IRA-International Journal of Technology & Engineering
ISSN 2455-4480; Vol.07, Issue 02 (2017)
Pg. no. 35-45
Institute of Research Advances
http://research-advances.org/index.php/IRAJTE

A Review of Test Data Generation and Adequacy Criteria

Sheetal Rajput
Assistant Professor, Management Department, DME, India.

Type of Reviewed: Peer Reviewed.
DOI: http://dx.doi.org/10.21013/jte.v7.n2.p4

How to cite this paper:
Rajput, S. (2017). A Review of Test Data Generation and Adequacy Criteria. *IRA-International Journal of Technology & Engineering* (ISSN 2455-4480), 7(2), 35-45. doi:http://dx.doi.org/10.21013/jte.v7.n2.p4

ABSTRACT

In the software testing a real way of measuring effective testing is that the system passes an adequate suite of test cases, and then it must be correct or dependable. But that's impossible because the adequacy of test suites is provably undecidable. So we'll have to settle adequacy Design rules to highlight inadequacy of test suites. Design rules do not guarantee good designs. Good design depends on talented, creative, disciplined designers and that can be the best test suite set to fulfill adequacy criteria.

Keywords: adequacy criteria, automatic test case generation, test data generation.

Abbreviations: GA- Genetic Algorithm, SUT- Software under Testing

I.INTRODUCTION

If we talk about testing then the role of test case designing is very important. A Test case is a set of inputs, execution conditions, and a fail/pass criterion. A requirement to be satisfied by one or more test cases.A test suite is created by such test cases. Test obligation is a partial test case specification, requiring some property deemed important to thorough testing.

If a test suite fails to satisfy some criterion, the obligation that has not been satisfied may provide some useful information about improving the test suite. If a test suite satisfies all the obligations by all the criteria, we do not know definitively that it is an effective test suite, but we have some evidence of its thoroughness.

II.TEST DATA GENERATION

Software testing accounts for 50% of the total cost of software development. 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 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.

Test data is basically the input needed for executing the test code, which has been written to test the 'software under test' (SUT) and Automated Test Data Generation (ATDG) Technique is a technique used for automatic test data generation for SUT.

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 for instance mean paths leading to high code coverage.

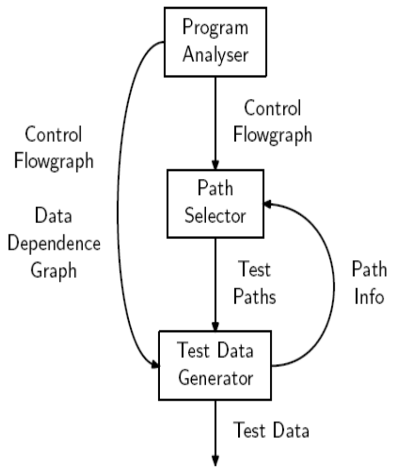


Figure 1: Architecture of a test data generator system.

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.

A control flow graph, or just flow graph when context is clear, is a graphical representation of a program. There exist many different definitions on control flow graphs throughout the literature. The definition used here has been inspired by Beizer as well as Korel. Let's consider a program to understand.

```
inttriType( 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)
7if(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.

Using the program in figure 2 we can prepare a flow graph as in figure 3.

A *test case* is a software testing activity, which consists of event, action, input, output, expected result and actual result. Clinically defined (IEEE 829-1998) a test case is an input and an expected result. These steps can be stored in a word processor document, spreadsheet, database or other common repository.

The term *test script* is the combination of a test case, test procedure and test data. Initially the term was derived from the byproduct of work created by automated regression test tools. Today, test scripts can be manual, automated or a combination of both. The most common term for a collection of test cases is a *test suite*. The test suite often also contains more detailed instructions or goals for each collection of test cases.

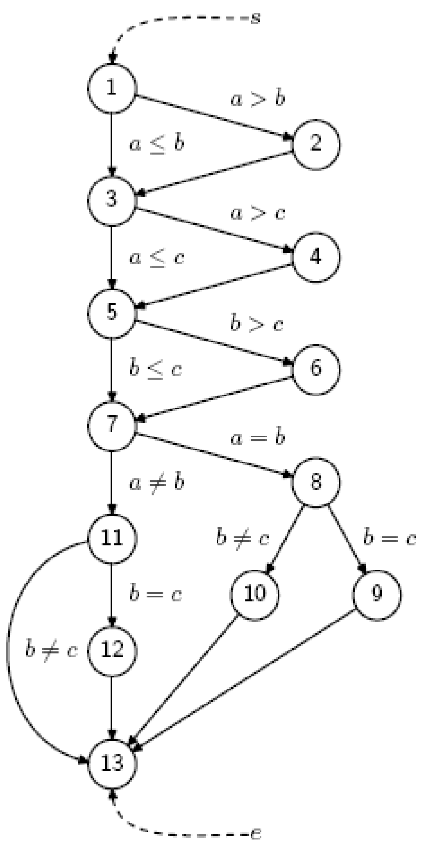


Figure 3: A flow graph of the program in figure.

The demand of automated test data generation for dynamically testing object-oriented software increases. The automatic creation of test data is still an open problem in object-oriented software testing, and many new techniques are being researched. For object orientated software, the automatic test data generation technique is not sufficient, because besides input data used for testing, it additionally has to produce the right sequences of method calls, and the right artifacts, to bring the object under test in the required state for testing.

Test cases for testing object-oriented software include test programs which create and manipulate objects in order to achieve a certain test goal. Strategies for the encoding of test cases to evolvable data structures as well as ideas about how the objective functions could allow for a sophisticated evaluation are proposed.

III.GENETIC ALGORITHM

Genetic algorithms have already been used to tackle typical testing problems with success, but the use of genetic programming applied to automatic test case generation is relatively new and promising.

Our work shows how genetic algorithms combined with different types of software analysis creates new unit tests with a higher amount of program coverage. Together with static analysis, the genetic algorithm is able to generate tests for more real world programs in a shorter amount of time. This presents an approach with which to apply evolutionary algorithms for the automatic generation of test data for the white-box testing of object-oriented software.

Genetic algorithms are basically search algorithms that are based on the ideas of genetics and evolution in which new and fitter set of string individuals are created by combining portions of fittest string individuals of an older set [Gol89].

Evolutionary computational methods have been of interest to computer scientists, mathematicians, biologists, psychologists and other investigators since the earliest days of computing research. Mitchell (1996) surveys some of the early work conducted on “evolution strategies” and other types of “evolutionary programming” efforts. Evolutionary algorithms for optimization and machine learning were among the first such topics researched. Genetic algorithms (GAs) with selection, crossover, and mutation were first introduced by Holland (1975) as an adaptive search technique that mimics the processes of evolution to solve optimization problems when traditional methods are deemed too costly in processing time.

A *genetic algorithm* is a heuristic that mimics the evolution of natural species in searching for the optimal solution to a problem. In the test-data generation application, the solution sought by the genetic algorithm is test data that causes execution of a given statement, branch, path, or definition–use pair in the program under test. The genetic algorithm conducts its search by constructing new test data from previously generated test data that are evaluated as good candidates. The algorithm evaluates the candidate test data, and hence guides the direction of the search, using the program’s control dependence graph.

A GA operates on strings of digits called chromosomes, each digit that makes up the chromosome is called a gene, and a collection of such chromosomes makes up a population. Each chromosome has a fitness value associated with it, and this fitness value determines the probability of survival of an individual chromosome in the next generation. After the next generation is created a percentage of the chromosomes are crossed and a small percentage of the genes are mutated.

A basic algorithm for a GA is as follows:

1. Generate a random population ofchromosomes.

2. Calculate the fitness of each member ofthe population.

3. While best chromosome (highest fitness)is below some threshold or the number ofgenerations is below some limit.

- Generate next population(reproduction) allowing greaterchance of more fit populationmembers to survive.
- Crossover a percentage ofpopulation members to create twonew members.
- Mutate a percentage of genes inthe population.
- Recalculate fitness of thepopulation.

4. Finish.

While the algorithm begins with a random population, it is the iterative process of reproduction, crossover, and mutation that gives the process a structure. A chromosome on its own may have a poor fit or low fitness, but it may contain some characteristics which when combined with another chromosome (crossover) or adjusted slightly (mutation) will create an offspring with a much higher fitness.

A genetic algorithm execution begins with a random initial population of candidate solutions {is} to an objective function f(s). Each candidate solution is generally a vector of parameters to the function f(s) and usually appears as an encoded binary string (or bit string) called a chromosome. An encoded parameter is referred to as a gene, where the parameter's values are the gene's alleles. The number of parameters and the size of the bit encoding of each parameter defines the length of the chromosome. In this part a chromosome represents an encoding of a test case.

After creating the initial population, each chromosome is evaluated and assigned a fitness value. Evaluation is based on a fitness function that is problem dependent. From this initial selection, the population of chromosomes iteratively evolves to one in which candidates satisfy some termination criteria or, as is also in our case, fail to make any progress. Each iteration step is also called a generation.

Each generation may be viewed as a two stage process [Whi94]. Beginning with the current population, selection is applied to create an intermediate population and then crossover and mutation are applied to create the next population. The most common selection scheme is the roulette-wheel selection in which each chromosome is allocated a wheel slot of size in proportion to its fitness. By repeatedly spinning the wheel, individual chromosomes are chosen using "stochastic sampling with replacement" to construct the intermediate population. Other selection schemes also exist [Gol89], for instance the elitist scheme in which the fittest chromosomes survive from one population to the other.

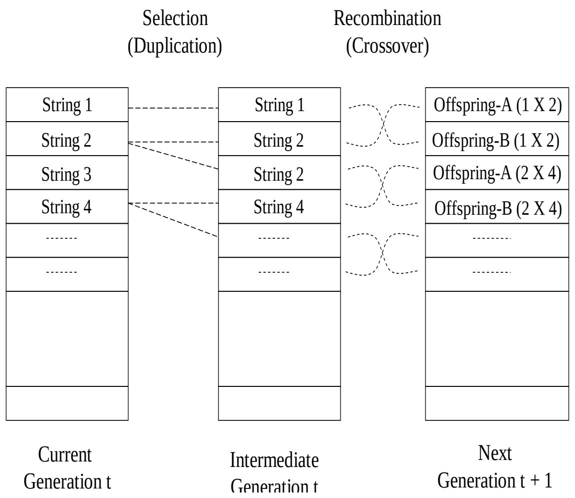
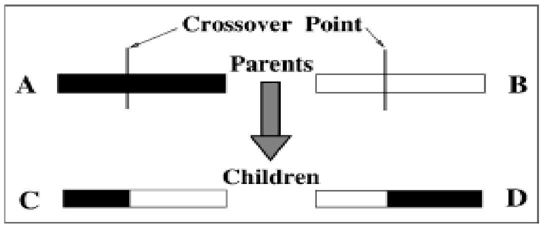


Figure 5: One generation is broken down into a selection phase and recombination phase.

This figure shows strings being assigned into adjacent slots during selection. In fact, they can be assigned slots randomly in order to shuffle the intermediate population. Mutation (not shown) can be applied after crossover.



After selection & crossover recombination is applied to randomly paired strings with a probability. Amongst the various crossover schemes. The one point, two point and the uniform crossover schemes are few popular schemes [Gol89, Whi94]. In the one point case a crossover point is identified in the chromosome bit string at random and the portions of chromosomes following the crossover point. In the paired crossover chromosomes, are interchanged, while in the two point case two crossover points are identified in the bit string and the middle portions are exchanged in the paired chromosomes. In the case of uniform crossover each bit in the string is exchanged according to a predefined crossover probability.

In addition to crossover, mutation is used to prevent permanent loss of any particular bit or allele. As an operator, mutation application also introduces genetic diversity. Mutation results in the flipping of bits in a chromosome according to a mutation probability which is generally kept very low.

The chromosome length, population size, and the various probability values in a GA application are referred to as the GA parameters. Selection, crossover, mutation are also referred to as the GA operators.

IV. ADEQUECY CRITERIAN

Adequacy criterion is a predicate that is true (satisfied) or false (not satisfied) of a ⟨program, test suite⟩ pair. It is the set of test obligations. A test suite satisfies an adequacy criterion if all the tests succeed (pass)every test obligation in the criterion is satisfied by at least one of the test cases in the test suite.

Adequacy criteria create test cases to cover faults that could possibly occur in the software, introduce mutations in the code. The statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was “pass”. Sometimes no test suite can satisfy a criterion for a given program.

```
if (a==0)
{
c=b/a;
throw new Logic Error (“a must be non zero here!”)
}
```

Simply put, code coverage is a way of ensuring that your tests are actually testing your code. When you run your tests you are presumably checking that you are getting the expected results. Code coverage will tell you how much of your code you exercised by running the test. There are a number of criteria that can be used to determine how well your tests exercise your code.

V. COVERAGE MATRICS

A number of different metrics are used determine how well exercised the code is ?The most simple is statement coverage, which simply tells you whether you exercised the statements in your code. When working with code coverage, we should keep in mind that “Testing never proves the absence of faults, it only shows their presence.”

Code Coverage is a way to try to cover more of the testing problem space so that we come closer to proving the absence of faults, or at least the absence of a certain class of faults. The code coverage is just one weapon in the software engineer's testing arsenal.

The Statement Coverage is the most basic form of code coverage. In software testing practice, testers are often required to generate test cases to execute every statement in the program at least once. A test case is an input on which the program under test is executed during testing. A test set is a set of test cases for testing a program. The requirement of executing all the statements in the program under test is an adequacy criterion. A test set that satisfies this requirement is considered to be adequate according to the statement coverage criterion. Sometimes the percentage of executed statements is calculated to indicate how adequately the testing has been performed. The percentage of the statements exercised by testing is a measurement of the adequacy.

A statement is covered if it is executed. A statement does not necessarily correspond to a line of code multiple statements on a single line.

Where there are sequences of statements without branches it is not necessary to count the execution of every statement, just one will suffice. But people often like the count of every line to be reported especially in summary statistics.

It can be quite difficult to achieve 100% statement coverage. There may be sections of code designed to deal with error conditions, or rarely occurring events such as a signal received during a certain section of code. There may also be code that should never be executed:

```
int a=10;
if (a<0)
print "This should never happen!";
```

Statement coverage, or something very similar, can also be called statement execution, line, block, basic block or segment coverage.

The goal of *Branch Coverage* is to ensure that whenever a program can jump, it jumps to all possible destinations. Similarly the branch coverage criterion requires that all control transfers in the program under test are exercised during testing. The percentage of the control transfers executed during testing is a measurement of test adequacy. The simplest example is a complete if statement:

```
if (x)
print "it will be execute";
else
print "it will not be execute";
```

Full coverage is achieved here only if x is true on one occasion and false on another.

Achieving *full branch coverage* will protect against errors in which some requirements are not met in a certain branch. For example:

```
if (x!=0)
{
print "x is nonzero";
z=y/x;
}
else
{
print "x is zero";
}
```

This code will fail if x is false. In such a simple example statement coverage is as powerful, but branch coverage should also allow for the case where the else part is missing, and in languages which support the construct, switch statements should be catered for:

```
if (x!=0)
{
print "x is nonzero";
z=y/x;
}
```

100% branch coverage implies 100% statement coverage. Branch coverage is also called decision, arc or all edges coverage.
There are classes of errors which branch coverage cannot detect, such as:

```
if (x)
{
print "it will be execute";
}
if(y)
{
print "it will also be execute";
}
```

100% branch coverage can be achieved by setting (x, y) to (1, 1) and then to (0, 0). But if we have (0, 1) then things go bang. The purpose of path coverage is to ensure that all paths through the program are taken.

Development efficiency = size of software/ Time required

Time = Person days, Person Hours, Size= loc

Old generation languages may reduce productivity as there are less chance of reusability. So for all above mentioned adequacy criteria, objective of software's has better chance of code coverage & will have better development efficiency due to reusability of code.

When a boolean expression is evaluated it can be useful to ensure that all the terms in the expression are exercised. For example:

```
a if x || y;
```

To achieve *full condition coverage*, this expression should be evaluated with x and y set to each of the four combinations of values they can take.

In many software programming languages, most boolean operators are short-circuiting operators. This means that the second term will not be evaluated if the value of the first term has already determined the value of the whole expression. For example, when using the || operator the second term is never evaluated if the first evaluates to true. This means that for full condition coverage there are only three combinations of values to cover instead of four.

Condition coverage gets complicated and difficult to achieve, as the expression gets complicated. For this reason there are a number of different ways of reporting condition coverage which try to ensure that the most important combinations are covered without worrying about less important combinations. Expressions which are not part of a branching construct should also be covered:

z = x || y;

Condition coverage is also known as expression, condition-decision and multiple decision coverage.

Mutation Adequacy is often aimed at detecting faults in software. A way to measure how well this objective has been achieved is to plant some artificial faults into the program and check if they are detected by the test. A program with a planted fault is called a mutant of the original program. If a mutant and the original program produce different outputs on at least one test case, the fault is detected. In this case, we say that the mutant is dead or killed by the test set. Otherwise, the mutant is still alive. The percentage of dead mutants compared to the mutants that are not equivalent to the original program is an adequacy measurement, called the mutation score or mutation adequacy [Budd et al. 1978; DeMillo et al. 1978; Hamlet 1977].

From Good enough and Gerhart's [1975, 1977] point of view, a software test adequacy criterion is a predicate that defines "what properties of a program must be exercised to constitute a 'thorough' test, i.e., one whose successful execution implies no errors in a tested program." To guarantee the correctness of adequately tested programs, they proposed reliability and validity requirements of test criteria. Reliability requires that a test criterion always produce consistent test results; that is, if the program tested successfully on one test set that satisfies the criterion, then the program also tested successfully on all test sets that satisfies the criterion. Validity requires that the test always produce a meaningful result; that is, for every error in a program, there exists a test set that satisfies the criterion and is capable of revealing the error. But it was soon recognized that there is no computable criterion that satisfies the two requirements, and hence they are not practically applicable [Howden 1976]. Moreover, these two requirements are not independent since a criterion is either reliable or valid for any given software [Weyuker and Ostrand 1980]. Since then, the focus of research seems to have shifted from seeking theoretically ideal criteria to the search for practically applicable approximations.

Adequacy criteria provide a way to define a notion of "thoroughness" in a test suite.But they don't offer guarantees; more like *design rules to highlight inadequacy*

VI. Conclusion

Adequacy criteria provide a way to define a notion of "thoroughness" in a test suite. But they don't offer guarantees; more like *design rules to highlight inadequacy*.

The test criteria are a central problem of testing. Numerous adequacy criteria have been proposed, analyzed and compared. A lot of research has been done on the issue of evaluating and comparing criteria. The tendency is toward systematic approaches in testing using adequacy criteria.

References

1. [Chang et al., 1996] Chang, K., Cross, J., Carlisle, W., and Liao, S. (1996). A performance evaluation of heuristics-based test case generation methods for software branch coverage. International Journal of Software Engineering and Knowledge Engineering, 6(4):585{608.

2. [Chilenski and Miller, 1994] Chilenski, J. and Miller, S. (1994). Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, pages 193{200.

3. [Duran and Natfos, 1984] Duran, J. and Natfos, S. (1984). An evaluation of random testing. IEEE Trans-actions on Software Engineering, pages 438{444.

4. [Gallagher and Narasimhan, 1997] Gallagher, M. J. and Narasimhan, V. L. (1997). Adtest: A test data generation suite for ada software systems. IEEE TSE, 23(8):473{484.

5. [Horgan et al., 1994] Horgan, J., London, S., and Lyu, M. (1994). Achieving software quality with testing coverage measures. IEEE Computer, 27(9):60{69.

6. [Korel, 1996] Korel, B. (1996). Automated test data generation for programs with procedures. In Proceedings of the 1996 International Symposium on Software Testing and Analysis, pages 209{215. ACM Press.

7. [Michael et al., 1997] Michael, C., McGraw, G., Schatz, M., and Walton, C. (1997). Genetic algorithms for test data generation. In Proceedings of the Twelfth IEEE International Automated Software Engineering Conference (ASE 97).

8. [Ruchika Malhotra and Mohit Garg] Journal of Information Processing Systems, Vol.7, No.2, June 2011.An Adequacy Based Test Data Generation. Technique Using Genetic Algorithms.
9. [R. Mall.]Fundamentals of Software Engineering. Prentice Hall of India, 3rd edition. 2009
10. [Elfriede Dustin] Effective Software Testing. Addison Wesley, 2002, ISBN 0-20179-429-2
11. [M. Tikir and J. Hollingsworth] “Efficient instrumentation for code coverage testing”, Int’l. Symp. on Software Testing and Analysis, Rome, Italy, 2002.
12. [Offutt J., Liu S., Abdurazik A., and Ammann P] “Generating test data from state-based specifications”, Software Testing, Verification and Reliability, Vol. 13, pp 25-53, 2003.