Program Slicing with Test Data Generation Competitive Milestone 1

Punyapich Limsuwan
May 5, 2016

I. Background

Previously, I proposed to implement program slicing with the random test generator. In general, the idea of this algorithm is to reduce search space by constraining the test suite before generating random data. Slicing criteria need to be defined and the program will be executed once to check if variables are affected. Therefore, the variables that are affected will be included in the slice. Then, slice will be used to identify constraints for generating test data. The algorithm for test data generation requires specific goal statement, which will be satisfied by generated random input from the slice's constraints.

II. Algorithm

To implement program slicing with the given randomtester.py and sut.py, I have to think it differently because there is neither variable nor statement that I could focus on. The random tester runs several actions and check if there are existing failures, also performs reduction, normalization and generalization if needed. Thus, the slicing may not be applied easily since we didn't directly generate test cases over avl.py. I have come up with the idea to be using the general concept of program slicing with a little bit of adjustment. Thus, I need to reduce the search space by using a constraint while randomly generating test cases. In our case, I will focus on generating actions in order to find the suspect actions that may cause the failure. Instead of enabling all actions, limiting to the small set of actions at first allows us to verify that given test cases are either bug-free or buggy. Furthermore, the set of actions will be expandable, if the failures are not found in the limited amount of time, the size of set will grow larger, until it reaches the size that contains all actions. However, if the failure is found within the time, the size of set will stay the same for another amount of time to see whether it will produce new failure or the same failure, if it produces the new failures the time will be expanded until timeout or find the same failure. From my aspect, this algorithm may take longer than the generic random testing to have code coverage, but I hope that it is more efficient to locate failures, and also easier to see the test cases that cause the failure came from which actions. The function in sut.py that I use for this algorithm is randomEnableds, which allows me to specify number of actions in the list. Thus, when it constrains is satisfied by time, the input number for randomEnableds will be increased, until it reaches the maximum number defined.

III. FUTURE WORK

Up to this point, I haven't proved the effectiveness of algorithm so far, the next part I need to run the test generator with several setting of arguments and see how they work. I still have an idea to improve my algorithm, which is about to decrease the randomness of the test suite in order to make it more likely be unit test. However, I will keep that in mind and try to improve my algorithm to be solid, first.