International Journal of Software Engineering.
ISSN 0974-3162 Volume 8, Number 1 (2017), pp. 13-29
© International Research Publication House
http://www.irphouse.com

Automated Test Data Generation

Zarah Bello-Yashe

Department of Mathematical Sciences, Kaduna State University, Nigeria.

Abstract

Software testing is a labor intensive and expensive task that generally accounts for a significant amount of software development cost. To reduce the high cost of manual testing and increase the reliability of the process, practioners have tried to automate it. This paper examines both the theoretical and practical role of testing in software development, then goes on to survey the purposes and limitations of testing as presently conceived. Various methods of automating test-data are presented, then the focus is reduced to program-based test data generation: using structural testing approaches such as data flow testing and control flow testing to demonstrate a systematic procedure for selecting valid and reliable test data. Various attempts have been made to automate the generation of test-data from an actual program but there are still unresolved problems to date. Despite flaws in the current methods, it is a step forward from manual testing and practioners endeavor to use these automated test tools in conjunction with manual ad-hoc testing to reduce cost and increase the reliability of software systems, while researchers endeavor to resolve the existing issues. Automated testing is an important topic in software engineering and the amount of research in the field suggests, both practioners and researchers believe that – **Automation is the Key!**

Keywords: Software testing, test-data, automated testing, test-data generator

1 INTRODUCTION

1.1 Software Testing

Software testing is an activity that aims to identify correctness, completeness, security and quality of computer software [1] by measurement and evaluation of the software's attributes and capabilities. It is a rapidly maturing area within the software engineering process, and over the years has been receiving increasing notice in the software engineering industry.

Testing is an expensive and labor-intensive process, which accounts for 50% of total development cost. Failure to detect errors can result in significant financial loss or even disaster in the case of safety critical systems. Complete testing is impossible due to practical limitations such as time, exhaustion, and the huge input spaces involved. So generally, a set of coverage criterion are set and test coverage is stated in terms of the chosen criterion.

There is almost always a trade-off between quality and cost. So typically, testers seek techniques that will help achieve sufficient rigor at an acceptable cost. A narrower view considers testing as a way to affirm the quality of software systems by systematically exercising the software in a controlled test environment using the most effective and efficient methods available.

This can be done using a selection of techniques, including both static and dynamic analysis. Static analysis refers to methods that are used to determine or estimate program quality without reference to actual executions. It is important to appreciate that although static and dynamic analysis is sometimes inseparable in practice, they can be discussed separately [4]. Dynamic analysis deals with specific methods of ascertaining or approximating software correctness through actual program executions with real data, under real conditions, and it is dynamic analysis that this paper is concerned with.

Test cases are specifications of inputs to test the expected output from the system and they include a statement of what is being tested. Test data are the inputs that have been devised to test the program. Automatic test case generation is impossible because output of the tests can only be predicted by people who understand what the program should do. Alternatively, test data generation can be automated. Several attempts have already been made over the years -This is the topic of my research: **Program based test data generation:** where generation starts from the actual program.

1.2 Introduction to the problem

An error is a human action that produces incorrect results; a fault is a manifestation of an error in software. Faults may result in failure, which is the deviation of the software from its expected delivery or service. Problems in software can arise from any of these ways, and faults that go undetected at early stages of development can easily multiply to cause a number of failures at a later stage. Failure can cost nothing or it can cost a lot. Even tiny faults can result in catastrophic failures. Software faults in safety critical systems can cost lives not just money, in the past, erroneous bank overdraft letters have led to suicide. This is why software testing is so important, it identifies faults, whose removal increases the software quality by increasing its potential reliability.

This is the underlying motivation of program testing; to affirm computer software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems in order to reduce risk, increase quality and gain confidence within resource constraints. Although testing cannot guarantee that software is free of errors, it helps to minimize them and also build confidence in the software by convincing both developers and users that the software is stable enough for operational use.

The whole testing process is an expensive and laborious process, which typically accounts for 50% of total development cost [2]. It is generally accepted that the process places great strain on resources but if the process were to be automated, it would greatly reduce the cost of development and even increase the reliability of testing. For research purposes, this paper will focus on automating the generation of test-data, to facilitate one of the most tedious tasks in software development. There have been many attempts at automating the generation of test-data, these can be categorized as: specification-based, architecture-based and program-based. Each will be discussed, giving particular attention to program-based test-data generation.

2 BACKGROUND

2.1 Role of Testing

Contemporary thinking within the software engineering research and development community considers the software life cycle as composed of five distinct stages: requirements specification, system design, implementation, quality assurance and maintenance. Validation and verification takes place at each stage during development in order to identify faults as early as possible. The V-model shown in figure 2-1 describes how these testing activities are integrated with other tasks in the project lifecycle.

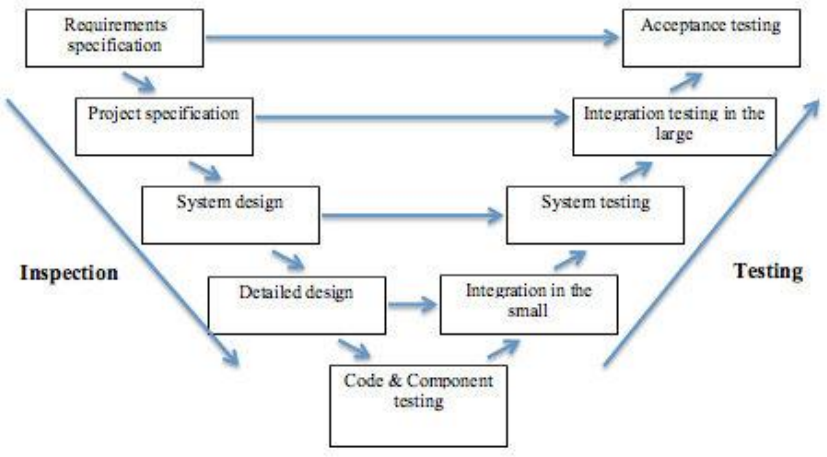


Figure 2-1: The V-Model

The fundamental test process comprises of planning, specification, execution, recording and checking completion as shown in figure 2-2.

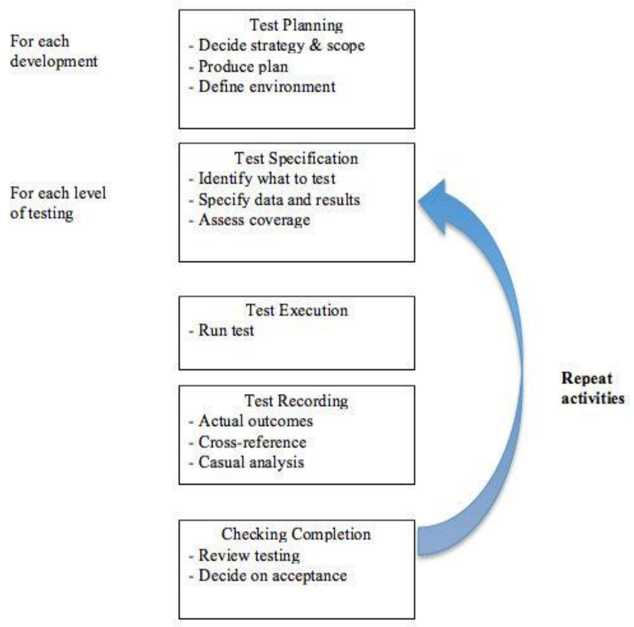


Figure 2-1: Test processes

2.2 Types of Testing

Testing all possible input values would clearly provide the most complete results in terms of the program's behavior, but in most cases the input domain would be far too large for exhaustive testing to be practical. Usually, relatively small subsets of inputs are selected which in some cases are representative of the whole input domain to test the code. Ideally, the aim is to choose a set of test-data that will detect all the errors present, but so far it has generally been impossible to find such an ideal set of test-data [7]. There are a number of techniques used for code analysis and selection of appropriate test sets. Given a set of inputs to a program, the resulting output can be classified as either correct or incorrect with reference to some specification; the manner in which the test sets are determined varies. The most common of them are discussed below.

2.2.1 Black-Box Testing

In some cases, the internal details of code are not used and so the test sets are created wholly from the specification. In such cases the software being tested is seen as a 'black box', and so the derivation of test sets is known as black-box testing [6].

2.2.2 White-Box Testing

Alternatively, it is possible to analyze the internal code to see how test sets actually execute certain elements of the implementation. For a set of inputs, a program must follow a sequence of small execution steps as their effects allows a more rigorous analysis of the test results. This method of testing is traditionally known as white-box testing, or more recently termed clear-box testing [6].

Both methods are equally effective depending on the circumstances and tools available. In some cases, where new software has been developed, it may be difficult to instrument the code to give the tester the information they need regarding the smaller state transitions during execution. In other cases when there is legacy code, the actual source code may not be available, in such cases black-box testing is the only option.

2.3 Coverage Criteria

If a test must satisfy some particular property - coverage criteria defines what constitutes adequate coverage in relation to set of tests each test contributing to satisfying the collective requirement - then test-data generation reduces to 'find test-data that satisfies the property'. The criteria can be used in many ways, firstly as a stopping rule indicating whether or not enough testing has been done. Secondly it can be used to measure the quality of testing by associating a degree of adequacy with each test set. The notion of coverage criteria is essential for any testing method, as well as directing the selection of test-data; it also decides the sufficiency of a given test set [9].

3 PROGRAM-BASED TEST-DATA GENERATION

The process of test-data generation is laborious and time consuming. In order to reduce this high cost and at the same time increase the reliability of the process, researchers and practitioners have tried to automate it. Through the years a number of different methods for generating test data have been presented and categorized according to the source artifact for the generation: specification-based, architecture-based and program-based. Although a reasonable amount of research has already been carried out on each, program-based testing has elicited the greatest amount of attention, so I will focus on that.

Program-based test data generation is concerned with generating test-data based from the program. According to the underlying approach, program-based testing can be divided into three further subsections: fault-based testing, error-based testing, and structural testing.

3.1 Mutation Testing: Fault-Based Testing

Mutation testing is a fault-based technique that focuses on measuring the quality of a test set according its ability to detect specific faults. It is a fault-based technique where a large number of simple faults, such as simply altered operators, constant values and variables, are introduced into the program under test, one at a time. The resulting programs are called mutants. Then the goal is to generate test cases that can distinguish each mutant from the original program by its outputs. If the mutant can be distinguished by at least one of the test cases in the test set then we say the mutant is killed, otherwise we say that the mutant is alive.

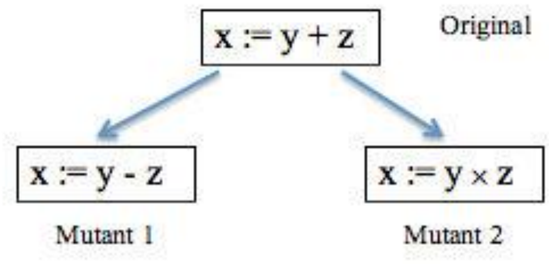


Figure 3-1: Mutant construction

Figure 3-1 demonstrates an example of mutant construction, if the input variables are y and z, and the output variable is x, then taking the case (y=0, x=0) cannot kill either of the mutants because the output x will be the same for the original and all mutant programs. Sometimes the mutant cannot be killed due to equivalence of the mutant and the original program, in such cases the adequacy of the test can be assessed using the following equation:

$$\text{AdequacyScore} = \quad D / M - E$$

Where D is the number of mutants that have been killed, M is the total number of mutants, and E is the number of equivalent mutants.

Mutation testing is based on two assumptions: the competent programmer hypotheses and the coupling effect. The competent programmer hypothesis assumes that the programmer creates programs that are close to being correct. This is why we create the mutant by deviating the original program slightly, rather than considerably. The coupling effect assumes that a set of test-data that can uncover all simple faults in a program is also capable of detecting more complex faults. This assumption assures that the ability of the test set we generate is not limited to recognizing only simple faults.

The benefit of mutation testing is that it allows a great degree of automation. When mutant operators are defined, the generation of mutants, the execution of mutants and the original program, and the comparison of the results can all be automated, thus the mutation adequacy score can also be calculated automatically. The mutation operators must be carefully defined because it is very easy, for example to make a mutant of a program that does not compile (e.g. because the mutant is not type correct). Although such mutants can easily be detected by the compiler, it would be a very inefficient test case and it is better to create only semantically correct mutants.

One of the disadvantages of mutation testing is the huge computation cost required. It is estimated that the number of mutants that can be generated is on the order of N^2 where N is the number of lines of code in the program. Additionally once a mutation adequate input set has been generated, the outputs need to be calculated, and this often has to be done manually. Basically, test cases must be created, not just inputs. Another problem with mutation testing is the significant human cost of examining equivalent mutants. Mutation testing is largely an academic preserve used to give an independent measure of the effectiveness of other testing techniques.

3.2 Domain Testing: Error-Based Testing

Error-based testing is concerned with the use of test cases to check programs on error-prone points. Boundaries are often what give rise to errors. These errors can be divided into to areas: computational errors and predicate errors. An incorrect predicate can easily go undetected, a wrong computational assignment that affects a predicate can also easily escape detection, thus to thoroughly test on and around the boundaries of domain partitions can be very profitable [11]. To get the boundaries, the input domain is partitioned into sub-domains, where the input-output behavior of the software for each sub-domain should be equivalent, meaning the program takes the same path.

3.3 Structural Testing

Structural testing specifies testing requirements in terms of the coverage of a particular set of elements in the structure of the program or the specification. It can be either control-flow based or data-flow based.

3.3.1 Control-Flow Based Testing

Control flow testing is based on the knowledge of the control structure of the program. A variety of control-flow testing criterions have been defined as listed and explained in section 4.1. Control-flow based testing is not guaranteed to find all errors, even if all feasible paths through the code have been executed by the test set coverage criteria is discussed in further the next section but generally, control-flow criteria alone are not sufficient and further criteria should be considered.

3.3.2 Data-Flow Based Testing

In data-flow testing the criteria are constructed so that critical associations between definitions of a variable and its uses are examined during program testing. The idea is that if the result of some computation has never been used, one has no reason to believe that the computation was correct.

The criteria are based on the following terminology, where each variable occurrence is classified as being definitional, computation-use, or predicate-use occurrence, referred to as *def, c-use* and *p-use,* respectively.

- *def:* a variable that is declared or assigned to or contained in an input statement.
- *c-use:* a variable that forms part of the right hand side of an assignment statement or is used as an index of an array or contained in an output statement.
- *p-use:* occurs when a variable forms part of a predicate in a conditional-branch statement.

Below are the most popular path selection criteria based on data-flow:

- *All-defs:* requires that a definition clear path from each variable definition to a use of the variable be executed at least once.
- *All-uses:* requires that a definition clear path from each variable definition to every reachable use of that variable be exercised at least once.
- *All-du-paths:* requires that every du-path from each variable definition to every use of the definition be executed at least once.

Data-flow testing is clearly based on path selection. The criteria indicated demand that certain sub-path in a program be executed. In this way, it is similar to ranch coverage, in control-flow criteria. In much the same way, there may well be errors that test based on data flow criteria will miss.

4 THE AUTOMATIC TEST-DATA GENERATION SYSTEM

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. The test data generation system consists of these parts: a program analyzer a path selector, and a test data generator. Below an explanation of the path selector and data generator, the program analyzer is not further investigated in this paper.

4.1 Path Selector

Effectiveness of the whole system is highly dependent on how paths are selected. In path selection we define the automatic test data generation problem as a given program P, find the least set of paths in P such that it meets a specified coverage criteria. This means it is vital to find good test data. Baring in mind that no strategy can guarantee adequate testing except exhaustive testing which is infeasible. But by carefully selecting paths, we can come up with a strategy that covers the whole program in terms of the chosen coverage criterion. The stronger criterions, the more paths have to be selected. A test strategy can be based on coverage of one or more of the following:

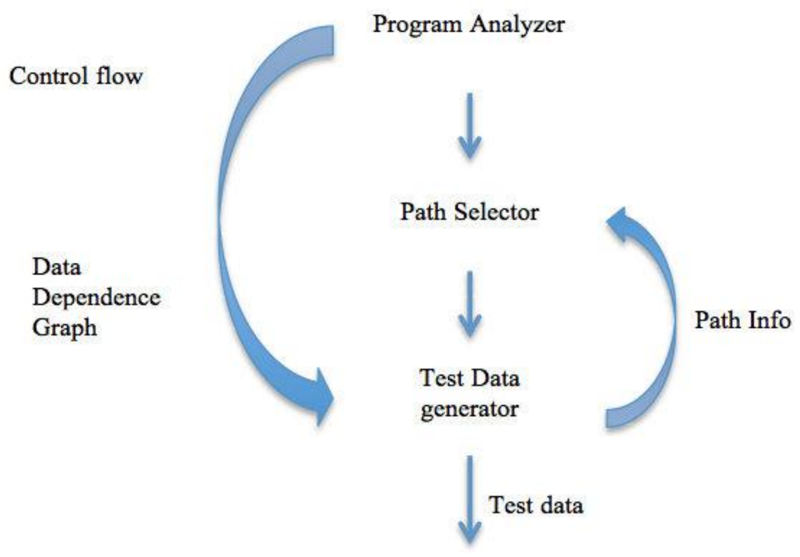


Figure 4-1: Architecture of a test data generator

4.1.1 Statement Coverage

Statement coverage executes all statements (nodes) in the control flow graph. Although it may not be obvious, it is the weakest of all the criteria. Figure **4-2** helps us to illustrate this: statement coverage can be achieved by simply selecting the path <1**,2,3,4>,** but this means that faults depending on non-execution of node **2** are left uncovered, [10].

4.1.2 Branch Coverage

Branch coverage executes all branches (edges) in the control flow graph. In other terms the predicate of an if-statement must be evaluated to both *true* and *false.* We can achieve branch coverage by selecting the paths <1**,3,4>** and **<1,2,3,4>.** This criterion allows us to execute the tests that would have been left uncovered in statement coverage. Hence confirming that branch coverage is strictly stronger than statement coverage because if you have covered all the branches (edges) then you have definitely covered all the statements (nodes). On the other hand if you cover all statements (nodes) it doesn't necessarily mean that you have covered all branches (edges), [10].

4.1.3 Condition Coverage

Condition Coverage is where each clause within each condition of the control flow graph must be executed to both *true* and *false,* sometime during the execution, [10, 6].

4.1.4 Multiple-Condition Coverage

In this strategy each combination of truth-values of each clause of each condition must be executed during execution, [10, 6].

4.1.5 Path Coverage

A simple path is one in which no edge is traversed more than once. But actual path coverage refers to traversing all paths in the control flow graph. This is infeasible because an infinite set of test will be generated, [10, 6].

The stronger criteria of condition, multiple-condition and path coverage are often infeasible to achieve for programs of more than moderate complexity, and for this reason branch coverage has been recognized as the basic measure for testing [10].

4.2 Test Data Generator

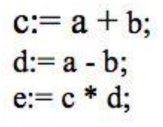
Here we will define the problem of automatic test data generation as follows: given a program P and a (unspecific) path U, generate input x E S, so that x traverses U. Assuming we have **a** program **analyzer** and a path selector working adequately to provide all information about the programs data-dependence graphs and control flow graphs along with the paths for which the test data generator will derive input values for. Depending on the type of generator system, paths could either be specific or unspecific.

The aim is to generate input values that will traverse the paths received from the selector. This is done in two steps, firstly to find the path predicate for the path. Secondly to solve the path predicate in terms of input variables [12]. The solution will be a system of inequalities describing how input data should be formed in order to traverse the path. Having such a system, various search methods can be applied to it to come up with a solution. Examples of search methods are simulated-annealing, alternating variables, and different heuristics based on equation rewriting systems [15,16].

Due to the complexity of the derived equation systems some techniques solve one branch predicate at a time. This leads to loss of performance because it is necessary to check that violations of previously solved predicates do not occur. There are three main approaches when constructing a test data generator: randomly generate test data, generate the test data for a specific path, or generate test data for an unspecific path. These fall into three classes of random, goal-oriented, and path-oriented test data generation, each of which can be implemented statically or dynamically.

4.3.1 Static and Dynamic Test Data Generation

To come up with a transformed system of equations we can use symbolic execution or actual execution, so generation occurs either statically or dynamically. Executing a program symbolically means that instead of using actual values, variable substitution is used. This data is to end up with an expression in terms of input variables. For example, take **a** and **b** as input variables:



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

The e in this code will contain **a*a- b*b.** It has been realized that this technique requires a lot of computer resources to carry out these computations, since all expression have to be resolved and transformed. Symbolic execution also implies that a symbolic evaluator for the particular language is built, this clearly involves a large amount of work. But there are some benefits to symbolic execution as well. For instance, it requires no validation checks for branch predicates since everything can be solved at once.

The opposite of symbolic execution is actual execution. Instead of using variable substitution, the program is run with some selected input. Consequently, the values of variables are known at any time of the execution. Monitoring the program flow allows the system to determine if the intended path was taken. If not, then it backtracks to the node where the flow took the wrong direction, and using different search methods the flow can be altered by manipulating the input in a way that the intended branch is taken. This is a very expensive technique, and can require many iterations before a suitable input is found. When the flow is changed at a particular node, the flow at an earlier stage may accidentally change. Actual execution also suffers from the speed of execution for the program to analyzer. There are benefits from both symbolic and actual execution, but they are most effective when used together [12].

4.3.2 Random Test-data Generation

Random testing is the simplest of the generation techniques. It could actually be used to generate input values for any type of program since a data type such as an 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.

Contrarily, random testing mostly does not perform well in terms of coverage because it merely relies on probability so the chances of finding semantically small faults are quite low. A semantically small fault is one where the fault is only revealed by a small percentage of the program input. Consider the following piece of code:

```
void show (int a, int b) {
        If (a=b) then
                write (sl);
        else
                write (s2);
        }
```

The probability of exercising s1 statement is 1/**n,** where n is the maximum integer, since in order to execute this statement variables **a** and **b** must be equal. Knowing this, it is easy to imagine more complex structures will produce even worse probabilities.

4.3.3 Goal-Oriented Test-data Generation

The goal-oriented approach is much stronger than the random generation approach in terms of providing guidance towards a certain set of paths. Instead of allowing the generator to generate input that traverses from the entry to the exit of a program, it generates input that traverses a given unspecific path u. So it is then sufficient for the generator to find input for any path p that is a subset of u. This reduces the risk of encountering relatively infeasible paths and provides a way to direct the search for input values as well. Two main methods were found using this technique: the chaining approach and assertion-oriented approach, the latter being an interesting extension of the chaining approach. They have all been implemented in the TESTGEN system [13,17].

The chaining approach typically uses 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 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 test (int a) {

        int  b  =  (a+I)*(a-l);
        assert   (b   !=   0);
        write(l/b);
```

Before executing l/**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. The advantage with assertion-oriented testing is that the oracle (discussed in section 5) is given in the code. So in all 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 then expected value is provided within the assertion.

4.3.4 Path-Oriented Test-data Generation

Path-oriented generation is the 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. It is the same as the goal-oriented approach in this way, except that it uses specific paths. Successively this leads to a better prediction of coverage, but on the other hand it is harder to find test data. CASEGEN [11] and TESTGEN [13] are two systems that use this technique, but because they are solely based on the control flow graph they often lead to selection of infeasible paths.

DeMillio and Offutt [18] have proposed a constraint-based test data generation method that is focused on fault-based testing using mutants. However it is no clear how the paths are selected and since the technique is quite similar to assertion-oriented testing it could be classified as goal-oriented generation as well.

5 PROBLEMS OF TEST DATA GENERATION

Due to the complexity of the generation problem, a great deal of the work has been based on toy programs, these are 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 adequately representing the systems that are currently being developed in industry. As a result there are still a number of problems with automating the generation of test data that are yet to be solved.

5.1 Arrays and Pointers

Arrays and pointers are very similar constructs and so suffer from similar types of problems. During symbolic execution, arrays and pointers complicate substitution because the values of the variables are not known. An approach was proposed to solve this problem by creating a new instance of the assigned array whenever there is an unambiguity [11]. Whenever such instances are resolved, the array instance is also resolved. As expected, this technique has huge performance penalties for the program, although in the case if actual execution this is not an issue since the values are assigned at runtime.

Another issue that has to be addressed is the shape problem. For example, when using a program that takes a complex dynamic data type such as a heap as the input domain and performs some function on it. To generate this heap structure as input, the generator must figure out internal shape of the structure (e.g. How heap nodes are connected), and also how large the structure to generate (e.g. The number of nodes I the heap). To date, only one attempt has been made to solve the problem of generating dynamic data structures [13], the method was based on actual execution, but how well the method works is still unclear.

5.2 Objects

By definition, generating an object is similarly as hard as the array and pointer problems since they often are dynamically allocated. Considering the linked concepts of abstract classes, inheritance and polymorphism, it becomes impossible to determine which part of the code is to be called at compile time. This means any solutions to this problem have to be dynamic, and current research indicates that no papers have been written regarding this issue to date.

5.3 Loops

As long as the given path to generate is specific loops in themselves wont cause any problems since the exact amount of iterations can be derived from the path. The issue arises when you have Loops that depend on input variables i.e. don't have a constant number of iteration. This is merely a problem of tuning the loop variables; although it can be a much bigger problem for loops that lie in the unspecific part of a path. It was proposed that a maximum number of iterations K is set for each loop, where K is chosen by either the user or by the test data generator [11].

5.4 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 [12]. The problem is that solving an arbitrary system of equations is un-decidable. If the system is linear we can by Gaussian elimination conclude whether that path is feasible. For non-liner systems it becomes more inconvenient. Most methods set a highest no of iterations to execute before abandoning the path as infeasible to avoid falling into an infinite loop.

5.5 Constraint Satisfaction

With a few exceptions, all test data generation methods have had to satisfy some constraint. Most of which poor constraint satisfaction techniques are used, mainly because it is a difficult area. Because of function calls, all constraints cannot be solved in symbolic execution, but dynamic approaches do not suffer from function calls to the same extent, although there will still be constraints to satisfy. There have been some solution for solving constraints such as simulated annealing, alternating variables, genetic algorithms, and various others including different heuristics [12, 14, 15, 16, 17].

5.6 Oracle Problem

One positive way to reduce the effort of testing is to have an oracle that would check if the test case passed or failed. Using 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, like a requirement or design specification, adding assertions or some other form of logic description of the program.

6 DISCUSSIONS

In conclusion, this paper has provided an overview of the underlying principles concerning software testing and its essential role in program development: which lead to the need for automation. Presumably every piece of software undergoes some type of testing, yet my search has shown that the fundamental techniques of test-data selection have not previously been as carefully and critically analyzed as have some less widely used techniques such as proof of correctness. For this reason there are still a number of open problems with automated test data generation as discussed in section 5.

The weakness of testing lies in concluding that from the successful execution of selected test data, a program is correct for all data. Criteria for test data selection based solely on internal program structure are clearly too weak to insure confidence in the results of a successful test. Since such methods are too weak to necessarily reveal design errors or many types of construction errors. Reliability is the key to meaningful testing and lack of reliability is the reason for the current weakness of testing as a method for insuring software correctness. Assuming that software testers are competent in their work, then the reliability of manual testing can be judged by the tester to some extent, taking into consideration the fact that human beings are prone to making mistakes. In automated testing, it defeats the purpose if someone has to manually check that every set of test data is reliable. Further work is needed to show when a test is reliable.

A systematic approach to test data selection is clearly necessary to obtain effective test-data with reasonable effort. The problem is most systematic methods proposed to date have been based solely on knowledge of a program's internal structure, and yet such knowledge is insufficient by itself to yield adequately reliable test.

The most promising search methods seem to be simulated annealing and genetic algorithms for their data type independence and iterative relaxation for its predictability. The AI algorithms also seem to be better in terms of dealing with more complex structures. This is another area in which there is more to investigate, particularly with in the object-oriented field.

With respect to actual and symbolic execution, my conclusion is to combine both. Using the shape problem as an example, it would be best solved using both static and dynamic analysis and perhaps some extension to the data structure declaration, like introducing assertions. In this way an analyzer gets help in deriving the 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, Modules, Data and control dependency, Oracle problem, Assertions, Program Analyzer, and also maybe so further investigation into path-selection.

ACKNOWLEDGEMENTS

I would like to thank Professor Dov CeAbbey for making many helpful comments, which helped me in preparing this paper and Dr. Yohanna Tella for continuously supporting and encouraging me in my academic career.

REFERENCES

[1] Wiklepdeia definition, http://en.wikipedia.ora/wiki/Software_testing. Last Accessed July 2014.

[2] B. Beizer. Software Testing Techniques. Van Nostrand reinhold, 2nd Edition, 1990.

[3] Edsger Dijkotra's quote about software engineering.

[4] Edward Miller, William. E. Howden. Tutorial: Software Testing & Validation Techniques. 2nd Edition, 1981.

[5] Edward Miller. Introduction to Software Testing Technology, IEEE Computer Society Press, 1981.

[6] N. E. Fenton. Software Metrics: A Rigorous and Practical Approcah, 1997.

[7] J. B. Goodenough and S. L. Gerhart. Toward a Theory of Testing Data Selection, IEEE Transactions on Software Engineering, June 1975.

[8] Hong Zhu. Patrick A. V. Hall and John h. R. May, Software Unit test Coverage and Adequacy. ACM Computing Survey Vol. 29, No.4, December 1997.

[9] Mark Harman, CSMART Lecture Slides. Kings College London. October 2006.

[10] C. V. Ramamoorthy, S. F. Ho, and W. T. Chen. On the automated generation of program test data. IEEE Transactions on Software Engineering, December 1976.

[11] N. Gupta, A. P. Mathur, and M. L. Soffa. Automated test data generation using iterative relaxation method. In Proceedings of the ACM SIGNSOFT sixth international conference on Software Engineering, Agust 1990.

[12] B. Korel. Automated software test data generation. IEEE Transaction on Software Engineering, August 1990.

[13] 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, March 1991.

[14] R. Ferguson and B. Korel. The chaining approach for software test data generation. IEEE Transactions on Software Engineering, January 1996.

[15] 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 analsis, March 1998.

[16] R. Ferguson and B. Korel. The chanining approach for software test data generation. IEEE Transactions on Software Engineering, January 2006.

[17] H.Tahbildar and B. Kalita. International Journal of Software Engineering & Applications (IJSEA), Vol.1, No.4, October 2010.