Advanced Topics in Computer Science: Testing

Path Testing

Luke Gregory 321512

Professor H. Schligloff and Dr. M. Roggenbach

Abstract

Testing is a vital component of software engineering. As complex programs become integrated into all aspects of society, it is important that there exist no errors that could compromise safety, security or even financial investment. In this paper, we specifically examine path testing, and how it can be used to detect errors within a piece of code. The core reason that path testing is implemented is to provide code with a level of test coverage; that is, to find out how much of a piece of software has been examined for faults. We will be examining two methods, DD-Path and Basis Path testing, each of which provide us with different test coverage metrics, as well as their own unique benefits. Much of the work carried out on these test coverage metrics actually led to important discoveries in the field of testing, such as how using only structured constructs in a piece of code can minimise the number of required test cases. This paper will provide us with an analysis of this discovery, before concluding with a brief discussion of the merits of using a structural testing method over the functional testing methods that are also widely available to software developers.

Contents

Introduction

The reason behind testing is quite simple; to find as many faults as possible so that a piece of software will work to its maximum capabilities. Although many different types of testing exist such as data flow and equivalence class testing, this paper is going to concentrate on one particular method: path testing.

Path testing is a structural testing method that involves using the source code of a program to attempt to find every possible executable path. The idea is that we are then able to test each individual path in as many ways as possible in order to maximise the coverage of each test case. This gives the best possible chance of discovering all faults within a piece of code.

The fact that path testing is based upon the source code of a program means that it is a white box testing method. The ability to use the code for testing means that there exists a basis on which test cases can be rigorously defined. This allows for both the test cases and their results to be analysed mathematically, resulting in more precise measurement.

In order to examine path testing, we will firstly be looking at program graphs. These provide the foundation upon which the two main testing methods that will be examined in this paper are based upon. The first of these testing methods that will be analysed is DD-Paths, a concept first devised by Miller in 1977. Part of this analysis will involve studying test metrics, and how these are used to attempt to discover as many faults as possible within a program.

The second testing method that will be examined is the Basis Path Method. This paper will concentrate specifically on the work that Thomas McCabe carried out in relation to this method in the mid-1970s. In particular, there will be an examination of how the Basis Path Method works well in theory, but has difficulties that have to be overcome if it is to be implemented successfully. The thesis will finish with a brief analysis of how McCabe's work has helped to improve the way in which software is constructed so that the testing process is made easier.

Program Graphs

Program graphs are a graphical representation of a program's source code. The nodes of the program graph represent the statement fragments of the code, and the edges represent the program's flow of control.

Figure 1.1 shows pseudocode for a simple program that simply subtracts two integers and outputs the result to the terminal. The number subtracted depends on which is the larger of the two; this stops a negative number from being output.

```
1.  Program 'Simple Subtraction'
2.  Input (x, y)
3.  Output (x)
4.  Output (y)
5.  If $x > y$ then DO
6.  $x – y = z$
7.  Else $y – x = z$
8.  EndIf
9.  Output (z)
10. Output "End Program"
```

Figure 1.1 Pseudocode for the simple subtraction program.

The construction of a program graph for this simple code is a basic task. Each line number is used to enumerate the relevant nodes of the graph. It is not necessary to include basic declarations and module titles in the program graph, and so line 1 of the pseudocode in Figure 1.1 will be ignored. For a path to be executable it must start at line 2 of the pseudocode, and end at line 10. In the corresponding program graph of this code in Figure 1.2, this is demonstrated by the fact that every legal path must begin at the source node and end at the sink node.

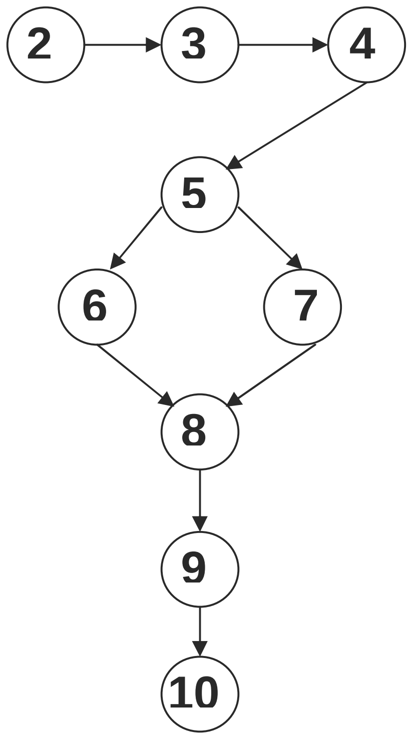

Figure 1.2 A program graph of the simple subtraction program

Due to the simplicity of our code example, it is a trivial task to find all of the possible executable paths within the program graph shown in Figure 1.2. Starting at the source node and ending at the sink node, there exist two possible paths. The first path would be the result of the If-Then clause being taken, and the second would be the result of the Else clause being taken.

A program graph provides us with some interesting details about the structure of a piece of code. In the example graph of Figure 1.2, we can see that nodes 2 through to 4 and nodes 9 to 10 are sequences. This means that these nodes represent simple statements such as variable declarations, expressions or basic input/output commands. Nodes 5 through to 8 are a representation of an if-then-else construct, while nodes 2 and 10 are the source and sink nodes of the program respectively.

By examining a program graph, a tester can garner an important piece of information; is the program structured or unstructured? It is at this point that an important distinction must be made between structure and simplicity. A program may contain thousands of lines of code and remain structured, whereas a piece of code only ten lines long may contain a loop that results in a loss of structure, and thus spores a potentially large number of execution paths. This is shown by the simple program graph in Figure 1.3 [Schach, 1993].

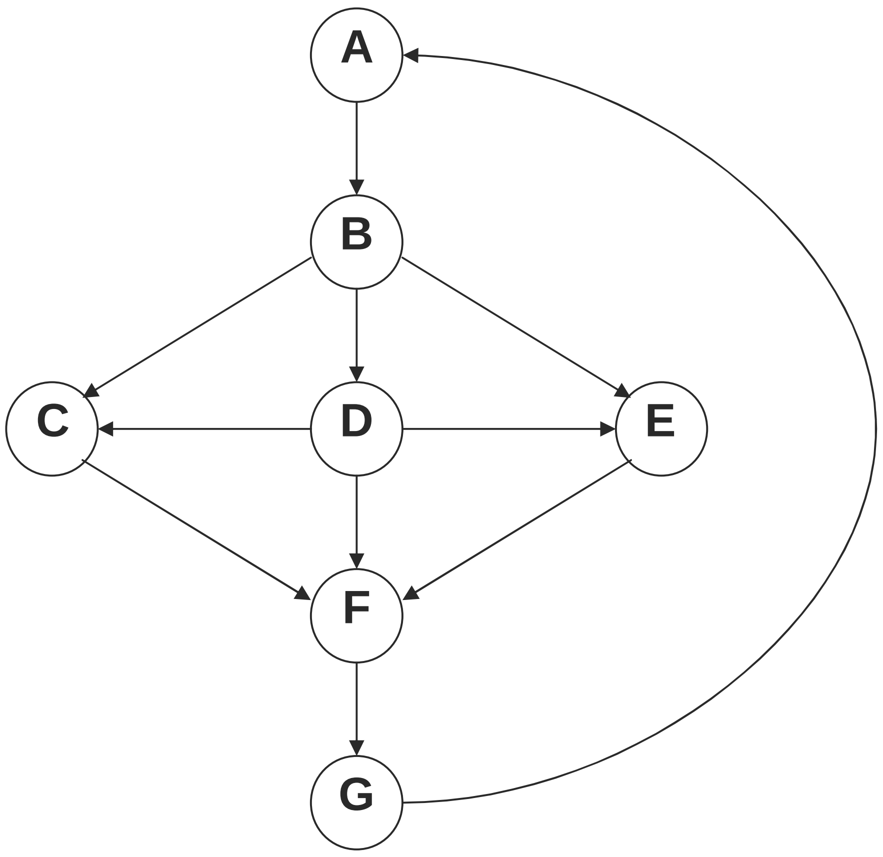

Figure 1.3 A simple yet unstructured graph

Although containing fewer nodes than the program graph in Figure 1.2, this program graph would be much more complex to test, solely because it lacks structure. This reason behind this lack of structure is due to the program graph containing a loop construct in which there exists internal branching. As a result, if the loop from node G to node A had 18 repetitions, it would see the number of distinct possible execution paths rise to 4.77 trillion [Jorgensen, 2002]. This demonstrates how an unstructured program can lead to difficulties in even finding every possible path, while testing each path would be an infeasible task. From this we can conclude that when writing a program, a software engineer should attempt to keep it structured in order to make the testing process as simple as possible. When studying the work of Thomas McCabe later in this paper, we will be looking at how he has analysed program graphs and devised a methodology to retain a program's structure, thus keeping test cases to a minimum.

DD-Paths

The reason that program graphs play such an important role in structural testing is due to the fact that they form the basis of a number of testing methods, including one based on a construct known as decision-to-decision paths (more commonly referred to as DD-Paths). The idea is to use DD-Paths to create a condensation graph of a piece of software's program graph, in which a number of constructs are collapsed into single nodes known as DD-Paths.

DD-Paths are chains of nodes in a directed graph that adhere to certain definitions. Each chain can be broken down into a different type of DD-Path, the result of which ends up as being a graph of DD-Paths. The length of a chain corresponds to the number of edges that the chain contains. The definitions of each different type of DD-Path that a chain can be reduced to are given as follows:

Type 1: A single node with an in-degree = 0.
Type 2: A single node with an out-degree = 0.
Type 3: A single node with in-degree ≥ 2 or out-degree ≥ 2.
Type 4: A single node with in-degree = 1 and out-degree = 1.
Type 5: The chain is of a maximal length ≥ 1.

All programs must have an entry and an exit and so every program graph must have a source and sink node. Type 1 and Type 2 are needed to provide us with the capability of defining these key nodes as initial and final DD-Paths. Type 3 deals with slightly more complex structured constructs that often appear in a program graph such as If-Then-Else statements and Case statements. This definition is particularly important, as it allows for branching to be dealt with in the testing process, a concept that will be examined more closely when we come to analyse test coverage metrics. Type 4 allows for basic nodes such as expressions and declarations to be defined as DD-Paths. As it is these types of nodes that make up the main part of a program, Type 5 is used to take chains of these nodes and condense them into a single node. It is important that we find the final node within a chain in order to have the smallest number of nodes as possible to test; it is for this reason that the definition of a Type 5 DD-Path must examine the maximal length of the chain.

In order to successfully demonstrate how the above definitions can be used to create a DD-Path graph, we will apply them to the program graph in Figure 1.2. The result of this application is a DD-Path graph of the simple subtraction problem, as shown in Figure 1.4.

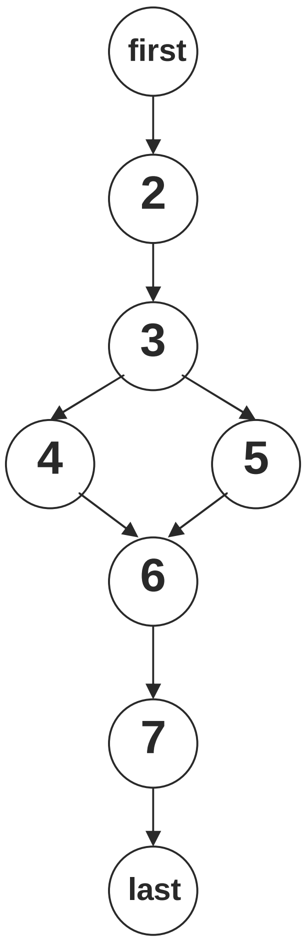

Figure 1.4 A DD-Path graph of the simple subtraction program

We can immediately identify some differences between the program graph in Figure 1.1 and its DD-Path graph. The source and sink nodes of the graph have been replaced by the words 'first' and 'last' in order to identify the nodes that conform to Type 1 and Type 2 DD-Paths. Perhaps more interestingly, there exists one less node. This is due to the fact that nodes 3 and 4 in the original program graph were a chain of maximal length ≥ 1, and so they have been condensed into a single node in the DD-Path graph

There also exist similarities between the two graphs. Node 7 remains unchanged while the If-Then-Else construct is still visible. Nodes 3 and 6 obey the Type 3 definition, while nodes 4 and 6 are simply chains of length 1 and so are defined as Type 4 DD-Paths.

Having defined the concept of DD-Paths we can now see that the construction of a DD-Path graph presents testers with all possible linear code sequences. Test cases can be set up to execute each of these sequences, meaning all paths within the DD-Path graph of the program can be tested. As a result, DD-Paths can be used as a test coverage metric; software engineers know that if they can test every DD-Path then all faults within the DD-Path graph of a program are likely to be located.

Test Coverage Metrics

We have already acknowledged the fact that DD-Paths are useful due to the fact that they act as a method for providing an accurate level of test coverage, but what does the term 'test coverage metric' actually mean?

Functional testing has the problem of gaps. This is to say that functional methods such as boundary value testing only checks maximum, minimum and nominal values, thus leaving areas of code untested for faults. Structural testing methods attempt to prevent this by having a number of different test coverage metrics that identify to the tester how much of the code has been checked for errors. Simply put, test coverage describes the degree to which a program has been tested.

There are a number of different test coverage metrics that exist. We will begin by examining the most basic: statement coverage. This is known as a C_0 test metric, and is achieved by simply testing each statement. A statement exists as a single node within a program graph, and so if each node of the graph is traversed then so is each statement. Of course, this has the obvious problem that if a node has two edges, both of which will lead to every remaining node being traversed, the C_0 metric could be satisfied without every edge having been tested. Although it does not necessarily provide 100% coverage, the C_0 test metric is still widely accepted, with it having been practiced throughout IBM since the mid-1970s [Jorgensen, 2002].

Statement testing can be improved if we allow program graphs to have nodes made up of statement fragments rather than complete statements. For example, take the following line of pseudocode:

```
If x > y then x else y
```

With average statement testing, this statement would be placed into a single node, and so only one of the predicates would be tested. However, if we allow for statement fragments, then we would get the following pseudocode:

```
If x > y then
x
Else y
```

With each of these statements now separate, they will each be placed in a different node. As a result, statement testing will go through each node with the result that we also achieve predicate outcome coverage.

Due to the fact that most program graphs are made up of complete statements, DD-Path coverage is the usual test metric used to provide coverage of both predicates and their outcome. We know that DD-Paths break If-Then-Else constructs and CASE statements into their constituent parts. This means that DD-

Path coverage will traverse each edge of a program graph rather than just each node, as this is the only way for full predicate outcome coverage to be achieved. Testing every DD-Path is the equivalent to branch testing, and is known as the C_1 test metric.

The C_2 test metric is loop coverage. Loops are one of the most complicated structures to test due to the fact that they are a highly fault prone section of code. The reason for this is that loops can be traversed hundreds of times, possibly with different values controlling the loop each time it is entered.

In 1983, Boris Beizer defined three different types of loop; concatenated, nested and horrible. These are all illustrated in Figure 1.5.

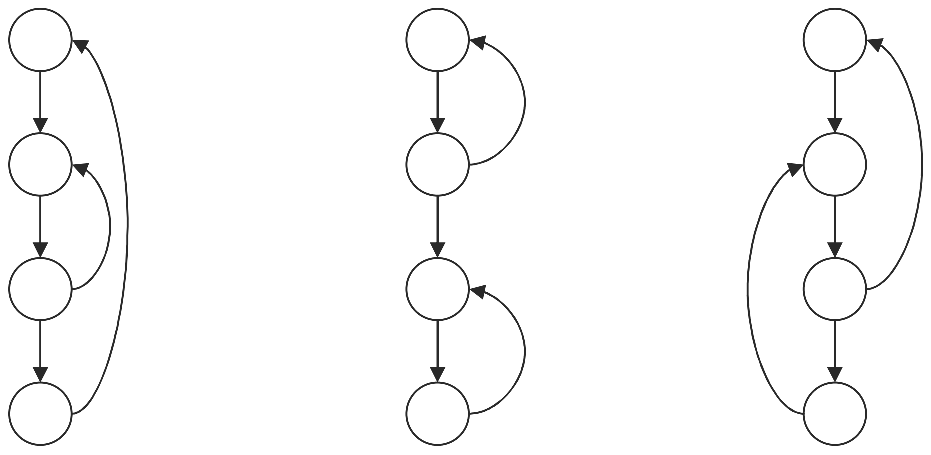

Figure 1.5 Nested, concatenated and horrible loops

Nested loops can present difficulties to a software engineer. Five tests for a single loop would be increased to 25 tests for a pair of nested loops, and 125 tests for three nested loops [Beizer, 1983]. This exponential increase of required tests means that nested loops should be avoided as a program construct. However, in some cases this construct may be unavoidable, and it is possible to employ a number of tactics to limit the number of necessary tests. For more information on this, see [Beizer, 1978].

Concatenated loops occur when it is possible to leave one loop and immediately enter into another. If the iteration values of one loop affect those of another loop, they must be treated in the same way as nested loops.

Horrible loops are a construct that involves one loop branching into the interior of another loop. This is a very difficult structure to test, as it is a challenging task to select iteration values that will provide adequate loop coverage. It is vital that good coverage of horrible loops is achieved, as if one loop alters the index value of the other loop, it is possible that the construct will be traversed infinitely. Horrible loops are now an uncommon structure, as improvements in program design allows for these constructs to be identified early on, and thus removed.

The most basic form of testing loops is to test each decision. This simply involves entering the loop once and not entering the loop. It is also possible to take a modified boundary value approach, testing a minimum, maximum and nominal value. Faults tend to congregate around loop boundaries; an example is incorrect use of equality symbols. This means that testing boundary values will make it more likely that faults will be uncovered. However, if the program contains a large number of loops, there will be a trade off between the number of faults discovered and the time taken to devise the test cases.

Another form of test coverage is the C_{MCC} test metric, commonly referred to as multiple condition coverage. The ides behind this is to carry out predicate outcome testing, but also to examine how these outcomes are reached. For example, take the following statement:

If x == 2 | | x == 6 && Boolean == true then Do

In order to carry out multiple condition coverage on this statement, it is possible to construct a truth table in which all eight possible combinations could be executed. However, there is a problem that we encounter if we attempt to do this. Multiple condition coverage does not take into account that there may exist dependencies between variables. For example, let us look at the truth table for our statement in Figure 1.6.

X	X	Boolean	Validity
T	T	T	Invalid
T	T	F	Invalid
T	F	T	Valid
F	T	T	Valid
T	F	F	Valid
F	T	F	Valid
F	F	T	Valid
F	F	F	Valid

Figure 1.6 Truth Table

We can immediately see that there are two rows in the truth table that are invalid. The reason for this is that these are the combinations in which both x values are said to be true. It is clear that it is impossible for x to be both 2 and 6, and so complete coverage can never be achieved. This makes testing all of the ways in which an outcome can be reached an infeasible criterion. Another problem is that there may exist an exponential number of possibilities that may have to be tested (a statement containing five clauses would result in 32 different tests). This makes multiple condition coverage a generally unsuitable test coverage metric to use; instead, it is more often implemented in conjunction with other test metrics as a way of making the testing process more robust.

Numerous other test coverage metrics exist, each with their own advantages and disadvantages. The majority of the coverage metrics used in testing came from the early work of E.F Miller [Miller, 1977] (Appendix A). However, the aim of all these test coverage metrics is C_∞. To achieve this type of coverage, all possible execution paths must be tested. Clearly this is not always feasible, as some pieces of software will contain infinitely many different execution paths. However, this remains the theoretical aim for testers, even if it is very rarely possible in practice.

Trying to achieve a particular test coverage metric can be an extremely difficult and arduous task. In order to hep with this process, a number of automated test coverage analysers have been produced. The idea behind a test analyser is that it produces a set of test cases that will provide coverage for a certain metric. The tester then implements all of these test cases on the program. Once completed, the analyser uses the results to produce a coverage report, indicating to the tester the degree to which the test cases that it produced cover the code. For example, if the wish of the tester was to provide C_2 coverage (loop coverage), then the test coverage analyser would find each loop within the program and provide test cases for them. Numerous different automated test coverage analysers are currently on the market (Appendix B). For instance, BullseyeCoverage produces an analyser that provides test cases for decision (loop) coverage of C++ code.

Basis Path Testing

In the 1970's, Thomas McCabe came up with the idea of using a vector space to carry out path testing. A vector space is a set of elements along with certain operations that can be performed upon these elements. What makes vector spaces an attractive proposition to testers is that they contain a basis. The basis of a vector space contains a set of vectors that are independent of one another, and have a spanning property; this means that everything within the vector space can be expressed in terms of the elements within the basis. What McCabe noticed was that if a basis could be provided for a program graph, this basis could be subjected to rigorous testing; if proven to be without fault, it could be assumed that those paths expressed in terms of that basis are also correct.

The method devised by McCabe to carry out basis path testing has four steps. These are:

1. Compute the program graph.
2. Calculate the cyclomatic complexity.
3. Select a basis set of paths.
4. Generate test cases for each of these paths

In order to show the workings of McCabe's basis path method, we will progress through each step before finally demonstrating how it could be utilised as a structural testing method. To begin, we need a program graph from which to construct a basis. Figure 1.7 shows an example taken from [McCabe, 1982]. This is a commonly used example, as it demonstrates how the basis of a graph containing a loop is computed. It should be noted that the graph is strongly connected; that is, there exists an edge from the sink node to the source node. This is necessary, for reasons that will be made clear shortly.

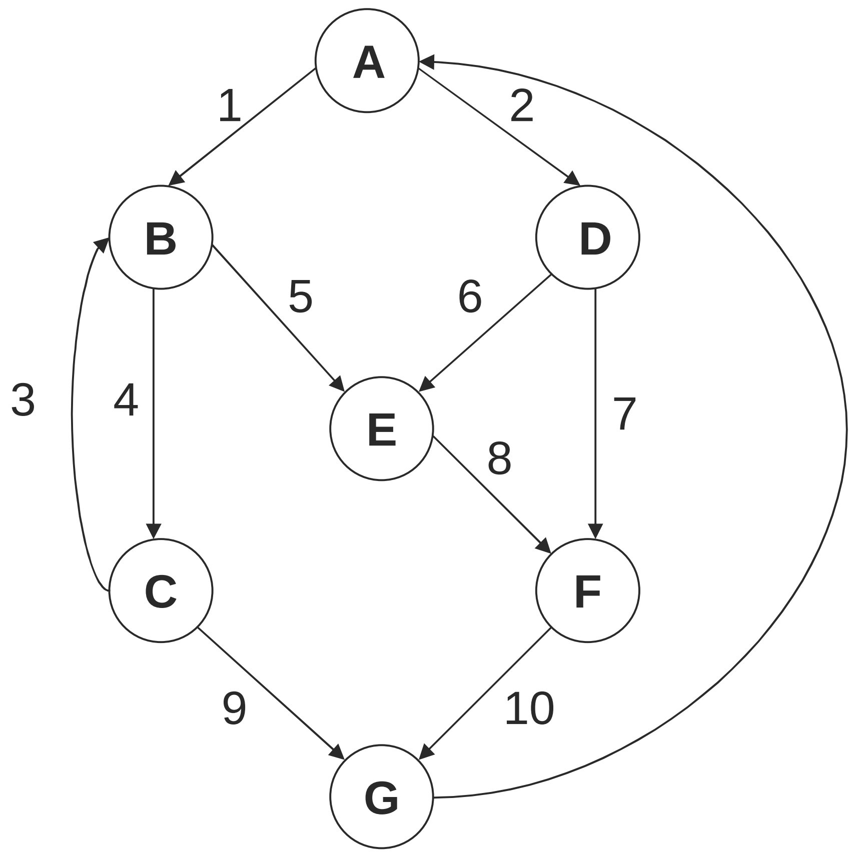

Figure 1.7 McCabe's strongly connected program graph

In graph theory, there exists a proof that states that the cyclomatic complexity of a strongly connected graph is the number of linearly independent circuits in a graph [Jorgensen, 2002]. McCabe realised that path testing commonly makes use of program graphs, and so felt that this theorem could be used to find a basis that could be tested.

The cyclomatic complexity of a strongly connected graph is provided by the formula V(G) = e – n + p. The number of edges is represented by e, the number of nodes by n and the number of connected areas by p. If we apply this formula to the graph given in Figure 1.7, the number of linearly independent circuits is:

$$\begin{aligned} V(G) &= e - n + p \\ &= 11 - 7 + 1 = 5 \end{aligned}$$

If we now delete the edge from G to A, we can see that we have to identify 5 different independent paths to form our basis. An independent path is any path through the software that introduces at least one new set of processing statements or a new condition [Pressman, 2001]. To find these paths, McCabe developed a procedure known as the baseline method [McCabe, 1996]. The procedure works by starting at the source node. From here, the leftmost path is followed until the sink node is reached. If we take the example in Figure 1.7, this provides us with the path A, B, C, G. We then repeatedly retrace this path from the source node, but change our decisions at every node with out-degree ≥ 2, starting with the decision node lowest in the path. For example, the next path would be A, B, C, B, C, G, as the decision at node C would be 'flipped'. The third path would then be A, B, E, F, G, as the next lowest decision node is B. Two important points should be made here. Firstly, if there is a loop, it only has to be traversed once, or else the basis will contain redundant paths. Secondly, it is possible for there to be more than one basis; the property of uniqueness is one not required.

The five linearly independent paths of our graph are as follows:

Path 1: A, B, C, G.
Path 2: A, B, C, B, C, G.
Path 3: A, B, E, F, G.
Path 4: A, D, E, F, G.
Path 5: A, D, F, G.

This now forms the basis set of paths for the graph in Figure 1.7. In theory, if we allow for the basic notions of scalar multiplication and addition, we should now be able to construct any path from our basis. Let us attempt to create a 6th path: A, B, C, B, E, F, G. This is the basis sum p2 + p3 – p1. This equation means to concatenate paths 2 and 3 together to form the path A, B, C, B, C, G, A, B, E, F, G and then remove the four nodes that appear in path 1, resulting in the required path 6.

We clearly now have a basis, which, in theory, can be rigorously tested, thus verifying all other paths constructed from its elements. However, in the real world, this is not always possible. The first problem is that the basis sums that need to be calculated in order to create paths are unsatisfactory. Terms such as p1 could be misinterpreted. We use it to mean remove all nodes traversed in path 1 from the path that we are currently constructing. However, it could be read as meaning do not carry out p1, or even execute the path p1 backwards. For testers, terms that are vague could result in them missing faults within a basis path, thus resulting in the whole testing procedure being compromised.

However, the basis path method hides an even larger problem. When constructing the basis set of a program graph, it is possible that some of the paths that make up the basis could be infeasible. This would result in the ability to construct further infeasible paths, and could lead to testers trying to find faults in paths that cannot exist. The basis path method considers decisions to be independent. If a decision is taken at the source node, it does not take into account that it could have a direct impact taken on a node further down the program graph. For example, a decision may be taken at node B in Figure 1.7 that makes it impossible to travel through node E. However, the nature of the baseline method for calculating paths is just to reverse the decision at each node, including node B, meaning that we now have a path in our basis that cannot possibly ever be travelled.

There do exist different solutions to this problem. Rather than just 'flipping' the decision at each node with out-degree ≥ 2, there is first a check to see if the path is semantically legal. Likewise, rules can be identified that can be put in place before the basis path method is executed, thus restricting the basis to feasible paths only. Of course, both these solutions have the disadvantage of more work for the tester. However, if these solutions can be implemented then the basis path method is a very elegant way of testing every decision outcome. This is the equivalent of DD-Path testing, and thus provides an average of 85% program coverage [Miller, 1991].

Essential Complexity

When carrying out his work on the basis path testing method, McCabe developed the notion of what is now known as essential complexity. This is the term given for using the cyclomatic complexity to produce a condensation graph; the result is a graph that can be used to assist in both programming and the testing procedure.

The concept behind essential complexity is that the program graph of a piece of software is traversed until a structured programming construct is discovered; examples of these constructs are shown in Figure 1.8. Once located, the structured programming construct is collapsed into a single node and the graph traversal continues. The desired outcome of this procedure is to end up with a graph of V(G) = 1, that is, a program made up of one node. This will mean that the entire program is composed of structured programming constructs.

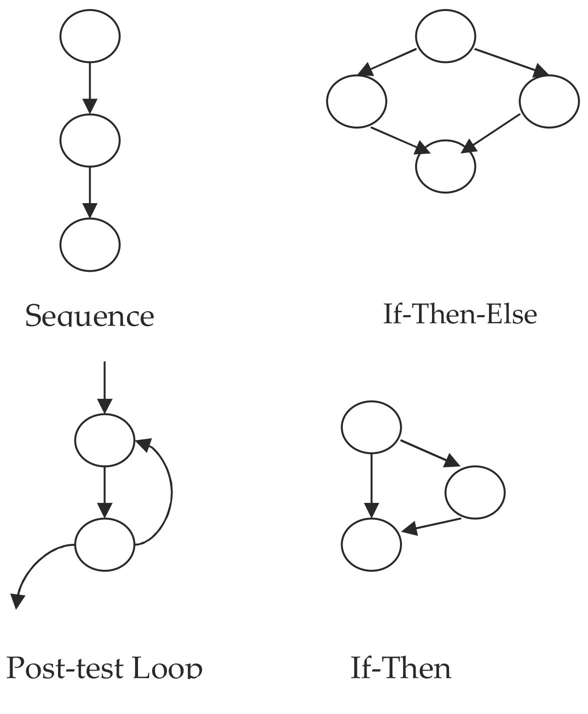

Figure 1.8 McCabe's Structured Constructs

If a graph cannot be reduced to one in which there is a cyclomatic complexity of 1, then it means that the program must contain an unstructured programming construct. Some examples of these unstructured constructs are shown in Figure 1.9. The reason that these are not structured is because they contain three distinct paths. The result of all of this is that if an unstructured construct is found in the program graph, it means one extra path will exist that cannot be collapsed, and so the cyclomatic complexity increases. What McCabe unearthed was the fact that if one unstructured programming construct exists, then there must be at least one more. This means the use of one of these 'unstructures' can result in a serious deterioration in the structure of the program.

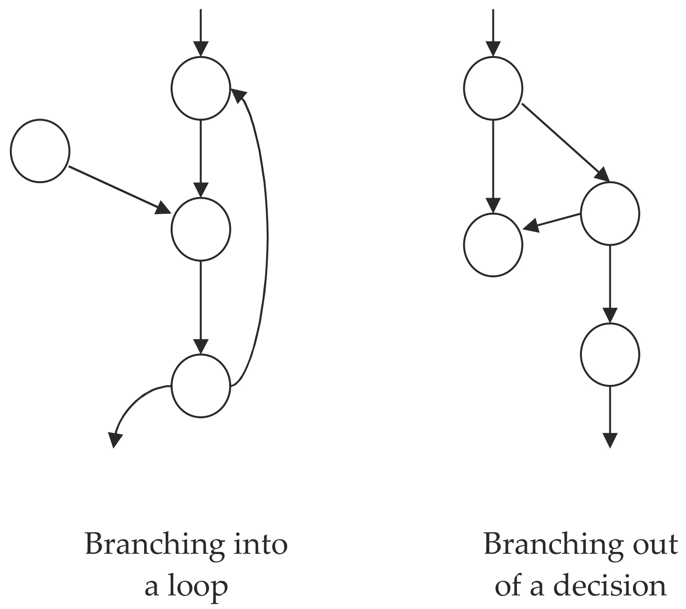

Figure 1.9 McCabe's unstructured constructs

The question is, how does McCabe's work benefit testers? For every increase in the cyclomatic complexity, the minimum number of test cases for that program increases. This means that it can be used by testers as a way of finding how much work they have to carry out in order to discover the maximum number of faults. However, what essential complexity does is improve the craft of programming. Software engineers know that unstructured programming constructs can lead to programs with a large number of minimum test cases, and so they can avoid using these constructs, thus making the code easier to test. The majority of organisations that use this cyclomatic complexity metric have a maximum acceptable complexity for a program, with V(G) = 10 being a common choice [Jorgensen, 2002].

Conclusion

In this paper we have carried out an in-depth analysis of path testing, and the different methods that exist. What we have found is that no matter which method is implemented, be it DD-Path testing or the Basis Path testing method, the aim of path testing remains the same; to provide a test coverage metric so that the tester has information concerning the amount of faults within the code that are likely to have been uncovered.

The reason behind why the test coverage metrics provided by path testing is so important is that they provide a target for testers to aim for. Without this target, when would the testing process end? It is impossible to test until all faults have been removed as this can never be guaranteed, while testing until no new faults are uncovered can be an expensive process. However, it is viable to test a program until a specific test coverage metric has been achieved. This means that once a program has been fully tested in respect to a certain metric, the tester has an idea of the percentage of faults that have been uncovered. For example, if DD-Path coverage is achieved, statistics show that 85% of faults are found on average. This level of information makes test coverage metrics one of the few possible methods of determining when the testing procedure is complete.

Throughout this paper, we have mainly extolled the virtues of path testing, but like any testing method, it has weaknesses. It is virtually impossible to test every path, and it can also be difficult to test programs that contain many unstructured constructs, although this is the case with most testing methods. Path testing also has the stigma of being slightly outdated; much of the work concerned with path testing was carried out in the 1970s and 1980s, and was associated more with imperative languages such as C. We now live in an age where the development of object-oriented programming has meant that functional testing methods have become more popular. Functional testing has the benefit of not needing the source code; it is a black-box testing method, and so it only tests the program through external interfaces. This means that functional testing methods such as equivalence class testing checks whether the behaviour detailed in the program specification is achieved; path testing cannot accomplish this.

This does not mean that there is not a place for structural testing methods such as path testing. They have the ability to find incorrectly implemented behaviours such as viruses, something functional tests cannot uncover. Path testing also has one major advantage over functional testing methods: it does not suffer from gaps or redundancies. In the worst case, the functional method of boundary value testing can leave a number of paths untested, while testing others more than once [Jorgensen, 2002].

In reality, functional and structural testing methods are often blended together to achieve the best possible level of testing. The result of this is that a method known as data flow testing is often implemented. This combines the benefit of providing test coverage metrics with the advantages provided by functional testing (for more information on this method of testing, see [Clarke, 1989]). However, for the time being, the benefits of path testing, along with the number of test coverage analysers available on the market, means that it is still likely to be a force in the field of testing for the foreseeable future.

References

[Beizer, 1978] Beizer, B., *Micro-Analysis of Computer System Performance,* 1978. New York: Van Nostrand Reinhold.

[Beizer, 1983] Beizer, B., *Software Testing Techniques*, pp. 37-73, 1983. New York: Van Nostrand Reinhold.

[Clarke, 1989] Clarke, Lori A., *A Formal Evaluation of Data Flow Path Selection Criteria* pp. 1318-1332, November 1989.

[Jorgensen, 2002] Jorgensen, P., *Software Testing: A Craftman's Approach,* 2nd Edition, pp. 137-156, 2002. CRC Press Inc.

[McCabe, 1982] McCabe, Thomas, J., *Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric,* 1982. NIST Special Publication 500-99, Washington D.C.

[McCabe, Watson, 1996] McCabe, Thomas, J., Watson, Arthur, H., *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric,* 1996. NIST Special Publication 500-235.

[Miller, 1977] Miller, E.F., *Tutorial: Program Testing Techniques*, 1977. COMPSAC '77 IEEE Computer Society.

[Miller, 1991] Miller, Edward, Jr., *Automated Software Testing: A Technical Perspective*, Vol. 4, pp. 38-43, 1991. Amer, Programmer, No. 4.

[Pressman, 2001] Pressman, Roger, *Software Engineering: A Practitioner's Approach, 5th Edition*, 2001. McGraw Hill, Boston.

[Schach, 1993] Schach, Stephen, R., *Software Engineering, 2nd Edition*, 1993. Richard D. Irwin, Inc., and Asken Associates, Inc.

Appendix A

Structural Test Coverage Metrics

Metric	**Description of Coverage**
C_0	Every statement
C_1	Every DD-Path (predicate, outcome
C_{1p}	Every predicate to each outcome
C_2	C_1 coverage + loop coverage
C_d	C_1 coverage + Every dependent pair of DD-Paths
C_{MCC}	Multiple condition coverage
C_{ik}	Every program path that contains up to k repetitions of a loop (usually k = 2)
C_{stat}	"Statistically significant" fractions of paths
C_∞	All possible execution paths

Appendix B

Selected Automated Test Coverage Analysers

BullseyeCoverage: Provides functional coverage for quickly assessing overall coverage, and decision coverage for detailed testing of C++ source code.

McCabe IQ: Provides multiple levels of comprehensive code coverage, including branch, path and module coverage, as well as class coverage for object oriented source code.

Ncover (Code Mortem): Code coverage analysis tool that gives line-by-line code coverage statistics.

Prof-It for C#: A standalone profiler for C# that measure execution frequencies for each statement while keeping the instrumentation of the source code to a minimum.

Resource Standard Metrics (RSM): A source code metrics and analysis tool providing a standard method for analysing C, C++, C# and Java source code across operating systems.

Swat4j: Provides an automatic source code-auditing tool for Java. It comes with over 30 test coverage metrics for Java, as well as over 100 "best practice" rules for industry coding.