tester that uses a coverage based approach. The algorithm borrows heavily from concepts that we have studied in class. In addition to this coverage based tester, I was also working on a genetic algorithm that is similar to the algorithm written in class. A few slight modifications have been made to the algorithm discussed in class. I will mention this algorithm and some of the modifications briefly in this paper.

## Algorithm:

- **finaltester.py**

The proposed method uses a sort of an exploit/explore approach as well. I will discuss how it works very shortly. The algorithm consists of the following stages: a random action is generated based on a random seed. The action is then performed and it is checked whether the action results in a failure or not. If there is a failure the failure is printed and the test associated with the failure is written to a file. As the action is performed, the tester checks whether the actions performed or the test results in new statements that have not been previously covered. If new statements are found, the test associated is saved. This is done so we can backtrack to this test in the future. This concept has been discussed in class and is known as exploring and exploiting. Basically, the tester explores new statements/branches and saves tests that produce new coverage. Based on a probability defined earlier in the program, it then decides whether to continue exploring or backtrack to one of the previously explored tests as to exploit it. This tester basically checks the least covered statements and see if we have ever hit that statement. However, if we have never hit a statement, then that means it is worth exploring even more than the least covered statement. It basically exploits coverage information such as the least covered statement and actions that produce new statements. Thus, this tester is good for combinational locks which have little coverage and this allows it to lock on to them. The algorithm stops when the terminating condition is reached i.e. when the timeout value is reached.

- **mytester.py**

In addition to the above algorithm, another algorithm was also written which was also based heavily on the concepts discussed in class. This was the genetic algorithm with a slight modification on the mutation and crossover methods. As we saw in class, fine tuning the mutation aspect of the algorithm required to explore a lot of factors. However, we wrote a simple mutation function which takes a test and randomly selects up to an action and beyond that action it mutates or changes the actions. Finally, this function returns this test as the mutated test. My modification of this algorithm was to randomly decide to mutate the same algorithm up to two times. In addition to that, it also did crossover between two tests. It would take the first half of one test and the second half of another test and combine to produce a new test. Other than that, this algorithm contained all the previous features of the first algorithm such as timeout condition, saving faults etc.

**How to run:**

The first tester (finaltester.py) requires a few different parameters as inputs from command line. These parameters include timeout, seed, depth, width, faults, coverage and running. Timeout is the time for how long the test needs to be run. Seed is used to determine the random generator seed. If the fault option is enabled, the failed test is saved to a file. Coverage enables the final results to be displayed such as branch count and statement count. Running info allows branch information to be displayed as the tests are performed.

## Evaluation:

The evaluation results are based on the tests performed in the competition bakeoff of class and some other experiments that I ran. The goal was to gain similar performance as to the TSTL pure random tester in terms of coverage. However, this was not achieved. The pure random tester in TSTL performed better due to the fact that it is that –a pure random tester. It does not have much overhead or decision making; it just performs a test and moves on to the next. Thus, it is very simple and effective. My algorithm uses an explore/exploit approach and thus requires a little bit of decision making and therefore loses to the pure random tester in TSTL. The proposed tester was used to test SQLPARSER and Z3. The test results were used to compare the proposed tester to a pure random tester and the swarm tester. The results are as following:

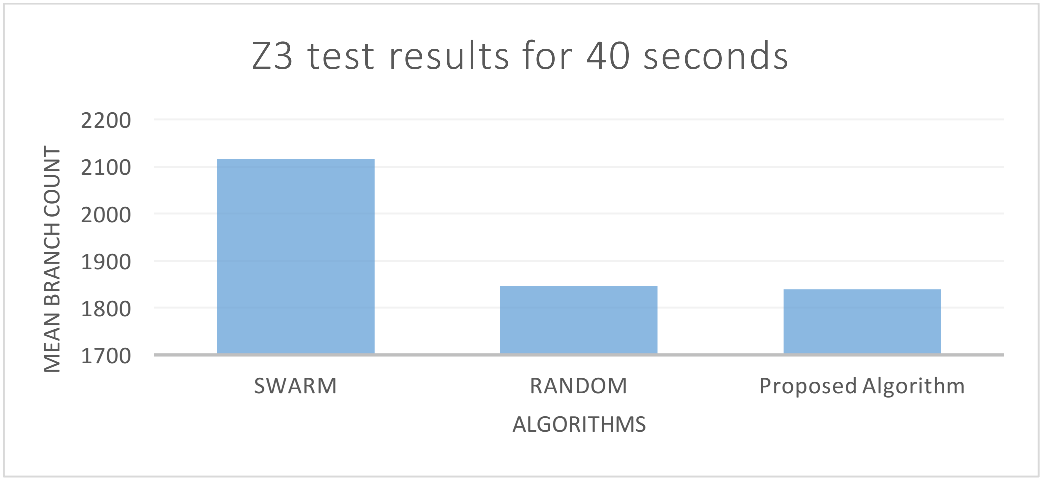

**Fig. 1: Graph of mean branch count for Z3 test results for the three algorithms**

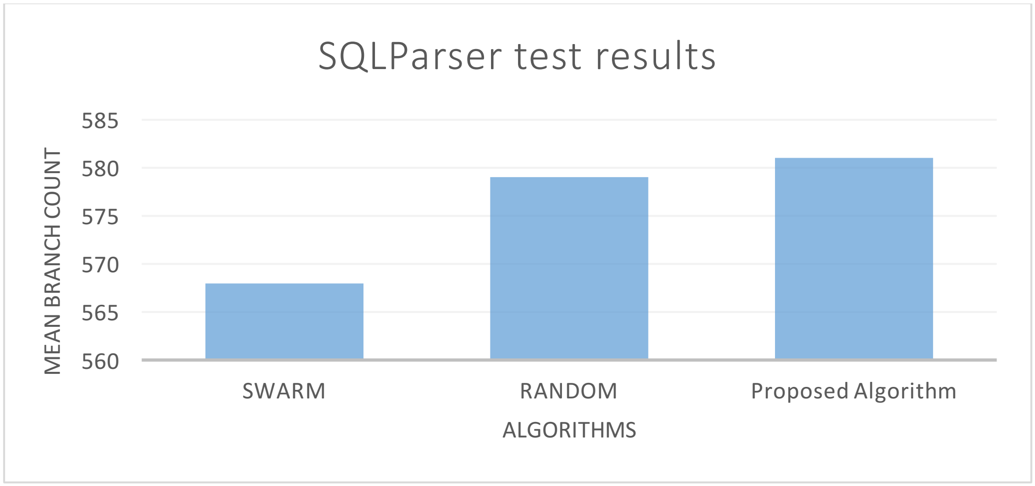

**Fig. 2: Graph of mean branch count for SQLParser test results for the three algorithms**

As you can see from the graphs above, one type of tester may be suited well to a certain type of sut. From figure 1, it can be seen that swarm tester provided the highest number of branch count. My tester was based on the random tester and produced a similar result for Z3. However, the random tester performed slightly better than mine. This is because the random tester is faster and does not involve any complex decision making whereas my tester explores test cases and then exploits them based on a random probability. For the SQLParser, my tester performed better than the random tester and the swarm tester. This is because it collects information regarding tests which produce new statements and it can be beneficial to explore least covered statements and tests that produce new statements. In addition to coverage, my tester also found bugs in another SUT known as FSM.

## Conclusion:

As we can see, different types of testers perform differently on each SUT. However, some testers such as the genetic algorithm will perform better if they are finely tuned. For example, a more involved mutation function and an addition of a fitness function will definitely prove to yield better results. Overall, this class was very interesting and provided a good solid foundation for testing using TSTL.

## References:

[1] T. Y. Chen, F. C. Kuo, R. G. Merkel and S. P. Ng, "Mirror adaptive random testing," Quality Software, 2003. Proceedings. Third International Conference on, 2003, pp. 4-11.

[2] F.-C. Kuo ,An Indepth Study of Mirror Adaptive Random Testing,&rdquo, Proc. Ninth Int',l Conf. Quality Software, pp. 51-58, 2009.

[3] T. Y. Chen, "Adaptive Random Testing," 2008 The Eighth International Conference on Quality Software, Oxford, 2008, pp. 443-443.

[4] Z. Q. Zhou, "Using Coverage Information to Guide Test Case Selection in Adaptive Random Testing," Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual, Seoul, 2010, pp. 208-213.