Automated Software Test Data Generation Using Improved Search Procedure

Shubham Agarwal and Abhimanyu Bhattar

Abstract—Automated software test data generation is a complex and one of the most challenging task. Generation of test data manually is tedious and error-prone. The base algorithm, which has been improved in this paper involves redundant and un-necessary execution cycles. In this paper, we have proposed an efficient search procedure that reduces the time complexity of test data generation significantly. Our experiments conducted on varying sizes of source code showing ratio of execution time of automated test data generation using Basic Search Procedure and our Improved Search Procedure conclude that, with the increase of source code size, test data generation takes place in lesser time using our improved search procedure.

Index Terms—Automated test data generation, basic search procedure, risk factor, white box testing, time complexity.

I. INTRODUCTION

Software testing is the process of validating and verifying that a program or application meets the requirements defined in development stage of the application. The application must work as expected. Also, it must satisfy the needs of the stakeholders. Software testing is done to check the reliability of the software, to make sure that software does not contain any bug which can become a reason for failure in later stages of software development. Software testing is a vital but a complex process. It has many general problems like stakeholder involvement and commitment issues, test process issues with inadequate test evaluation methods and inadequate test maintenance. It also has testing and test communication issues involving inadequate test documentation, ambiguous, missing, incomplete or incorrect requirements.

Software test data generation is an important part of software testing. It is the process of creating a set of data for testing the adequacy of new or revised software applications. As per B. Korel [1], Test data generation in program testing is the process of identifying a set of test data which satisfies given testing criterion. Most of the existing test data generators [2]-[6] use symbolic evaluation to derive test data.

However, in practical programs this technique frequently requires complex algebraic manipulations, especially in the presence of arrays. In this research paper, we present an improved version of “Automated Software Test Data Generation” [1]. The basic concept is of Basic Search Procedure, which involves execution of each instruction in the source code file taken as an input. This research paper improves the time efficiency of test data generation algorithm in [1] and also incorporates prioritizing factors (risk factors), which helps in categorizing high priority variables to test first.

This paper is organized as follows. In Section II, we introduce some background information about Basic Search Procedure [1].

In Section III, we address the inefficiency of Basic Search Procedure [1] along with some other shortcomings. Section IV describes our methodology. In Section V, we report our experiment and improvement. In Section VI, we discuss our results. In Section VII, conclusions are drawn based upon experiments conducted and results obtained.

II. RESEARCH BACKGROUND

Test data generation has been profoundly researched over the past. According to the already established research, there are typically three types of test data generators: pathwise test data generators, data specification test data generators and random test data generators. All of them fall under the category of white box testing, as the structure of the program under testing is analyzed in them. This paper focuses on the first type, i.e., pathwise test data generation. This type of test data generation requires specific paths to be given as input along with the program to be tested.

In pathwise test data generation, evaluation of a program can be done using symbolic execution or actual execution. In symbolic execution, instead of supplying normal inputs to the program, one supplies symbolic representations to inputs variables. Each variable is symbolically assigned a value and its value is updated with execution of each statement of the program. The challenge is posed when a conditional statement is encountered. The condition is evaluated for both, its truthfulness and falseness on the basis of input variables and condition(s) [7], [8]. Therefore, a single path is bifurcated into two, with each conditional statement. Input variables are updated corresponding to each path as applicable. To pictorially represent the program flow and various possible paths, an execution tree is formed [7], [8]. Each node of the tree represents an executable statement in the program, and each edge represents control flow from one such statement to another. Conditional statements are represented in the tree by marking true (‘T’) or false (‘F’) on each edge emanating from the conditional node.

In actual execution, test data are generated using actual values of input variables. This involves Basic Search procedure and Dynamic Data Flow Based search [1]. In Basic Search Procedure, each instruction of the program under test
is executed. When a conditional statement is encountered, first its validity is checked. If valid, then we move onto the next statement. If not, the arguments are parsed and checked for user input variables since only such variable’s values can be varied for testing. We start searching for a minimum value of the conditional expression by changing first input variable, say \( x_1 \), (using a one dimensional search procedure) while keeping all the other input variables constant until the solution is found (the conditional expression becomes negative) or the positive minimum of the conditional expression is located. In the latter case, the search continues from this minimum with the next input variable \( x_2 \). The search proceeds in this manner until all input variables \( x_1, x_2, \ldots, x_n \), are explored in turn. After completing such a cycle, the procedure continuously cycles around the input variables until the solution is found or no progress (by incrementing/decrementing the conditional expression) can be made for any input variable. In the latter case, the search process fails to find the solution even if the positive minimum of the conditional expression is located. Such paths are called ‘infeasible paths’ [9].

In case of more than one if statements, on each increment or decrement of the variable in an if statement causes the control to move back to the top and check that every statement above that statement holds true for the change that has been made. That is, suppose a change in the value of a variable in the third ‘if’ statement occurs, then every statement above that statement (specially the if’s) must be in the same path. Our IMPROVED SEARCH PROCEDURE eliminates these redundant cycles after each increment/decrement of variable.

For prioritizing variables, a new field is maintained called ‘risk factor’ for each variable. With each appearance of a variable in a conditional statement, its risk factor is incremented by one. Higher the value of risk factor, more is its priority to mutate its value first.

The following examples represent source codes for which test case needs to be generated. The examples contain declaration, assignment, input, and conditional statements.

Example 1:
1. int a;
2. scanf("%d", &a);
3. if(a<-5);

Example 2:
1. int a;
2. int b;
3. a=4;
4. scanf("%d", &b);
5. if(a<b);

Example 3:
1. int a;
2. int b;
3. int c;
4. scanf("%d%d%d", &a, &b, &c);
5. if(a>4);
6. if(b>8);
7. if(b<a);

For example 1, in the ‘if’ statement (line 3), condition is checked (which is false as ‘a’ is initialized to 0). As ‘a’ is a user input, it is incremented by 1 in order to satisfy the condition statement. After incrementing the variable by 1 the new value of the conditional expression is evaluated and is compared with the previous one. If the newly computed value turns out to be better than the previous (i.e. the condition nears to be true), then we keep incrementing the variable by 1 until either condition is satisfied or in the case when condition moves away from being true. In the latter case, variable is decremented by 1 and the procedure is repeated. If still no progress is found, we move onto the next user input. But if no other user input exists in ‘if’ statement, and if there is no progress observed with the current variable, then search fails to generate any valid test case.

In our case we get the final value of ‘a’=-6.

In example 2, the ‘if’ condition (line 5) is checked which is not true initially as ‘a’ is 4 and b is initialized to 0. The code will check for first variable that is user defined (which is b) so b will be incremented to 5 which is the final result.

In example 3, the first if condition (line 5) is checked which is not true initially as ‘a’ is zero so the condition ‘a>4’ does not hold true. The code will check for the first variable which is user defined which is ‘a’ so ‘a’ will be incremented to 5. Now, the file pointer will move to the next instruction that is if (b>8) (line 6) so in order to make this instruction true, ‘b’ will be increased by 1 and the file pointer will move to the first line and check that every instruction above it holds true if b=1 (in this case). Until b>8 holds true, b will be increased by 1 and the same procedure will follow, so the file pointer will move to the next instruction when b=9.

The next statement is if (b<=a) (line 7). In this case, b will be decreased by 1 and file pointer will move to the first instruction to check that every instruction above the current instruction holds true, which is not the case in this example because for the above instruction to hold true b should be > 8 which has been void in this case, so the program will track back and check for other user input for last if statement, which is ‘a’ in this case, so will be increased and all the above statements will be checked as in the above cases to finally get the output, \( a=10 \).
Final output: \( a=10, b=9 \).

III. RESEARCH GAPS

The Basic Search Procedure [1] involves a lot of execution overhead when programs are huge, which is usually the case at enterprise level. Every time the value of a variable is changed in a conditional statement, the complete code above it is run in order to check if the required path above it is not violated. This leads to huge amount of execution overhead after any change made in any variable. The effect is alarming in the case when say, the code is of hundreds of lines and there are two or more if conditions placed close to each other. In this case, these hundreds of lines are executed for each increment/decrement in each of these if statements. The algorithm also has a setback in the case when the difference between the current user input variable and the value it needs to satisfy the condition is way too much.

Example:
In this example, suppose until line 998, value of ‘a’ is set to 0. When we reach line 999 in order to satisfy conditional statement, ‘a’ should be > 1000, for which ‘a’ will be incremented by 1 and the 998 lines above it will be executed to check whether all the above conditions satisfy. This will be done for 1000 times in line 999 and another thousand in line 1000 which will explode the execution time.

The algorithm suggests to increment/decrement value of the user input variable by 1 every time, in order to satisfy the given conditional statement. This is fine when the difference between the current values of the user input variable and the value it needs to satisfy the condition is small. But when the difference between them is big then huge amount of computations are needed to be done. This can be improved by directly computing the values of the user input variable required to satisfy the condition if the degree of the variable whose value is being changed is less. For example, for the conditions like, if (a>1000), if (a^2+2a+1>20).

In these cases we can directly compute the values of ‘a’ for which the condition satisfies.

The risk factor only depends on conditional statements. It should also depend on assignment statements since as soon as a variable is assigned a value or another variable, its mutability trait may vanish and hence its risk factor since it is always going to attain the assigned value despite of what user gives it initially. This means that variable dependencies [10] hold important place in determining risk factor.

IV. METHODOLOGY

To address to problem of time complexity inculcated due to execution overhead, a data structure of each variable is maintained having its name and value details along with the range of values it can take depending upon various conditional statements encountered. A flag is also maintained to keep a track if the variable is user input or otherwise. This eliminates the problem of unnecessary execution cycles since after incrementing the value of any variable during the evaluation of a conditional statement, we do not have to re-execute the whole program again as we already have the information of permitted range of values of the variable in its data structure based on previous conditional statements.

The condition in a conditional statement can be evaluated directly instead of doing Basic Search Procedure, if that condition is of lower degree. For example, if the condition is (a>100), with present value of a being say 1. Then for truthfulness of this condition, a’s can be directly computed as 101, instead of incrementing the value of a by 1 using Basic Search Procedure. Here, the ‘lower degree’ can be kept as 2, or may be 3.

To include the involvement of assignment statement in risk factor, value of risk factor is updated besides conditional statements, depending on the type of assignment being done.

We have implemented our automated test data generation software using a non-object-oriented language, C. Object oriented approach for Aspect Oriented Programming(AOP) can be referred from [11], [12].

V. EXPERIMENTAL SETUP

The tool that we have used to implement this research work is “gcc compiler in LINUX environment” which is used to compile codes written in C programming language. The following are the header files used in our code.

```c
#include<string.h>
#include<stdio.h>
#include<stdbool.h>
#include<ctype.h>
#include<math.h>
#include<stdlib.h>
#include<string.h>
```

Our working software test data generator code incorporates following types of statements and data structures:

1) **Variable structure**: A structure is defined for each variable declaration that is encountered while reading the text file of the source code. The structure includes the following parameters:
   2) name ➔ name of variable encountered
   3) value ➔ current value of the variable(initialized to 0)
   4) hi ➔ the highest allowed value of the variable corresponding to that structure.
   5) lo ➔ the lowest allowed value of the variable corresponding to that structure.
   6) scanflag ➔ 1 for user input, 0 otherwise.

7) **Declaration statements**: Whenever a declaration statement is encountered, a ‘variable structure’ is declared (as explained above) assigning lexeme of the variable to the structure’s member ‘name’. ‘value’ and ‘scanflag’ are initialized to 0. ‘hi’ is initialized to highest value of integer and ‘lo’ to the lowest.

8) **Assignment statement**: Whenever an assignment statement is encountered, the statement is parsed, the LHS of ‘=’ is searched in the array of structures of variables and is assigned the ‘value’ corresponding to the RHS which can either be a number or a variable. In case of a variable, it is searched in array of structures, its ‘value’ is extracted and assigned to the LHS variable. The corresponding structure member ‘scanflag’ of LHS variable is set to 0.

9) **User input statements**: Whenever a user input statement (scanf) is encountered, the statement is parsed and the variable names are searched in the array of structure of variables, comparing the variable names with the member ‘name’ of the structure. The corresponding structure's member ‘scanflag’ is set to 1.

10) **Conditional statement**: Whenever the conditional statement (‘if’) is encountered, the following steps are to be followed. Parsing the argument and finding out the operator. Finding out whether the LHS and the RHS of the operator are variables or numbers and if variables,
whether it is a user input or not.

A. Improvement in Basic Search Procedure

The working of our software is as follows. Each statement of the source code is executed. If the encountered statement is declaration, assignment or user input type, the above mentioned steps are followed for each such statement. If it is a conditional statement, value of a variable in the condition is incremented/decremented by 1 if it lies between ‘lo’ and ‘hi’, to satisfy the condition. If while incrementing or decrementing the value of variable, the condition is satisfied. Then that new value is assigned to the variable and the execution proceeds. Note that unlike Basic Search Procedure, after each increment/decrement, all the statements above it are not executed to check if path is violated. This is because the range of values in variable’s data structure stores the possible values of the variable for which path won’t be violated. But if, while incrementing or decrementing value of variable, one of its extreme (hi or lo) is reached without satisfying the condition, then next variable in the condition is picked and the procedure is repeated. This way, re-execution of the whole program again is prevented, to check if with this new value of the variable, any previous conditional statement is violated. If all the variables of the condition are explored this way without finding out any solution, then the process is terminated as no test case can lead to the desired execution path. One more important thing is risk factor. Its value is used to pick up variables first, whose risk factor values are high. This increases the chances of finding the correct variable to mutate in order to meet the required condition of the expression.

VI. RESULTS AND DISCUSSION

Following are some example codes whose execution time of test data generation is evaluated using our implemented software for Automated Test Data Generation. There are two outputs for each example. One consisting of normal Basic Search Procedure [1] and the other as a result of the improvement mentioned in this research paper. Time is computed using the <sys/time.h> library of C language.

Example 1:
1. int a;
2. int b;
3. a=0;
4. scanf("%d",&a);
5. scanf("%d",&b);
6. if(a>0);
7. if(b<12);
8. if(a>b);

OUTPUT OF BASIC SEARCH PROCEDURE CODE
Total execution time = 0.000899 seconds
OUTPUT OF IMPROVED CODE
Total execution time = 0.000644 seconds
Improved code time/ original code time=1.39

Example 2:
1. int a;
2. int b;
3. a=0;
4. b=50;
5. scanf("%d",&a);
6. scanf("%d",&b);
7. if(a>0);
8. if(b<12);
9. if(a>b);

OUTPUT OF BASIC SEARCH PROCEDURE CODE
Total execution time = 0.001242 seconds
OUTPUT OF IMPROVED CODE
Total execution time = 0.000386 seconds
Improved code time/ original code time=3.2

Example 3:
1. int a;
2. int b;
3. a=0;
4. b=50;
5. scanf("%d",&a);
6. scanf("%d",&b);
7. if(a>0);
8. if(b<12);
9. if(a>b);
10. if(a>1);
11. if(a>0);
12. if(b<12);
13. if(b<11);
14. if(b<10);
15. if(a>b);

OUTPUT OF BASIC SEARCH PROCEDURE CODE
Total execution time = 0.001694 seconds
OUTPUT OF IMPROVED CODE
Total execution time = 0.000196 seconds
Improved code time/ original code time=8.6

In all the above examples, negative difference between improved code execution time and basic search procedure execution time is due to elimination of execution cycles after each increment/decrement of the variable. Let’s discuss using Example 3 above, how test data is generated using Basic Search Procedure and Improved Search procedure respectively and analyze how improved procedure reduces execution time.

Here the required path assumed is execution of all the instructions. Initially after multiple assignments of variables ‘a’ and ‘b’, user is asked for their values as input (Lines 8 and 9). This leads to marking the variables as ‘user input’ since only such variables can be altered. Line 10 states the first conditional statement, a>1. Last assigned value of ‘a’ was 2 (Line 6) and 2>1 satisfies this condition so control moves to next conditional statement (Line 11) which it also satisfies. Control now moves to line 12. Last assigned value of ‘b’ was 45 (Line 7). 45 is not less than 12, hence value of ‘b’ is decremented by 1. Now, in basic search procedure, on each such decrement, all the statements above it are executed to check if this new value of variable ‘b’ (i.e. 44) violates any condition. In this example, none of the previous conditions are violated on decrementing the value of ‘b’ by 1, ‘b’ can be set to 44 if it satisfies the current condition. Since 44 is not less than 12, ‘b’ is again decremented and again all the statements above it are checked for validity. At last b is set to 11 which satisfies the current condition.
All these redundant executions are eliminated in our improved search procedure. In each variable’s data structure, its range of values is stored. Before executing line 10, range of values of ‘a’ is all integer values. Lines 10 and 11 are executed same as Basic search procedure. In line 12, since condition is false, value of ‘b’ is decremented by 1. But this time, statements above it are NOT re executed to check if the path is violated due to this change. This is automatically done by the range of values we already have in b’s data structure. So we just need to check if this new value of b (i.e. 44) lies in this range. Rest of the statements are executed similarly to generate test data for required path. The reduction in execution time value is indicative of these eliminated cycles.

VII. CONCLUSION

As evident from the generated results, we conclude that the improved code generates test cases faster than the original code. The difference depends on the size of the source code. The more the size of source code, the more is the execution time ratio of the improved code and the original code, and hence greater is the improvement. This improvement is more with complex dependencies, with the source code. So we need to check if this new value of b (i.e. 44) lies in this range. Rest of the statements are executed similarly to generate test data for required path. The reduction in execution time value is indicative of these eliminated cycles.

REFERENCES


Abhimanyu Bhatt was born and brought up in Jaipur, Rajasthan, India. He was born on 14th October 1993. He is a fourth (and last) year undergraduate student of Birla Institute of Technology and Science, Pilani, Rajasthan, India in the field of Information Systems. He has done a two months summer internship in IT Development team. His interests lie in Data Structures Algorithms and Databases.