A Survey on Automatic Test Data Generation[*]

Jon Edvardsson,
Dept. of Computer and Information Science,
Linköping University, Sweden

E-mail: `joned@ida.liu.se`

Abstract

In order to reduce the high cost of manual software testing and at the same time to increase the reliability of the testing processes researchers and practitioners have tried to automate it. One of the most important components in a testing environment is an automatic test data generator — a system that automatically generates test data for a given program. Through the years several attempts in automatic test data generations have been made. The focus of this article is program-based generation, where the generation starts from the actual programs. Thus, techniques such as GUI-based and syntax-based test data generation are not an issue in this article.

In this article I present a survey on automatic test data generation techniques that can be found in current literature. Basic concepts and notions of test data generation as well as how a test data generator system works are described. Problems of automatic generation are identified and explained. Finally important and challenging future research topics are presented.

1. Introduction

Software testing accounts for 50% of the total cost of software development [1]. This cost could be reduced if the process of testing is automated. One way to do this would be to generate input data to the program to be tested — program-based test data generation.

Through the years a number of different methods for generating test data have been presented. In 1996 Ferguson and Korel [5] divided these methods in three classes: *random*, *path-oriented*, and *goal-oriented* test data generation. This is the most appropriate classification in terms of test data generation, although the problem of path selection is not considered separately. The selection of a path can largely affect the whole process of test data generation.

Figure 1 models a typical test data generator system, which consists of three parts: program analyzer, path selector and test data generator. The source code is run through a program analyzer, which produces the necessary data used by the path selector and the test data generator. The selector inspects the program data in order to find *suitable* paths. Suitable can for instance mean paths leading to a high code coverage. The paths are then given as argument to the test data generator which derives input values that exercise the given paths. The generator may provide the selector with feedback such as information concerning infeasible paths.

The structure of this paper is as follows. In section 2 basic concepts and notions are explained. Section 3 discusses the test data generator system with focus on the generator and the path selector. The program analyzer is not further investigated in this article. In section 4 the problems I have identified in test data generation are discussed. Finally, in section 5 conclusions are made and future research topics are presented.

2. Basic Concepts

A program $\mathcal{P}$ could be considered as a function, $\mathcal{P} : S \rightarrow R$, where S is the set of all possible inputs and R the set of all possible outputs. More formally S is the set of all vectors $\mathbf{x} = (d_1, d_2, \cdots, d_n)$ such that $d_i \in D_{x_i}$ where D_{x_i} is the domain of input variable x_i.

An input variable x of $\mathcal{P}$ is a variable that either appears as an input parameter of $\mathcal{P}$ or in an input statement of $\mathcal{P}$, e.g. `read(x)`. Execution of $\mathcal{P}$ for a certain input $\mathbf{x}$ is denoted by $\mathcal{P}(\mathbf{x})$.

A *control flowgraph*, or just flowgraph when context is clear, is a graphical representation of a pro-

[*]Published as: J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the Second Conference on Computer Science and Engineering in Linköping*, pages 21–28. ECSEL, October 1999.

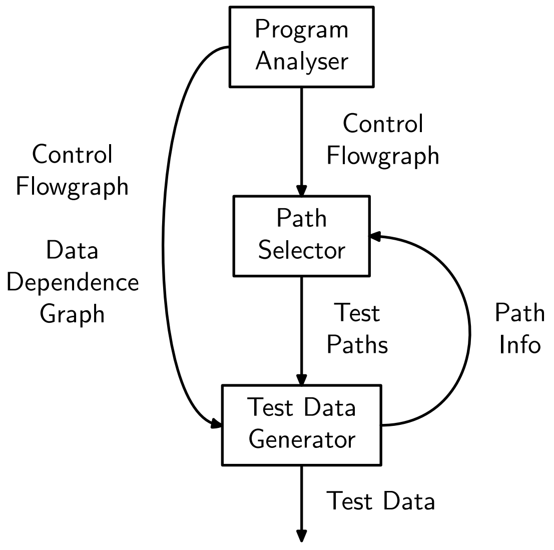

Figure 1. Architecture of a test data generator system.

```
int triType(int a, int b, int c) {
 1  int type = PLAIN;
 1  if (a < b)
 2    swap(a, b);
 3  if (a < c)
 4    swap(a,c)
 5  if (b < c)
 6    swap(b, c)

 7  if (a == b) {
 8    if (b == c)
 9      type = EQUILATERAL;
 .    else
 10     type = ISOSCELES;
 .  }

 11 else if (b == c)
 12   type = ISOSCELES;
 13 return type;
}
```

Figure 2. A program that determines the type of a triangle given its sides.

gram. There exist many different definitions on control flowgraphs throughout the literature. 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 flowgraph and its corresponding program.

A control flowgraph of a program $\mathcal{P}$ is a directed graph $G = (N, E, s, e)$ consisting of a set of nodes N and a set of edges $E = \{(n, m) | n, m \in N\}$ connecting the nodes. In each flowgraph there are two special nodes: one entry- and one exit-node, s and e respectively.

Each node is defined as a *basic block*, which is an uninterrupted consecutive sequence of instructions, where the flow of control enters in the beginning and leaves at the end without halt or possibility of branching except at the end. Intuitively this means that if any statement of the block is executed, then the whole block is executed. Furthermore, there are no jumps in the program targeting an instruction within the block.

An edge between two nodes n and m corresponds to a possible transfer from n to m. All edges are labeled with a condition or a *branch predicate*. The branch predicate might be the empty predicate which is always true. In order to traverse the edge the condition of the edge must hold. At any given time no node can have two or more edges with a condition yielding true (otherwise we would end up with a non-deterministic flowgraph). If a node has more than one outgoing edge we sometimes refer to the node as a condition and the edges as branches.

A *(specific) path* is a sequence of nodes $p = \langle p_1, p_2, \cdots, p_{q_p} \rangle$, where p_{q_p} is the last node of path p and $(p_i, p_{i+1}) \in E$ for $1 \leq i < q_p - 1$. Whenever the execution of $\mathcal{P}(\mathbf{x})$ traverses a path p, we say that $\mathbf{x}$ traverses p. A path is *(absolutely) feasible* if there exists an input $\mathbf{x} \in S$ that traverses the path, otherwise the path is *(absolutely) infeasible*. For a certain input $\mathbf{x}$, an absolutely feasible path p could be infeasible [12]. We say that p is infeasible *relative* to $\mathbf{x}$ or just relatively infeasible.

A path that begins with the entry node and ends with the exit node is called a *complete path*. Otherwise it is called an *incomplete path* or a *path segment*.

Let $p = \langle p_1, p_2, \cdots, p_{q_p} \rangle$ and $w = \langle w_1, w_2, \cdots, w_{q_w} \rangle$ be two paths then $pw = \langle p_1, p_2, \cdots, p_{q_p}, w_1, w_2, \cdots, w_{q_w} \rangle$ denotes the concatenation of p and w. Let $first(p)$ denote the first node p_1 of path p and let $last(p)$ denote the last node p_{q_p} of p. We say that two paths *connect* if $(last(p), first(w)) \in E$, where E is the set of edges.

If p and w are two specific paths (or path segements), we say that pw is an *unspecific path* if p and w do not connect. Moreover, we say that a path q *complements* pw if and only if pqw is specific.

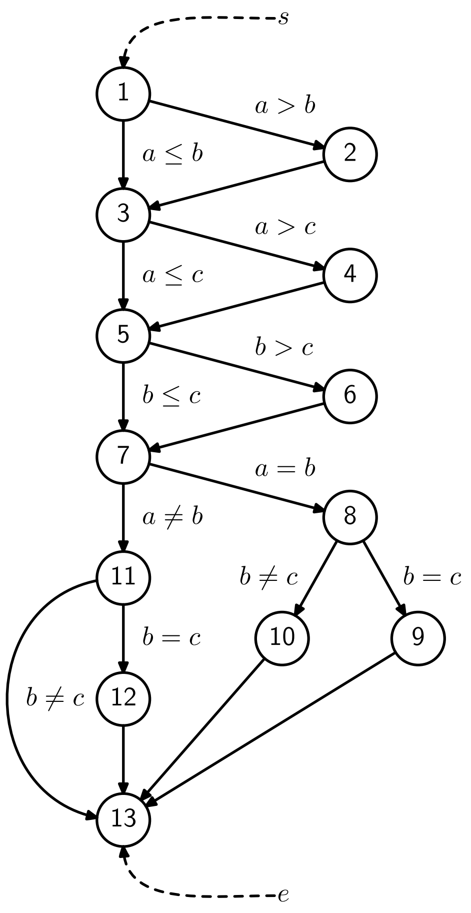

Figure 3. A flowgraph of the program in figure 2.

For an unspecific path $u = p_1 p_2 \cdots p_n$, where p_i is specific, we define the *closure* of u, denoted by u^*, as the set of all paths $p_1 q_1 p_2 q_2 \cdots q_{n-1} p_n$ such that q_i complements $p_i p_{i+1}$.

Example 1 Informally an unspecific path is a path with some path segments missing. For instance, $p = \langle 3, 10, 13 \rangle$ in figure 3 is an unspecific path segment composed by $\langle 3 \rangle$ and $\langle 10, 13 \rangle$. $\square$

Example 2 The complement of p in example 1 is the path segment $\langle 4, 5, 7, 8 \rangle$, since $\langle 3, 4, 5, 7, 8, 10, 13 \rangle$ is a specific path. $\square$

Example 3 Intuitively an unspecific path is a wild card for paths. The closure then represent a *listing* of those paths. For instance in figure 3 we have the path that begins in the start node and ends in the exit node $\langle s, e \rangle$. Its closure would then be all paths between the entry and exit node inclusively. The closure $\langle 1, 2, 13 \rangle^*$ contains all paths that start in node 1, end in node 13 and have 2 as the second node. The closure of path $\langle 3, 10, 13 \rangle$ is the set of paths $\{\langle 3,5,7,8,10,13 \rangle, \langle 3,4,5,7,8,10,13 \rangle, \langle 3,5,6,7,8,10,13 \rangle, \langle 3,4,5,6,7,8,10,13 \rangle\}$. $\square$

In order for execution to continue through a branch the corresponding branch predicate has to be true. Thus, to traverse a certain path a conjunction $P = c_1 \wedge c_2 \wedge \cdots \wedge c_n$ of branch predicates c_i must hold. We say that P is a *path predicate*.

3. An Automatic Test Data Generator System

A test data generator system consists of three parts: a program analyzer, a path selector, and a test data generator. In this article I will keep the focus on the selector and the generator. Therefore let us assume that the analyzer exists and works properly.

3.1. The Test Data Generator

At this point let us define the problem of automatic test data generation as follows: given a program $\mathcal{P}$ and a (unspecific) path u, generate input $\mathbf{x} \in S$, so that $\mathbf{x}$ traverses u.

This means that we can assume to have a program analyzer and a path selector such as in figure 1. The program analyzer provides all information concerning the program: data-dependence graphs, control flowgraphs etc. In turn the path selector identifies paths for which the test data generator will derive input values. Depending on the type of generator system paths could either be specific or unspecific.

Our goal is to find input values that will traverse the paths received from the selector. This is achieved in two steps. First find the path predicate for the path. Second, solve the path predicate in terms of input variables. The solution will then be a system of (in)equalities describing how input data should be formed in order to traverse the path.

Having such a system we can apply various search methods to come up with a solution. Examples of search methods are alternating variable, simulated annealing, and different heuristics based on equation-rewriting systems [5, 14, 3].

Due to the complexity of the derived equation systems some techniques solve one branch predicate at a time. This leads to a loss of performance since it makes it necessary to check that violations of other previously solved predicates do not occur.

Example 4 Find a path predicate for $p = \langle 1, 2, 3, 5, 6, 7, 8, 10, 13\rangle$ in figure 3.

Before getting in to the details of finding such a path predicate we will see what happens if we execute the program on the input $(5, 4, 4)$. Doing this we find that path p is actually traversed. Let us now construct a path predicate P' that is a conjunction of all branch predicates encountered when traversing the path.

$$P' = (a > b) \wedge (a \le c) \wedge (b > c) \wedge (a = b) \wedge (b \ne c)$$

By letting $a = 5$, $b = 4$, and $c = 4$, we check whether P' holds. Since $(5, 4, 4)$ *does* traverse the path p, then any path predicate corresponding to p *must* hold.

$$P' = (5 > 4) \wedge (5 \le 4) \wedge (4 > 4) \wedge (5 = 4) \wedge (4 \ne 4)$$

Plainly we see that this is not the case. But why? When we constructed the path predicate we ignored the execution of the nodes of 1, 2, 6, and 10. Consequently, by not letting the side effects propagate over the path predicate it ended up incorrectly. For instance, assume that the program is executed on input $(5, 4, 4)$ and that we pause the execution when it reaches node 7. Now, at this point we should expect $a = 4$ and $b = 5$, because before reaching node 7 the statement `swap(a,b)` was executed and thus setting $a = 4$ and $b = 5$. In the case of path predicate P' the `swap(a,b)` was not considered and therefore a and b still are equal to 5 and 4 respectively.

$$\begin{bmatrix} 1 & (a > b) & \texttt{int type = PLAIN;} \\ 3 & (a \le c) & \texttt{swap(a, b);} \\ 5 & (b > c) & \\ 7 & (a = b) & \texttt{swap(b, c);} \\ 8 & (b \ne c) & \\ 13 & \top & \texttt{type = ISOSCELES;} \end{bmatrix}$$

The above structure illustrates the data dependencies among the branch predicates. Each row depends upon execution of itself as well as the previous rows. For instance, before checking whether $(a = b)$ in row 7 holds, the following must be executed: `int type = PLAIN; swap(a, b); swap(b, c);`.

Thus, in order to adjust the branch predicates to take data dependence into account do the following. Start with the first row and execute its code. Update all succeeding rows (including the current condition) according to the side effects. Continue with the next row until all rows have been processed.

$$\ldots \overset{2\ iter}{\rightsquigarrow} \begin{bmatrix} 1 & (a > b) & \\ 3 & (b \le c) & \\ 5 & (a > c) & \\ 7 & (b = a) & \texttt{swap(a, c);} \\ 8 & (a \ne c) & \\ 13 & \top & \texttt{type = ISOSCELES;} \end{bmatrix}$$

$$\ldots \overset{4\ iter}{\rightsquigarrow} \begin{bmatrix} 1 & (a > b) \\ 3 & (b \le c) \\ 5 & (a > c) \\ 7 & (b = c) \\ 8 & (c \ne a) \\ 13 & \top \end{bmatrix}$$

Now each row corresponds to a branch predicate which is adjusted according to the execution of nodes 1, 2, 6, and 10. This gives us the new path predicate $P = (a > b) \wedge (b \le c) \wedge (a > c) \wedge (b = c) \wedge (c \ne a)$. If we again substitute $a = 5$, $b = 4$, and $c = 4$ we see that P holds.

$$P = (5 > 4) \wedge (4 \le 4) \wedge (5 > 4) \wedge (4 = 4) \wedge (4 \ne 5)$$

Thus, P is a valid path predicate for $p = \langle 1, 2, 3, 5, 6, 7, 8, 10, 13\rangle$. $\square$

Basically there are three approaches when constructing a test data generator: randomly generate test data, generate test data for an unspecific path, or generate test data for a specific path. These approaches fall into the three classes of *random*, *goal-oriented*, and *path-oriented* test data generation. Each of these can be implemented statically or dynamically.

3.1.1. Static and Dynamic Test Data Generation

To come up with 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 70's most approaches made use of symbolic execution. Executing a program symbolically means that instead of using actual values variable substitution is used. The idea is to end up with an expression in terms of input variables. For instance, let `a` and `b` be input variables.

```
c := a + b;
d := a - b;
e := c*d;
```

Then `e` in the above code will contain `a*a - b*b`. One realizes that this technique requires plenty of computer resources, e.g. expressions have to be resolved and transformed. It also puts a lot of restrictions on the program. For instance, how should function calls to modules where there is no access to source code be handled? Furthermore, symbolic execution also implies that a symbolic evaluator for the particular language is built which indeed requires a great amount of work.

But there are gains as well. For instance, it requires no violation checks of branch predicates since all can be solved at once.

The opposite of symbolic execution is actual execution. Instead of using variable substitution run the program with some, possibly randomly, selected input. 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 flow took the wrong direction. Using different kinds of search methods the flow can be altered by manipulating the input in a way that the intended branch is taken. This technique is quite expensive. It can require many iterations before a suitable input is found. Upon changing the flow at a particular node, the flow at an earlier point may accidently change. Actual execution also suffers from the speed of execution for the program to analyze. Besides, to monitor the program flow code is instrumented, i.e. to put probes in the program to ascertain path traversal.

In an article by Gupta et al. [7] a hybrid of the two forms is presented. It combines the gains of both kinds, thus it does not requires as many executions to find an appropriate input.

3.1.2. Random Test Data Generation

Random testing is the simplest method of generation techniques. It could actually be used to generate input values for any type of program since, ultimately, a data type such as integer, string, or heap is just a stream of bits. Thus, for a function taking a string as an argument we can just randomly generate a bit stream and let it represent the string.

On the contrary, random testing mostly does not perform well in terms of coverage. Since it merely relies on probability it has quite low chances in finding semantically small faults [11], and thus accomplish high coverage. A semantically small fault is such a fault that is only revealed by a small percentage of the program input. Consider the following piece of code:

```
void foo(int a, int b) {
  if (a == b) then
    write(1);
  else
    write(2);
}
```

The probability of exercising the `write(1)` statement is $1/n$, where n is the maximum integer, since in order to execute this statement variables `a` and `b` must be equal. We can easily imagine that generating even more complex structures than integers will give us even worse probability.

Often evaluation of search methods uses random testing as a benchmark [3, 2, 5], since it is considered to be of the lowest acceptance rate.

3.1.3. Goal-Oriented Test Data Generation

The goal-oriented approach is much stronger than random generation, in the sense of providing a guidance towards a certain set of paths. Instead of letting the generator generate input that traverses from the entry to the exit of a program, it generates input that traverses a given unspecific path u. Because of this, it is sufficient for the generator to find input for any path $p \in u^*$. This in turn reduces the risk of encountering relatively infeasible paths and provides a way to direct the search for input values as well.

Two methods using this technique have been found: the chaining approach and assertion-oriented approach. The latter is an interesting extension of the chaining approach. They have all been implemented in the TESTGEN system [5, 9].

Typical for the chaining approach is the use of data dependence to find solutions to branch predicates. The characteristics of chaining 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.

Since this method uses the *find-any-path* concept it is hard to predict the coverage given a set of goals.

Assertion-oriented testing truly utilizes the power of goal-oriented generation. Certain conditions, called assertions are either manually or automatically inserted in the code. When an assertion is executed it is supposed to hold, otherwise there is an error either in the program or in the assertion. For instance, with the following code:

```
void fie(int a) {
  int b = (a+1)*(a-1);
  assert(b != 0);
  write(1/b);
}
```

we say that before executing `1/b` the variable `b` must not be zero. The goal of assertion-oriented generation is then to find *any* path to an assertion that does not hold.

An advantage with assertion-oriented testing is that the oracle (see section 4.7) is given in the code. That is, in all the other methods the expected value of an execution of the generated test data has to be calculated from some other source than the code. With assertions this is not necessary since expected value is provided within the assertion.

3.1.4. Path-Oriented Test Data Generation

Path-oriented generation is strongest among the three approaches. It does not provide the generator with a possibility of selecting among a set of paths, but just one specific. In this way it is the same as a goal-oriented test data generation, except for the use of specific paths. Successively this leads to a better prediction of coverage. On the other hand it is harder to find test data.

CASEGEN [13] and TESTGEN [8] are two systems using this technique. Since they are solely based on the control flow graph they often lead to selection of infeasible paths (both relatively and absolutely).

DeMillo and Offutt [4] have proposed a constraint-based test data generation method. It is focused on fault-based testing using mutants, i.e. a deliberate change in the source code. However, it is not clear how paths are selected and since this technique is somewhat similar to assertion-oriented testing it could fit under goal-oriented test data generation as well.

3.2. The Path Selector

The other component of the test data generator system in figure 1 is the path selector. Effectiveness of the whole system is highly dependent on how paths are selected.

In path selection we would rather define the automatic test data generation problem as given a program $\mathcal{P}$ find the least set of paths in $\mathcal{P}$ such that it meets a specified coverage criterion.

This means, not only it is vital to find test data given a path, but also to find good test data. By carefully selecting paths we can come up with a set test of test data that covers the program. The stronger the coverage criterion the more paths have to be selected. Below is a list of the most cited criteria.

Statement coverage Execute all statements in the flowgraph.

Branch coverage Encounter all branches in the program, e.g. the predicate of an if-statement must be evaluated to both `true` and `false`.

Condition coverage Each clause within each condition of the flowgraph must be executed to both `true` and `false`, some time during execution.

Multiple-condition coverage Each combination of truth values of each clause of each condition must be executed during execution.

Path coverage Traverse each path in the flowgraph.

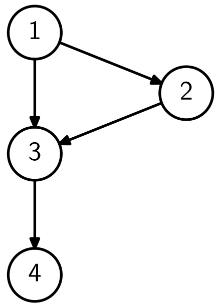

Figure 4. A sample control flowgraph

Statement coverage is the weakest of them, though it is not obvious. Figure 4 illustrates the weakness: to achieve statement coverage it is enough to select the path $\langle 1, 2, 3, 4\rangle$, on the other hand, to achieve branch coverage the path of $\langle 1, 3, 4\rangle$ must also be traversed. This means that in statement coverage it is *possible* that faults depending on non-execution of node 2 are left unexplored.

The stronger criteria of condition, multiple-condition and path coverage are often infeasible to achieve for programs of more than moderate complexity, and thus branch coverage has been recognized as basic measure for testing[12].

Up to my knowledge only one method of how to achieve coverage has been proposed: the path-prefix strategy introduced by Prather and Myers [12] that ensures branch coverage modulo (relatively) infeasible paths. The strategy has been adopted by Chang et al. [2] in their heuristic approach for test data generation. More research in path selection is desirable.

4. Problems of Test Data Generation

In this section problems encountered in test data generation are explored. Due to the complexity of the generation problem a great deal of the work has been based on toy programs, i.e. programs that either are very short in length, low in complexity, or lack the use of many standard language features such as abstract data types and pointers. Hence, not resembling anything that is developed for instance in industry.

4.1. Arrays and Pointers

Arrays and pointers are actually similar constructs and suffer from the same kind of problems, though some are more evident for one or the other. In symbolic execution arrays and pointers complicate the substitution, since values of variables are not known. Consider a condition statement using an array element by indexing with a variable:

```
input(i,j);
a[j] = 2;
a[i] = 0;
a[j] = a[j] + 1;
if (a[j] = 3) ...
```

If `i` is equal to `j` then `a[j]` in the if statement is `1` otherwise it is `3`. This is similar as saying `if (something = 3) ...`. Ramamoorthy et al. [13] propose an approach to solve this problem by creating a new instance of the assigned array whenever there is an unambiguity. Whenever such unambiguity is resolved so are the array instances as well. Of course this technique suffers a large performance penalty. In the case of actual execution this is not an issue, since values are known at runtime.

Not only the indexing problem has to be regarded, but also the *shape problem* has to be addressed. The shape problem is closely related to the loop problem in section 4.3. Think of a program that takes a complex dynamic data type such as a heap as input and performs some action upon it. In order to generate this structure as input, the generator must not only figure out internal shape of the structure (e.g. how heap nodes are connected), but also how large structure to generate (e.g. the number of nodes in the heap). Up to now only one attempt in solving the problem of generating dynamic data structures has been encountered [8]. The method was based on actual execution. How well this solution works is not clear.

4.2. Objects

Generating objects is by definition at least as hard as the pointer problem, since they often are dynamically allocated. To this the concepts of abstract classes, inheritance and polymorphism are added and thus makes it impossible at compile time to determine what code that is to be called. This means that any solutions to this problem have to be dynamic. Up to my knowledge there are no papers concerning this problem.

4.3. Loops

Loops depending on input variables, i.e. not having a constant number of iterations, becomes a trouble zone. Actually, as long as the given path to generate is specific loops in themselves cause no problem, since the exact amount of iterations then can be derived from the path. The problem is merely reduced to the problem of tuning the loop variables. But if the loop happens to lie in the unspecific part of a path it turns out to be a lot more difficult.

In the case of symbolic evaluation a closed form of the loop must be derived. This is generally not a simple task. Instead, Ramamoorthy et al. [13] suggest that the loop is executed K times, where K is chosen by the user or by the test data generator. The same is valid for actual execution.

4.4. Modules

Generally a program is divided into functions and modules. Considering symbolic execution, in the case of generating test data for a function containing other (non-recursive) function calls, Ramamoorthy et al. [13] have proposed two solutions: either the brute force solution by inlining the called functions into the target, or by analyzing the called functions first and generate path predicates for those functions.

But often source code of a function or a module is not accessible, e.g. precompiled libraries, and therefore a complete static analysis of the called functions is not possible. In actual execution, however, source code is not needed in the same extent.

4.5. Infeasible Paths

Generating test data in order to traverse a path involves solving a system of equations. If the system has no solution we can conclude that the path given is indeed infeasible. The problem is that solving an arbitrary system of equations is undecidable. If the system is linear we can by Gaussian elimination conclude whether that path is feasible [7]. For non-linear systems it becomes more inconvenient. All methods studied have set a highest number of iterations to do before abandoning the path as infeasible in order not to end up in an infinite loop.

4.6. Constraint Satisfaction

All encountered methods (except for random testing) have to satisfy some constraint, i.e. solve a path predicate or branch predicate. Most of the encountered methods use poor constraint satisfaction techniques, due to the fact that this is a difficult problem. Because of function calls all constraints cannot be solved in symbolic execution. The dynamic approaches do not suffer from function calls to the same extent, but there will still be constraints to satisfy.

Encountered search methods for solving constraints are among others alternating variable, simulated annealing, genetic algorithms, iterative relaxation and different heuristics [5, 14, 10, 7, 3].

4.7. Oracle

One way to drastically reduce the effort of testing is to have an oracle that would check if the test case failed or not. Having an oracle is especially important in automatic generation, since many inconceivable tests can be produced. Unfortunately, the only way of achieving an oracle is to supply extra information with the source code, e.g. a (requirement or design) specification, adding assertions or some other form of logic description of the program.

5. Conclusions

There have been several attempts in automation of test data generation. Attempts to simplify the process of constraint satisfaction have been made through introducing rule-based test data generation [3], and through the removal of path constraints in favor of goals [5].

The most promising search methods seems to be simulated annealing and genetic algorithms for their data type independence and iterative relaxation for its predictability. This is an area where there is much more to investigate, particularly in the object-oriented field.

A typical characteristic of the generators are that they handle only booleans, integers, reals and arrays (to some extent). However, there has been one attempt by Korel [8] in using pointers. The AI algorithms (e.g. genetic algorithms) tend to be better in dealing with more complex structures.

The answer whether to use symbolic execution or actual execution is to combine both of them. For instance, my intuitive opinion is that the shape problem is best solved using both static and dynamic analysis, and maybe some extensions to the data structure declaration, like introducing assertions. In this way an analyzer gets help in deriving a shape of the dynamic structure.

I have identified the following topics as interesting and challenging for further research:

- Constraint-satisfaction techniques
- Object-oriented programs
- Pointers and shapes
- Assertions
- Modules
- Path selection
- Data and control dependency
- Oracle problem

References

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.

[2] K. Chang, W. Carlisle, J. Cross II, and D. Brown. A heuristic approach for test case generation. In *Proceedings of the 1991 ACM Computer Science Conference*, pages 174–180. ACM, 1991.

[3] W. H. Deason, D. Brown, K. Chang, and J. H. Cross II. A rule-based software test data generator. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):108–117, March 1991.

[4] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[5] R. Ferguson and B. Korel. The chaining approach for software test data generation. *IEEE Transactions on Software Engineering*, 5(1):63–86, January 1996.

[6] R. Ferguson and B. Korel. Generating test data for distributed software using the chaining approach. *Information and Software Technology*, 38(1):343–353, January 1996.

[7] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings of the ACM SIGSOFT sixth international symposium on Foundations of software engineering*, pages 231–244, November 1998.

[8] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[9] B. Korel and A. M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th International Conferance on Software Engineering (ICSE)*, pages 71–80. IEEE, 1996.

[10] C. Michael and G. McGraw. Automated software test data generation for complex programs. In *13th IEEE International Conferance on Automated Software Engineering*, pages 136–146, October 1998.

[11] J. Offutt and J. Hayes. A semantic model of program faults. In *International Symposium on Software Testing and Analysis (ISSTA 96)*, pages 195–200. ACM Press, 1996.

[12] R. E. Prather and J. P. Myers, Jr. The path prefix software testing strategy. *IEEE Transactions on Software Engineering*, SE-13(7):761–765, July 1987.

[13] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, December 1976.

[14] N. Tracey, J. Clark, and K. Mander. Automated program flaw finding using simulated annealing. In *Proceedings of ACM SIGSOFT international symposium on Software testing and analysis*, volume 23, pages 73–81, March 1998.