Reviewing Automatic Test Data Generation

Dr. Mukesh Kumar1, Jyoti Chaudhary2
Associate Professor1, Assistant Professor & Research Scholar2
Department of Computer Engineering
TIT&S, Bhiwani, India1,2
UIET, MDU, Rohtak, India2

Abstract:
Automated Test Data Generation (ATDG) is an activity that in the course of software testing automatically generates test data for the software under test (SUT). It usually makes the testing more efficient and cost effective. Test Data Generation (TDG) is crucial for software testing because test data is one of the key factors for determining the quality of any software test during its execution. The multi-phased activity of ATDG involves various techniques for each of its phases. Several ATDG techniques are available, but emerging trends in computing have raised the necessity to summarize and assess the current status of this area particularly for practitioners, future researchers and students.

Keywords: Software Testing, SUT, Test Data

I. INTRODUCTION

Automated test data generation is one of the core factors that contribute towards automated testing. It is used for automatically generating test data for a SUT during software testing. Test data generation is a tedious, expensive and error prone process, if it is done manually. Automation in the test data generation process could curtail testing expenses and at the same time, increase the reliability of testing as a whole. For these reasons automated test data generation has remained a topic of interest for the past four decades. In [15]’s opinion ATDG is one of the most crucial activities of automated testing because test data is one of those key factors that determine the quality of a test during its execution. ATDG itself is a multi-phased process, which involves different techniques for every phase. For example program analysis, path generation and test data generation are the three basic phases of ATDG process that have been mentioned by [15] while explaining the architecture of a test data generator system. Similarly random, path-oriented and goal-oriented are some of the test data generation approaches that can generally be described as data generation techniques [17]. 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. Although many techniques for automatic test data generation have been developed [Coward 1988; Gallagher and Narasimhan 1997; Gupta et al. 1998; Sthamer 1995; Korel 1990b; Lapierre et al. 1999a; Meudec 2001; Michael et al. 2001], there is no overall evaluation and comparison of these techniques. Evaluation and comparison of existing techniques are useful for choosing appropriate approaches for particular applications.

The organization of this paper is alienated as follows:

Section II describes the basic concept and notion, section III discusses the test data generator system with focus on the generator and the path selector, section IV describes various techniques for test data generation and section V depicts the conclusion.

II. BASIC CONCEPT

According to Pargas et al [13] the intelligent test-data generation approach often relies on sophisticated analysis of the code to guide the search for new test data. However in the author’s opinion this approach can be extended up to the intelligent analysis of program specification as well. 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 path can for instance mean paths leading to 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.

Figure 1. Architecture of a test data generator system.
Generally software is tested for its structure or functions which are supposed to be performed by its components. Testing in the prior case is commonly known as structural or white box testing, while in the later case it is called functional or black box testing [10]. For either of the cases test data is required to “traverse” through the SUT. The outcome of this traversal determines correctness, performance, or in general, the quality of the SUT. In some cases the test data is already available or given but in most cases it is required to be generated. Test data generation is a complex activity because it involves so many steps and each step has several related issues. The architecture of a test data generator can better elaborate this notion of complexity. Figure shows the architecture of an automated test data generator using the path-oriented approach in simple and easily understandable format. Important to note here is that test data generation is not a single step process rather it involves several steps. Adequate test data can never be created if performance at any of the steps is poor or the execution is incorrectly performed.

III. 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 the focus is on selector and the generator. Therefore let us assume that the analyzer exists and works properly. At this point let us define the problem of automatic test data generation as follows: given a program P and a (unspecific) path p, generate input x \( \in S \), so that 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 flow graphs 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.

IV. VARIOUS TECHNIQUES

The most appropriate classification in terms of test data generation is: random, path-oriented and goal-oriented, although the problem of path selection is not considered separately.

A. 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 accomplishes high coverage. A semantically small fault is such a fault that is only revealed by a small percentage of the program input. This approach is quick and simple but might be a poor choice with complex programs and with complex adequacy criteria. A semantically small fault is a fault that is revealed only by a small percentage of the program input. However, random test data generation is used frequently as a benchmark because it is easy to implement and commonly reported in the literature.

Consider the following piece of code:

```c
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.

B. Goal-Oriented Test Data Generation

The goal-oriented approach is much stronger than random generation, in the sense of providing guidance towards a certain set of paths. Instead of letting the generator generate input that traverses the entry to the exit of a program, it generates input that traverses a given unspecific path p. Because of this, it is sufficient for the generator to find input for any path p \( \in \mathbb{N} \). 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 characteristic 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:

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

we can 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 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. In the goal-oriented approach test-data is selected from the available pool of candidate test data to execute the selected goal, such as a statement, irrespective of the path taken [13]. This approach involves two basic steps: to identify a set of statements (respective branches) the covering of which implies covering the criterion; to generate input test data that execute every selected statement (respective branch).

C. 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. In path-oriented test data generation the typical approach is generation of a control-flow graph. In this approach, at first a graph is generated first and subsequently, by using the graph a particular path is selected. With the help of a technique such as symbolic evaluation (in the static case otherwise it is called function minimization) test data is generated for that path in the end [13]. In symbolic execution variables are used instead of actual values while traversing the path. 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. The path-oriented approach might face the problems when generating paths/graphs, traversing test data through branches and predicates (infeasible path problem).

V. CONCLUSION
Although many techniques for automatic test data generation have been developed but there is no overall evaluation and comparison of these techniques. Evaluation and comparison of existing techniques are useful for choosing appropriate approaches for particular applications. For example, if the high quality of software is essential, a technique with the highest test coverage may be required regardless of cost. On the other hand, if execution time for the test data generation is the most important factor, a fast technique may be desirable, even though some tests that can expose faults may be omitted from the test suite. Evaluation and comparison of existing approaches also provide insights into the strengths and weaknesses of current methods. Random approach is quick and simple but might be a poor choice with complex programs and with complex adequacy criteria. The probability of selecting an adequate input by chance could be low in this case. The goal-oriented approach is much stronger than random generation, in the sense of providing guidance towards a certain set of paths. The path-oriented approach might face the problems when generating paths/graphs, traversing test data through branches and predicates (infeasible path problem), and while complexity of data types. 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. The intelligent test-data generation approach often relies on sophisticated analysis of the code to guide the search for new test data. However in the author’s opinion this approach can be extended up to the intelligent analysis of program specification as well.

VI. REFERENCES

