Program Slicing with Test Data Generation

Punyapich Limsuwan
June 6, 2016

I. Background & Motivation

Program Slicing is one of the static analysis techniques proposed by Weiser [1] in 1984. The main concept of program slicing is to concern only statements of the program that are involved with variables of interest. The main purpose of slicing is to reduce the complexity of the codes and constrain amount of variables, which allows us to find faults on suspicious variables and statements with less effort than analyzing the entire program. Program slicing is well known as generic method for software debugging, and it can be represented by Control Flow Graph (CFG) and Program Dependence Graph (PDG), which enable us to analyze data-flow and dependencies.
However, this static analysis method could help us to analyze data to be generated for testing. As we know that using only random test generator is somehow difficult to find faults in the program, since the data is arbitrary generated, we may need large number of data generations to satisfy the goal, which is time and resource consuming. From this aspect, test data generation could be more systematic if we define a scope and constraint to the test suite by implementing program slicing before generating data. This idea has been proposed by Samuel and Surendran in the paper called "Forward Slicing Algorithm based Test Data Generation [2]." Their approach utilizes dynamic slicing in order to narrow search space and decrease test effort. In dynamic slicing, they concern three parameters, which are variable, point of interest and sequence of input values. This makes the slice smaller than the static one. The direction of slicing implemented in this approach is Forward Slicing. Firstly, 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. In my opinion, this method would be effective because faults could be exposed easier if we have obvious goal and smaller search space rather than digging entire codes aimlessly. Program slicing also helps us track an error on set of variables rather than every variable, which could be difficult to check the statement that cause an error. Meanwhile, similar approach proposed by Zhang et. al further improve their approach by applying genetic algorithm to generate testing data along with static slicing to reduce search space [3]. The results from their approach indicate that slicing could reduce the iterative times to generate test.

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 defined set size or the maximum set 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. 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. Results

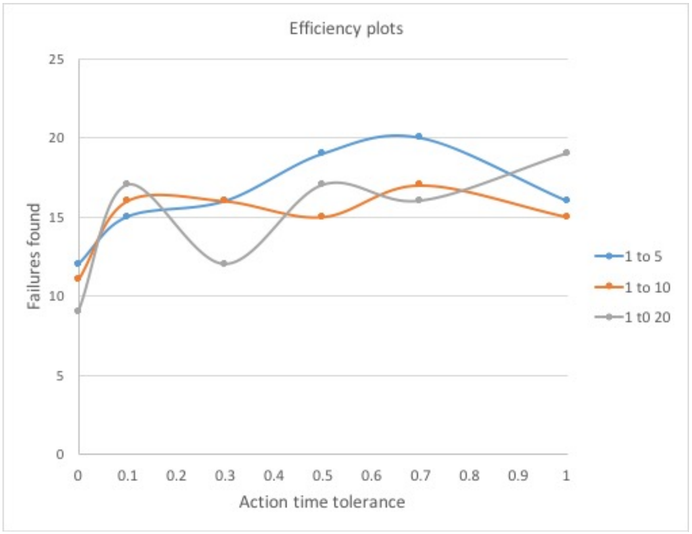

Figure 2 Efficiency plots

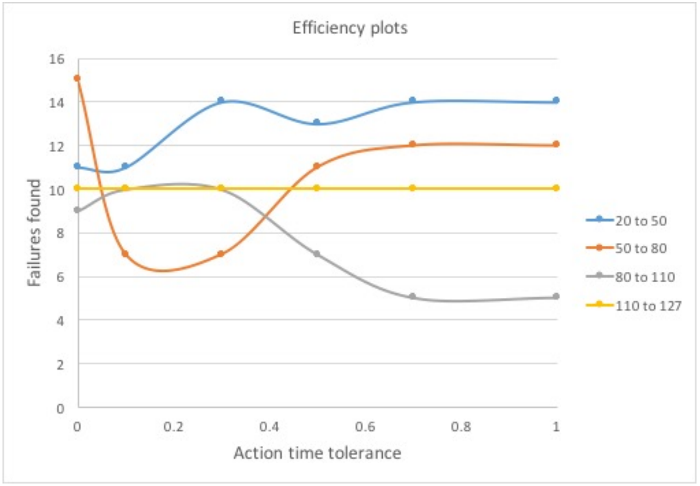

Figure 1 Efficiency plots

To test the efficiency of the algorithm. I conducted the test cases with several configurations. I wanted to measure how action set size affects the coverage. Therefore, I had to fix and changed the major constraints, which were the action time tolerance, number of initial set size, and number of maximum set size. The tstl file to be used in this test is avlefficient.tstl, which tested the avlbug1.py. I set the timeout to 30 seconds, the depth of 100, and the width of 1. For each pair of initial set size and maximum set size, I set the several action time tolerances to increase set size if all actions in set are run over the time tolerance. The figure 1 indicates the initial set size of 1 and the maximum set size of 5,10 and 20 respectively. The action time tolerances are set to 1, 0.7, 0.5, 0.3 and 0, respectively. The higher values mean that the test cases with same action set size would stay longer, and the lower values mean that the action set size would be increased faster until reaching the limit, as defined. The zero tolerance means the increment always happens after the loop ends. The result in figure 1 shows that zero tolerance of every case gives the lowest numbers of failure found. While increasing the tolerance could result in higher numbers in failure found, but the it fluctuates sometimes. In another test suite, I set the the ranges of set size as initial to maximum from, 20 to 50, 50 to 80, 80 to 110 and 110 to 127, respectively. The tolerance times also need to be adapted to 10 times lower than pervious test. I need to change this, because the number will grow too slow, and it would not reach the maximum set size. As shown in figure 2, the result in the range 20 to 50 is similar to the previous test by giving lowest number of failure found at zero tolerance, then the number of failure found is increasing as the tolerance increased. The range 50 to 80 gives the suspiciously highest number of failure at the zero tolerance, and then becomes normal pattern as the tolerance increase. However, when the size is large as 80 to 110, it gives contrast result from previous test by getting highest number of failure found at zero tolerance, and dropping down to the lowest number as tolerance increased. The largest action set size from 110 to 127 gives the same number of failure found regardless of timing tolerance.

IV. Conclusion

From the result on previous section, it indicates that using larger set size of actions tends to decrease the efficiency of finding failures. The numbers of failure found drop from 20 to 10 ranging from the smallest set size to the largest set size. The action time tolerance of the small set size should probably be between 0.5 and 0.7 in order to keep the size increasing neither too fast nor too slow. The result also emphasizes that having too large set size of action is also difficult to find more failure, and that proves my proposal algorithm of using small set size of action in order to get more potential to verify the actions. Finally, the submitted tester will be set the initial set size as 1 and the maximum set size of 20, the tolerance will be set as 0.7 seconds. Even though, this set up is not the highest efficiency found in my experiment, but I would like to keep it in the middle due to the different circumstances.

References

M. Weiser, "Program Slicing," in *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-357, July 1984.

P. Samuel and A. Surendran, "Forward slicing algorithm based test data generation," *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, Chengdu, 2010, pp. 270-274.*,*

Y. Zhang, J. Sun and S. Jiang, "An Application of Program Slicing in Evolutionary Testing," *2009 International Conference on Information Engineering and Computer Science*, Wuhan, 2009, pp. 1-4.