Automatic Program Based Test Data Generation

Mohd Nazim¹, Manoj Yadav²

¹Sr. Lecturer in Dept of C.S.E, NICT, Near Kalindi Kunj, Jamia Nagar, Okhla, New Delhi-25, India
²Assistant Professor, Dept. of CSE, Al Falah University, Dhau, Faridabad (Haryana), India

Abstract- To minimize the cost of the manual software testing and to increase the reliability of the testing processes, it is necessary to automate it. The automatic test data generator is an important component that automatically generates test data for a given program. The random testing can be outperformed by both concolic and search based testing. There are so many proofs to prove it. Instead of it, there is not so much work but very little to show the capability of either concolic testing or search based testing with entire real world software applications. An overview of automatic test data generation is given in this paper. The basic knowledge of concolic tool, CUTE, and a search based tool, AUSTIN is also being covered. The main aim of this paper is to get the basic concept of automated program based test data generation research. The syntax based and GUI-based data generation is not considered in this paper.

Keywords-- Concolic Testing, Search based testing, Test Case, CUTE, ASUTIN, Test Data Generator, Program Analyzer, Path selector, Random test data generation, Goal oriented test data generation, Static and dynamic test data generation.

I. INTRODUCTION

A challenge for IT industry is to develop software system that meets business needs [1]. To develop software free from bugs and defects, the software testing is used. Software testing is performed to support quality assurance [2]. By using an effective test method, quality software can be developed. We can say that Software Testing is an important part of the Software Development Life Cycle. The 50 percent of the total cost of the development of any software project is of software testing, and it could be minimized through the automated testing. There are three fundamental steps of software testing:

1) The design of test cases
2) The execution of these test cases
3) The determination of the correctness of the output produced.

The determination of the correctness of the output produced by the program under test cannot be automated without an oracle, which is rarely available. Fortunately, the problem of generating test data to achieve widely used notions of test adequacy is an inherently automatable activity [3].

An important part of software testing is test data generation. It is the process of creating a set of data for testing the adequacy of new or revised software applications. It may be the actual data that has been taken from previous operations or artificial data created for this purpose. Test data can be generated either based on specification [7, 8, and 9] or code. In this research, the basic focus is on code based test data generation. What we need is 100% automated testing to reduce overall cost of software development with high quality. A number of test data generation techniques such as random test data generator, path oriented test data generator, goal oriented test data generator and intelligent test data generator have been automated [4].

II. BASIC CONCEPT

The two widely testing methods have been concern in this paper.

1. Concolic testing
2. Search based testing

Concolic testing [34], [15], [8] originates in the seminal work of Godefroid et al. on Directed Random Testing [15]. Concolic execution merges symbolic [21] and concrete execution. The symbolic exploration of a program is driven by the concrete execution, and to simplify the path constraints produced by the symbolic execution, the dynamic variable values (which are achieved by real program execution) can be used.

Search based testing [26] formulates the test data adequacy criteria as objective functions, which can be optimized using Search Based Software Engineering [10], [17].

The space of possible inputs to the program under test is known as search space. The particular test capability measures of interest captured by the objective function.

This process has been used by the many types of testing, like functional [6] and non-functional [38] testing, mutation testing [4], regression testing [39], test case prioritization [24], [38], and interaction testing [11]. However, the most studied form of search based testing has been structural test data generation [30], [22], [32], [29], [37], [35], [18].
The purpose of this paper is to give the answers to the questions about the effectiveness of concolic and search based tools when applied to real world software applications, and the time taken by the concolic and search based tools to achieve certain levels of coverage.

It is necessary for the automated test data generation approaches to be evaluated using realistic unordinary programs without any human intervention to smooth over the possible encountered ‘real world’ challenges.

Since 1970s, the researchers show their interest in the automated structural test data generation. Currently, in the present decade, the two approaches have been emerged.

- Symbolic execution, It is the basis of concolic testing
- A method which reformulated the problem of executing a path through a program with floating point inputs into objective functions [30], known as search based testing.

### A. Concolic Testing

The idea of the symbolic execution is the basis of the Concolic testing. The symbolic execution for a given path in a program involves the following:

- Constructing a path condition, and
- A set of constraints (i.e. input variables) that describe on the execution of the path.

Example 1: Showing the symbolic execution (here the branch predicate is linear)

```c
void t1( int i, int j )
{
    i += 1; j -= 5;
    if (i == j)
        // ... 
}
```

To execute the `if` statement as true, the path condition will be \( i_0 + 1 = j_0 - 5 \). The \( i_0 \) and \( j_0 \) are the symbolic values of the input variables \( i \) and \( j \) respectively. To derive concrete input values, the path condition is solved by a constraint solver.

The path condition may be unsolvable, if it has the expressions whose handling is not possible by the constraint solvers. Most frequently, this may be the case with floating-point variables, or non-linear constraints.

Example 2: Showing the concolic execution and search based testing (here the branch predicate is non-linear)

```c
void t2( int p, int q )
{
    if ( p * q < 100 )
        // ...
}
```

In the above program, in the if condition, a non-linear predicate is appearing; therefore linear constraint solver might faces the problems.

Some of the problems of non-linearity can be alleviate by concolic testing by combining concrete execution with symbolic generation of path conditions. The main motive is to substitute the sub-expressions with concrete values to simplify the path conditions. The non-linear sub-expressions in a path condition can be removed by this process to make them amenable to a constraint solver.

Concolic execution originated from the work of Godefroid et al. [15]. The term concolic was coined by Sen et al. [34] in their work introducing the CUTE tool, which is based upon similar principles.

**The CUTE Tool:** Let assume that, for the execution of the true branch of the program of Example 2, an execution of the path is needed. With the help of some inputs, CUTE executes the program. For the default, the execution of a function is performed with all variables of primitive type set to zero. At the place of it, the random values can also be used.

Let consider that for the execution of the function, the random values 536 and 156 have been chosen for \( p \) and \( q \) respectively. Suppose the path taking the false branch is executed. The condition for the path is \( p_0 \cdot q_0 < 100 \). We know that this constraint is non-linear; therefore, \( p_0 \) will be replaced by its concrete value i.e. 536 with the help of CUTE. Now the path condition becomes linear i.e. \( 536 \cdot q_0 < 100 \). To find the appropriate value for \( y \), this linear path condition is passed to the constraint solver. CUTE try to execute all program paths. The path which is pass over with all zero or random inputs, is executed first. The next path taking the alternative branch at the last executed decision statement of the path. Therefore the new path condition is same as the previous, but with the last constraint negated. This allows the replacement of sub-expressions in the new path condition with sensible substantial (solid or factual) values just like in the example 2.

In the programs, which have unbounded loops, there may be an infinite number of paths; therefore CUTE may keep spreading out the body of the loop infinitely many times.
B. Search Based Testing

Like symbolic–execution–based testing, the first suggestion of optimization as a test data generation technique also emerged in the 1970s, with the seminal work of Miller and Spooner [30]. Miller and Spooner define the reformulation of the executed path as an objective function. Using the optimization techniques, the optima (i.e. the test data which execute the path) of this function can be found out.

The non-linear constraints shows some problems due to the using of optimizing search technique instead of a constraint solver.

The suggestions of Miller and Spooner were not subsequently taken up until Korel developed them further in 1990 [22], when he proposed the use of a search technique known as the ‘alternating variable method’. Since then the ideas have been applied to other forms of testing [6], [38], [4], [42], [24], [36], [11], using a variety of optimizing search techniques, including genetic algorithms [32], [29], [37]. The objective function has been further developed to generate test data for a variety of program structures, including branches, as well as paths [39].

The AUSTIN Tool: To generate branch sufficient or satisfactory test data for C programs, a tool is used named AUSTIN. To cover a target branch, this tool does not try to execute special or particular paths. For the Daimler Evolutionary Testing System, Wegener et al. [37] introduced an objective function used by AUSTIN. This function evaluates an input against a target branch using two matrices

(i) Approach level
(ii) Branch distance.

Example 3: Showing the objective function calculation for the AUSTIN tool.

```c
void t3 ( int I, int j, int k )
{
    (1) If ( i == j )
    (2) If ( j == k )
    (3) If ( I == k )
    (4) // …
}
```

The number of nodes that were not executed by a particular input on which the branch is control dependent is recorded by the approach level.

In Example 3,

- When the false branch of statement 1 is executed by an input, the approach level for executing the true branch of statement 3 is equal to 2
- When false branch of statement 2 followed the execution of the true branch of statement 1, the approach level for executing the true branch of statement 3 is equal to 1
- If the statement 3 is reached, then the approach level for executing the true branch of statement 3 is equal to 0.

The condition of the decision statement at which the flow of control deflected away from the current “target” branch is used to calculate the branch distance.

Like CUTE, AUSTIN also begins with all primitives set to zero.

If the execution of the target is not performed, the AVM perform 'pattern moves' guided by the objective function by cycles through each input of primitive type. The random values are used for restarting the search, if during a complete cycle of adjustments; there is no improvement in the objective value.

Let consider that in the Example 3; to executes the true branch of statement 3, with the input \(i = 2, j = 5\) and \(k = 10\), the execution of the program is performed. AVM performed the exploratory moves by taking the first variable, \(i\). The decrement and increment in \(i\) is of small amount \(\delta\).

For integers, the value of \(\delta\) is 1 while, for floating point variables, the value is 0.1 through the increased value, \(i’\) comes closer to ‘\(j’\'. it results in a better objective value.

This pattern moves is performed by an AVM as long as the objective function repeated to give rise in an improved value.

III. THE CONTROL FLOW GRAPH

It is a graphical representation of a program. There are many different definitions of the control flow graph.
Depending on the properties of the language to model, the definition might differ [5, 6]. The definition used here has been inspired by Beizer [1] as well as Korel et al. [5, 8]. Figure 3 shows a sample flow graph and its corresponding program.

There are two special nodes in each flow graph, one is entry node (s) and second one is exit node (e). Each and every node could be stated as a key or primary block. This block is an uninterrupted continuous series of instructions. At the beginning of the block, the flow of control enters while at the end, the flow of control leaves the block. Its mean, we can say that whole block is executed on the execution of any statement of this block.

Besides it, the program targeting an instruction within the block, have no jumps. Sometimes the node is considered as a condition whiles the edges as the branches. A complete path is a path that starts with the entry node and last at the exit node; otherwise we called it an incomplete path or a path segment.

The transformation of information between two nodes n and m is represented through an edge between n and m. The edges are well labeled with a condition or a branch predicate which might be the empty predicate. The condition of the edge should be hold to traverse the edge. Two or more edges with a true condition could not be hold by a node at a particular time.

Let consider an example of finding the type of the triangle when the sides are given.

```c
int type_of_tri (int x, int y, int z)
{
    int type = PLAIN;
    1. if ( x < y )
    2.    swap ( x, y );
    3. if ( x < z )
    4.    swap ( x, z );
    5. if ( y < z )
    6.    swap ( y, z );
    7. if ( x == y )
         { 
             8. if ( y == z )
             9.    type = Equilateral triangle;
                  else
             10. type = Isosceles triangle;
         }
    11. else if ( y == z )
    12.    type = isosceles triangle;
    13. return type;
}
```

Fig 2: Program to find out the type of a triangle.

The flow graph of the above program is given below.
Unspecific path: A path with some path segments missing. For example, in above figure, path (3, 10, and 13) is an unspecific path segment.

Let consider a specific path p= (3,4,5,7,8,10,13) in the above figure, since the path (3,10,13) is an unspecific path segment, therefore the complement of it will be (4,5,7,8).

In Fig 3, the path begins at the start node s and at the exit node e, it ends up. The closure of this path would be all paths between the both entry and the exit node. For example, The closure (1,2,13) have all parts that begin from node 1 and end at node 13 and have 2 as the second node. Let consider an another path, (3,10,13). The closure of this path is the set of the following paths:

$$\{ (3,5,7,8,10,13), (3,4,5,7,8,10,13), (3,6,7,8,10,13), (3,4,5,6,7,8,10,13) \}$$

IV. AUTOMATIC TEST DATA GENERATOR SYSTEM
There are three main parts of this system:
1. Program analyzer
2. Path selector
3. Test data generator

This paper will focus only on the generator and the selector. So I just assume that the working of analyzer is proper.

1. The Test Data Generator
The complete information about the program (data dependence graphs, control flow graphs etc.) is provided by the program analyzer. The paths for which the test data generator derive input values are identifies by the path selector. The paths could be either specific or unspecific, on the basis of type of generator system.

Our aim is to find input values to traverse the paths received from the selector. This can be performed in two steps:

1. Finding the path predicate for the path.
2. Solving the path predicate.

At a time only one branch predicate solved out by some techniques due to the complexity of the derived equation systems. It causes the performance loss.

For example: for a given path $p=(1,2,3,5,6,7,8,10,13)$ in the above fig 3, find a path predicate.

First of all we have to understand result getting out after the execution of the program on the input (5,4,4). We find that path q is traversed. Now construct a path predicate $Q$. the Q is a conjunction of all branch predicates encountered during the traversing of the path.
Q = (x > y) ∧ (x ≤ z) ∧ (y > z) ∧ (x = y) ∧ (y ≠ z)

Here we assume, x=5, y=4, and z=4 to check whether Q holds.

Q = (5 > 4) ∧ (5 ≤ 4) ∧ (4 > 4) ∧ (5 = 4) ∧ (4 ≠ 4)

The execution of the nodes of 1,2,6, and 10 is ignored during the construction of the path predicate.

As a result, it ended up incorrectly by not allowing the side effects propagate over the path predicate. Let consider that the program is executed on the inputs (5,4,4) and when it reaches node 7, we pause the execution.

Now as we know that, before reaching node 7 the statement swap(x, y) was executed, therefore we should use x=4 and y=5.

The swap(x, y) was not considered in the case of path predicate Q. Therefore still x=5 and y=4.

The following code was executed:

```c
1 (x > y) int type = PLAIN;
3 (y ≤ z) swap(x, y);
5 (y > z)
7 (x = y) swap(y, z);
8 (y ≠ z)
13 T type = ISOSCELES;
```

Hence, do the following in order to adjust the branch predicates to take the data dependence into account. Firstly execute the code of the first node and then update all the succeeding rows. Continue with the next row until and unless all rows have been processed.

Now each row corresponds to a branch predicate which is adjusted according to the execution of nodes 1,2,6, and 10. In result, the new path predicate

P = (x > y) ∧ (y ≤ z) ∧ (x > z) ∧ (y = z) ∧ (z ≠ x).

On substituting x,y and z as 5,4, and 4 respectively, we observed that P possess.

P = (5 > 4) ∧ (4 ≤ 4) ∧ (5 > 4) ∧ (4 = 4) ∧ (4 ≠ 5).

Therefore we can say that for p=(1,2,3,5,6,7,8,10,13), P is a valid path predicate.

There are the three classes of construction of a test data generator.

- Random test data generation(randomly generate test data)
- Goal oriented test data generation(generate test data for an unspecific path)
- Path oriented test data generation(generate test data for a specific path)

1.1. Static and Dynamic Test Data Generation

By using a transformed system of equations as in example 4 we can use either symbolic execution or actual execution, i.e. the generation occurs either statically or dynamically.

In the execution of a program symbolically, variable substitution is used in place of actual values variables. This type of execution was used by most approaches in the 70’s and end up with an expression in terms of input variables.

For example:

```c
let consider that a and b are input variables
r := p + q;
s := p - q;
t := r*s;
```

In the above code, e will contain p*p - q*q. The plenty of computer resources are required by this technique and also puts a lot of restrictions on the program.

The actual execution is the opposite of the symbolic execution. In this technique, the program is run with some randomly selected input instead of using substitution of variable. Consequently, values of variables are known at any time of the execution.

By monitoring the program flow the system can determine if the intended path was taken. If not, it backtracks to the node where the wrong direction is taken by flow. Using different kinds of search methods, by manipulating the input in a way that the intended branch is taken, the flow of the program can be altered.
This technique is quite expensive, and before a suitable input is found, it can require many iterations. The flow at an earlier point may change accidentally on changing.

1.2. Random Test Data Generation

It is the simplest method of generation techniques. It actually used to generate input values for any type of program. Since, a data type like an integer, string, or heap is just a stream of bits. Therefore, we can just randomly generate a bit stream for a function taking a string as an argument and let it represent the string.

On the adverse, random testing mostly does not effect efficiently in terms of coverage. Since it merely relies on probability it has quite low chances in finding semantically small faults [11], and thus accomplishes high coverage. A semantically small fault is a fault that is only disclosed by a small percentage of the program input.

1.3. Goal-Oriented Test Data Generation

This approach is much stronger than random generation. It generates input that traverses a given unspecific path u instead of letting the generator generate input that crosses from the entry to the exit of a program. Therefore, it is enough for the generator to find input for any path p ∈ u*. This shortens the risk of encountering relatively infeasible paths and provides a way to direct the search for input values as well.

There are two methods that use this type of technique:
- The chaining approach
- The assertion oriented approach

(it is an intersecting extension of the chaining approach)

They have all been implemented in the TESTGEN system [5, 9].

Chaining approach:
- It is the use of data dependence to find solutions to branch predicates.
- Its characteristic is to identify a chain of nodes that are vital to the execution of the goal node.
- This chain is built up iteratively during execution.
- It is hard to predict the coverage given a set of goals in this approach due to the uses of find-any-path concept.

Assertion oriented approach:
- It exactly utilizes the power of goal-oriented generation.
- Certain conditions, called assertions are inserted in the code. It may be either manually or automatically.
- When an assertion is executed it is supposed to hold, otherwise there is an error either in the program or in the assertion.

1.4. Path-Oriented Test Data Generation

It is the strongest approach between the above three approaches. It gives the generator just one specific path instead of selecting from a set of paths. So we can say that it is just like as a goal oriented test data generation, except for the use of specific paths. Successively this leads to a better prediction of coverage but on the other hand it is harder to find test data. CASEGEN [13] and TESTGEN [8] are two systems using this technique.

2. The Path Selector

The most important aspect of the effectiveness of the complete system is the selection of the paths.

We can get a set of test data that covers the program by selecting the paths. The number of paths has to be selected to rely upon the strength of the coverage criterion.

A list of most cited criteria is given below:
- Statement Coverage: it performs the execution of all statements in the flowgraph.
- Branch coverage: it encounter all branches in the program.
- Condition coverage: within each condition of the flowgraph, each and every clause must be executed to both true and false.
- Multiple Condition Coverage: during an execution, each combination of truth values of each clause of each condition must be executed.
- Path Coverage: Each path in the flowgraph is traversed by it.

V. CONCLUSION

The two approaches namely concolic (embodied in the CUTE tool) and the search based (implemented in the AUSTIN tool) has been examined by this paper which are used to automated structural test data generation. The result of this paper shows that to design the automatic test data generation tools robust, there are still so many challenges remaining. In automated test data generation, out of many open challenges, two may be most important for the future work. On a practical level, tools need to be able to prevent or recover from segmentation faults, so that they may continue the test data generation process to any effect. Secondly test data generation tools need to become much more heterogeneous in nature. In place of carrying out pure static or dynamic analysis, a combination of both is needed in order to control problems.
REFERENCES

[1] Automated Software Test Data Generation: Direction of Research (Hitesh Tabhildar1 and Bichitra Kalita2) 1Department of Computer Engineering and Application, Assam Engineering Institute, Guwahati, Assam 781003, India tabhil@rediffmail.com 2Department of Computer Application, Assam Engineering College, Guwahati, Assam 781003, India bichitra1_kalita@rediffmail.com


[3] Automated Test Data Generation for Coverage: Haven’t we solved this problem yet? Kiran Lakhota (King’s College London CREST centre Strand, London WC2R 2LS, UK) & Phil McMinn(University of Sheffield Regent Court 211 Portobello, Sheffield S1 4DP, UK).


[40] Jon Edvardsson. Dept. of computer and information science, Linkoping University, Sweden. E-mail:joned@ida.liu.se “A survey on automatic test data generation.”