CS569: Static Analysis and Model Checking for Dynamic Analysis
Part 1: Project Proposal

Hafed Alghamdi
Email: Alghamha@oregonstate.edu

Introduction:

In this class, we are trying to implement a Test Generator Algorithm that could possibly search and execute actions as fast as possible for the Software Under Test (SUT) generated by TSTL. The aim of this tester is working on multiple SUTs generated by TSTL. Furthermore, an essential goal in this project is generalizing the tester not only by having it working on different SUTs but also adapting inputs parameters available in SUTs generated by TSTL such as timeout, seeds, depth, width and coverage report generation. As we know, the current Random Tester is extremely fast as it is just randomly picking an action and executing it on the fly. So, finding an algorithm that could possibly capturing faults as early as possible seems to be a good idea. Therefore, we were able to find an algorithm or technique called "Adaptive Random Testing" that could chase faults faster than the regular Random Tester and possibly at the same speed rate. However, we don't guarantee that this algorithm will be faster than the regular Random Tester in searching and executing actions but when it comes to finding faults, it should be faster. [1][2][3]

Project Plan:

The plan for this project is exploring different papers related to Adaptive Random Testing algorithm and adapt these ideas in TSTL. In order to achieve that, the following steps must be followed:

1- Understanding TSTL random test generator options[5] including:

- depth <int>: Determines the length of generated tests.
- timeout <int>: Determines the maximum time spent generating tests, in seconds.
- seed <int>: Determines the random seed for testing.
- maxTests <int>: Determines the maximum number of tests to be generated.
- stutter <float P>: If this option is provided, the random tester will repeat any still-enabled
- actions with probability P.
- running: Produce on-the-y, time-stamped code coverage information, for analyzing performance of testing algorithms.
- replayable: Produce a log of the current test, so even it crashes Python the test can be reproduced, delta-debugged, made stand-alone, or otherwise analyzed.
- total: Produce a total log of all test activity, including across resets, for systems where reset is not complete (so tests across resets can be delta-debugged).
- quickTests: Produce \quick test" _les [29], each containing a minimal test to cover a set of branches of the SUT.
- normalize: Apply additional, custom term-rewriting based simpli_cations of test cases that often further minimize delta-debugged test cases.
- generalize: Apply generalization that elaborates each failing test with annotations showing similar tests that also fail.

2- Understanding some core methods in TSTL for testing an SUT[5], including:

- restart(): resets the system state and aborts the current test.
- test(): returns the current test.
- replay(test): replays a test, and returns a Boolean indicating success or failure of the test.
- enabled(): returns a list of all currently enabled actions.
- randomEnabled(random): given a Python random number generator object, returns a
- random enabled action, e_ciently (avoiding unnecessary guard evaluations).
- safely(action): performs action (usually changing SUT state) and returns a Boolean indi-
- cating whether the action performed raised any uncaught exceptions.
- check(): returns a Boolean indicating whether any properties fail for the current state.
- error(): returns either None (no error for the last action or check), or a Python object repre-
- senting an uncaught exception or failed property's backtrace.
- state(): returns the current SUT state, as a set of values for all pools; for systems where
- state cannot be restored by pool values, or deepcopy does not work, returns the current test.
- backtrack(state): takes a state or test produced by state and restores the system to it.
- allBranches(): returns the set of branches covered during all testing.

- newBranches(): returns the set of branches covered during the last action executed that had not previously been covered.
- currBranches(): returns the set of branches covered during the current test.

3- Understanding the existing randomtester.py and swarm.py available in TSTL[6]
4- Understanding Coverage report generation in TSTL functions
5- Understanding Adaptive Random Testing Techniques
6- Implementing a simple Adaptive Random Testing that accepts arguments from the command line as specified in “Part 2: COMPETITIVE MILESTONE 1”
7- Run the algorithm on multiple SUT’s such as AVLBlocks.py and compare the results with Swarm and Random tester algorithms those available in TSTL
8- Based on these results, we will try to enhance the algorithm by either making it search and execute actions more faster or enhancing its capabilities to find faults faster

What is your idea for test generation?

The idea is adapting “Adaptive Random Testing” algorithm to TSTL where is can find faults much faster than the regular Random Tester.

Why do you expect this method to possibly be effective?

Actually, we don’t know how well this algorithm is going to behave before implementing it. However, since most of the “Adaptive Random Testing” related papers show positive results, we could assume that it should generate good results in TSTL as well.

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