CHAPTER 1
INTRODUCTION

1.1 Software Testing

Testing is an important field in software engineering. Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. Software testing is an inevitable and an important phase in the life cycle of a software development process.

Software testing is defined as the process of executing the program with the intent of finding an error and evaluating the developed software to ensure that it correctly implements a specific function and is traceable to customer requirements. Testing is a critical element of software quality assurance that ensures the correctness, completeness and quality of the developed software [31].Software needs to be tested properly and thoroughly, so that any misbehaviour during the runtime can be detected and fixed in advance, before its delivery. Incorrect software which is released to the market without being fully tested could result in customer dissatisfaction. During software testing, a minimal number of test cases are designed so that it reveals as many faults as possible. Test cases are specification of the inputs to the test and the expected output from the system. A test case is said to be successful if it succeeds to discover new errors and unsuccessful otherwise. In other words, a good test case is one that has a high probability of detecting an as-yet undiscovered error [40].Test data are the inputs that is written to test the system.

Majority of the software practitioners agree with the motto “test early, test often and test enough” [26]. This motto helps to uncover problems early in the development process, which in turn reduces the size of effort required to perform adequate system testing by determining what needs to be tested. The recent iterative development processes focus on analyzing a little, designing a little, coding a little and testing what you can [26]. It cannot be sufficiently overemphasized that regular testing can detect failures early in the software development and save reworking in subsequent iterations. A general model for the testing process is shown in figure 1.1.

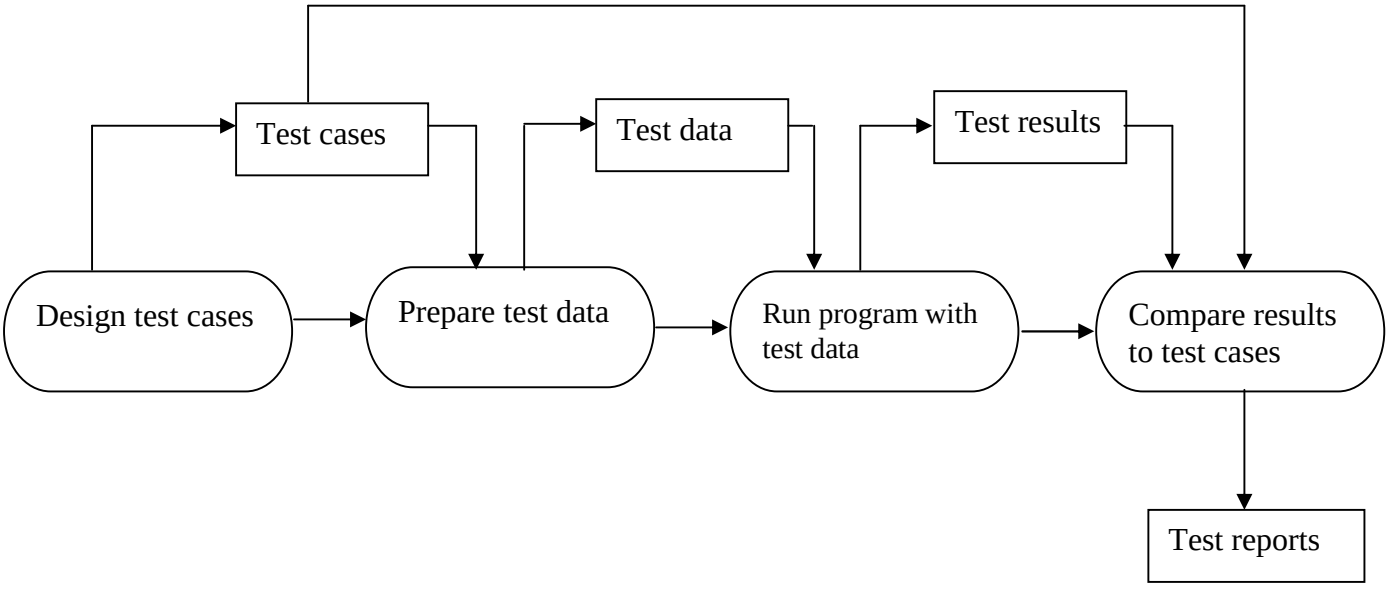


Figure 1.1: A block diagram of the software testing process

1.2 Testing Objectives

The software testing process has two distinct goals:

- To discover faults or defects in the software where the behaviour of the software is incorrect, undesirable or does not conform to its specification.
  This is the primary objective of testing. This goal leads to defect testing, where the test cases are designed to expose defects. For defect testing, a successful test is one that exposes a defect that causes the system to perform incorrectly.

- To demonstrate to the developer and the customer that the software meet its requirements.
  This goal leads to validation testing, where we expect to perform correctly using a given set of test cases that reflect the system's expected use. For validation testing, a successful test is one where the system performs correctly.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstances. Even well-tested software is also not guaranteed to be error-free or bug-free. Ideally, software testing guarantees the absence of errors in the software. According to Dijkstra, a leading early figure in the development of software engineering, eloquently stated "Testing can only show the presence of errors, not their absence." [1] [35]

Therefore the goal of software testing is to convince system developers and customers that the software is good enough for operational use [35]. Software testing also increases confidence of the programmers in the correctness and reliability of the program being tested [13].

1.3 Testing Principles

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. A set of testing principles are as follows:

- All tests should be traceable to customer requirements.
  As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects are those that cause the program to fail to meet its requirements.

- Tests should be planned long before testing begins.
  Test planning can begin as soon as the requirement model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

- The Pareto principle applies to software testing.
  Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

- Testing should begin “in the small” and progress toward testing “in the large”.
  The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

- Exhaustive testing is not possible.
  The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover a limited number of important logical paths and to test them.

- To be most effective, testing should be conducted by an independent third party.
  By most effective, it is meant that testing that has the highest probability of finding errors. So; the software engineer who created the system is not the best person to conduct all tests for the software.

1.4 Justification for Research

Software testing is a laborious and time consuming work of software development process; between 40% to 50% of software development cost is spent in software testing[5] [31][33][35][37][40].It consumes resources and adds nothing to the product in terms of functionality. The increasing visibility of software as a system element and the attendant costs associated with a software failure are motivating forces for well planned, thorough testing.

Automating the testing process is a sound software engineering approach, which can make the testing efficient and reliable. Software testing can significantly reduce the cost of developing software, even if only a small part of the whole process could be automated. However, automation of software testing is not a straightforward and easy process. Automated software testing requires the analysis of some formal object such as source code or a formal specification [34].One of the biggest problems in automation of software testing is how to determine test cases. One technique that is in widespread use is to draw the Control Flow Graph (CFG) of the program and generate a basis set of test paths.

For years, many researchers have proposed different approaches to generate basis set of paths. Despite growing interest of researchers in software metrics, construction of CFG is still empirical. Although mathematical support exists for studying and manipulating CFG; there is no systematic method either for representing or deriving CFG. Construction of CFG manually and generation of basis set of paths from CFG takes more time and reduces scalability of generator. CFG construction is a non trivial process for basis path testing and has usually been underestimated.

The procedure for deriving the CFG and even determining a basis set of paths can be automated. Such an automated software tool assists in basis path testing. Testing a program by basis path testing method will also enhance the testing efficiency effectively [16].The

requirement for higher quality software demands a more systematic approach to testing. Hence, the motivation for selecting this topic for research is due to the importance of software testing and its implications with respect to software quality.

1.5 Research Goal

The main objective of this research would be to design and develop a tool for the automatic construction of a CFG, computation of cyclomatic complexity and generating basis set of paths from CFG. The tool will consist of module for performing the following tasks:

- Drawing the CFG using the pseudocode of the program.
- Determining the cyclomatic complexity of the resultant graph.
- Determining the basis set of linearly independent paths.

1.6 Organization of Thesis

The rest of thesis report is organized as follows:

- Chapter 2 gives a succinct background of software testing, focusing on conventional and object-oriented testing paradigms as well as different software testing techniques.
- Chapter 3 gives the details of the methodology used to achieve the research goal and produce results.
- Chapter 4 presents the pseudocode of the program and different facts related to the program.
- Chapter 5 presents the experiments setup, results and analysis.
- Chapter 6 presents the conclusions of the thesis followed by future work.

CHAPTER 2

LITERATURE SURVEY ON SOFTWARE TESTING

Testing is the most widely used and accepted technique for verification and validation of software systems. It is applied to measure the extent to which a software system conforms to its original requirements and to demonstrate its correct operation. Testing identifies problems and failures in all the phases of software development.A successful test is one that uncovers an as-yet-undiscovered error [31].

There are two types of software's to test: conventional software and object-oriented software.

2.1 Conventional Software Testing

Conventional software testing refers to test systems following a procedural programming approach. These systems rely mostly on walkthroughs and inspections (static verification) to remove faults. Testing of conventional software can be carried out at three different levels:

- **Unit Testing:** It concentrates on testing each unit (or component) of the software as implemented in the source code. IEEE defines unit testing as "the testing of individual software or hardware units or groups of related units" [2]
- **Integration Testing:** It concentrates on systematic testing of multiple units of software. The components that are integrated may be off-the-shelf components, reusable components that have been adapted for a particular system or newly developed components.
- **System Testing:** It concentrates on testing the software as a whole. In an interactive development process, system testing is concerned with testing an increment to be delivered to the customer, in a waterfall process; system testing is concerned with testing the entire system.

2.2 Object-Oriented Software Testing

Object-Oriented (OO) software testing refers to test systems following an object-oriented programming approach. Object-Oriented (OO) paradigm increases software reusability, extensibility, interoperability and reliability.

Object-oriented testing, in spite of being strategically similar to conventional testing, is tactically very different. With OO systems the three traditional testing phases of unit testing, integration testing and system testing are replaced with the following four levels [11] [15][31]:

- **Algorithmic/Method Level Testing:** It tests the implementation of each member function in a given class.
- **Class Level Testing:** It tests the interaction between different methods and data in a given class. This is also known as intra-class testing as classes are tested in isolation.
- **Cluster Level Testing:** It tests the interaction between different classes in a given cluster. This is also known as inter-class testing as sets of classes are tested together.
- **System Level Testing:** It tests the interaction between different clusters in a given system. This is similar to conventional software's system level testing where the software is tested as a whole.

OO testing at the algorithmic and system levels is similar to conventional testing [12]. However, OO testing at the class and cluster levels poses new challenges. Unit testing is replaced with class level testing in object-oriented systems. Class level testing involves testing a single object in isolation. This is much more complex than traditional unit testing because a class may have several methods, attributes, possibly some kind of inheritance, and also relationships with other classes.

The advantages of object oriented approach become potential disadvantages while testing, as object-oriented code is considerably harder to read than conventional source code. According to [31], "the testing of object-oriented systems presents a new set of challenges to the software engineer". Object-oriented testing is much more complex than conventional testing due to the following reasons:

- **Encapsulation:** Encapsulation is defined as the packing or binding together of a collection of items**.** Encapsulation complicates testing because operations must be added to a class interface (by the developer) to support testing. A member function of a class may invoke several other member functions from different object classes to achieve an intended functionality. The implication of long invocation chains is that a tester has to understand the sequence of member functions and semantics of the class prior to preparing test cases.

- **Abstraction:** Abstraction is a mechanism that enables the designer to focus on the essential details of a program component (either data or process) with little concern for lower level details. Abstraction of data and code in OO programming complicates the testing process. Testing from the highest level of abstraction provides more effective and accurate set of tests.

- **Cyclic dependency:** It makes difficult to understand a given class in a large OO program if that class depends on many other classes. It is difficult to test inheritance, aggregation, association, template classes, dynamic object creation, polymorphism and dynamic binding relationships. This makes it hard for a tester to know where to start testing in an OO library. It is extremely costly to construct stubs. The impact of a small change may ripple throughout the OO program due to the presence of complex dependencies.

- **State behavior:** Documentation for OO testing is either missing or poor. Objects have states and state dependent behaviors. Control flow and state control is distributed over several classes, making it difficult to generate test cases.

- **Inheritance:** Inheritance is a process by which objects of one class acquire the properties of objects of another class. Inheritance occurs throughout all levels of a class hierarchy. Inheritance complicates the testing because bugs may propagate from a parent class to each of its descendants. Some of the test cases for superclass can be reused for subclasses. However, the use of inheritance solely for code and test case reuse will lead to difficulties.

- **Polymorphism:** Polymorphism is another important OO concept. Simply, polymorphism means the ability to take more than one form**.** Polymorphism puts more importance on testing a representative sample of runtime/dynamic configurations. Polymorphism leads to more run-time errors; hence requiring explicit unit testing.

- **Tool support:** Tool Support for OO testing is still in infancy. The tools that support OO testing (Computer Assisted Software Engineering (CASE) tools) are in the development phase. Formal specifications are rarely used for testing. Hence, testing remains a tedious process where the tester has to manually prepare the test cases.

2.3 Software Testing Techniques

Generally, software-testing techniques are classified into two categories: static testing and dynamic testing [4][14][19][20] [28] [35]. In static testing, a code reviewer reads the program source code, statement by statement, and visually follows the logical program flow by feeding an input. This type of testing is highly dependent on the reviewer's experience. Static testing uses the program requirements and design documents for visual review. In contrast, dynamic testing techniques execute the program under test on test input data and observe its output. Usually, the term testing refers to just dynamic testing.
The following subsections give a brief background on these two testing categories

2.3.1 Static Testing

For years, the majority of the programmers assumed that the programs are written solely for machine execution and are not intended to be read by human and that the only way to test a program is by executing it on a machine. This manner began to change in the early 1970s, because of the Weinberg's work on "The Psychology of Computer Programming". Weinberg [39] provided a convinced argument for why programs should be read by people and indicated that this could be an effective error-detection process.

Experience has shown that static testing, also known as, non-computer-based or human testing, methods are quite effective in finding errors [28][39]. Static analysis methods are meant to be applied during the period that is between the code completion and the beginning of the execution-based testing. Static testing is employed to verify the correctness of requirements, designs, and code before execution of test cases.

Typical static testing methods are code inspections, code walkthroughs, desk checking, and code reviews [4][21][23][28][31][35]. Code inspections and walkthroughs are the two primary static testing methods and they have a lot in common. Inspections and walkthroughs involve the reading or visual inspection of a program by a team of people. Both methods involve some preparatory works by the participants. The climax is a meeting of the minds, i.e. brainstorming, in a conference-like gathering held by the participants. The objective of the meeting is to find errors, but not to find solutions to the errors, i.e. to test but not to debug.

The different static testing methods are as follows:

2.3.1.1 Code Inspections

Code inspection is a set of procedures and error-detection techniques for group code reading to find errors, omissions and anomalies. Generally, inspections focus on source code, but any readable representations of the software such as its requirements or design model can be inspected. During the inspection session, two activities are conducted: code narration and code examination. Code is read statement by statement and analyzed with respect to a checklist of historically common programming errors (e.g. data-reference, data-declaration, computation, comparison, control-flow, input/output, interface).

2.3.1.2 Code Walkthroughs

The initial procedure is identical to that of the inspection process. The difference, however, is in that rather than simply reading the program or using error checklists, one of the participants designated as a tester comes to the meeting with a small set of paper test cases that represent sets of input and expected output for the tested program or module. During the meeting, each test case is mentally executed, i.e. the test data are walked through the logic of the program. The state of the program, i.e. the values of the variables, is monitored on paper or a blackboard.

Definitely, the test cases must be simple in nature and few in number, because people execute programs at a much slower rate than a machine. Thus, the test cases themselves do not play a critical role; rather, they serve as a vehicle for getting started and for questioning the programmer about his or her logic and assumptions. In most walkthroughs, more errors are found during the process of questioning the programmer than are found directly by the test cases themselves.

2.3.1.3 Desk Checking

Desk checking can be seen as a one-person inspection or walkthrough; a person reads a program, checks it with respect to an error list, and/or walks test data through it. There are three main reasons that desk checking, for most people, is relatively unproductive: completely undisciplined process, the principle that people are generally ineffective in testing their programs, and no competition like in the teamwork.

2.3.1.4 Code Reviews (Peer Ratings)

Code review is a technique for evaluating anonymous programs in terms of their overall quality, maintainability, extensibility, usability, and clarity. The purpose of the review is to provide programmer assessment. A group of programmers is given some selected programs to rate based on a certain scale written in the review forms.

2.3.2 Dynamic Testing

Dynamic testing techniques execute the program under test on test input data and observe its output. Usually, the term testing refers to dynamic testing. There are two popular ways to perform dynamic testing, namely black-box testing and white-box testing.Either of these two methods requires a set of well developed and well structured test cases. White-box testing takes into account the internal structure of the program under test i.e. source code of the program. It is concerned with the degree to which test cases exercise or cover the logical flow of the program [28].Black-box testing, on the other hand, doesn't consider the internal structure of the program and tests the functionalities of software regardless of its internal structure.

The following subsections give a brief background on these two types of dynamic testing.

2.3.2.1 Black Box Testing

Black-box testing, also known as, functional or specification based testing or behavioral testing, tests the functionalities of software against its specification, regardless of its structure. This testing technique is called black-box because the module to be tested is treated simply as a transfer function. The tester isn't interested in the interior makeup of the box, but just its functional performance as it converts input to output. The philosophy is that if the module accepts all test cases inputs and produces correct results (in function, timing and performance), the tester doesn't care how it is done.

Black-box testing attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors.

There are different types of black-box testing: Equivalence Partitioning, Boundary Value Analysis, Graph-Based Testing, Cause-and-Effect Graphing, and Error Guessing [4][28] [31][38].

i) Equivalence Partitioning

Equivalence partitioning is a black-box testing method that divides the input domain of a program into a finite number of equivalence classes from which test cases can be derived [31]. Each equivalence class contains a specific input condition and a description of a valid and invalid input for that input condition. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a boolean condition. One can assume that a test of a representative value of each class is equivalent to a test of any other value within the corresponding class. That is, if one test case in an equivalence class detects an error, all other test cases in the equivalence class would be expected to find the same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the equivalence class would find an error. The equivalence partitioning concept may be applied to white-box testing as well.

ii) Boundary Value Analysis

A greater number of errors tend to occur at the boundaries of the input domain rather than in the center [31]. It is for this reason that boundary value analysis has been developed as a testing technique. It leads to a selection of test cases that exercise bounding values. This test case design technique complements equivalence partitioning. Experience shows that test cases that explore boundary conditions have a higher payoff than test cases that do not [28].

iii)Graph-Based Testing Method

In this black box testing method, software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered[38].To accomplish these steps, the software engineer begins by creating a graph- a collection of nodes that represent objects, links that represent the relationships between objects, node weights that describe the properties of a node and link weight that describe some characteristics of a link.

iv)Cause-and-Effect Graphing

This technique also attempts to bring a systematic approach to design of modules tests by examining the module in terms of causes (inputs) and effects (outputs) [4]. By developing a

cause-and-effect graph by using a special notation or by simply developing a decision table for the module, the software engineer can enlarge the pool of applicable black box test cases.

v) Error Guessing

Error guessing is largely an intuitive and ad-hoc process, whose procedure is difficult to formalize. This technique needs an expertise that is able to smell out errors. The basic idea is to enumerate a list of possible errors or error-prone situations and then write test cases based on the list.

2.3.2.2 White-Box Testing

White-box testing is also known as structural testing or glass-box testing or clear box testing. White-box testing is described in [2][42] as follows: White-box testing refers to the testing that takes into account the internal mechanism of a system or component. It focuses on the procedural details (or internal structure) of the software when generating test data and test cases. Using white-box testing methods, the software engineer can derive test cases that

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true and false sides.
- Execute all loops at their boundaries and within their operational bounds
- Exercise internal data structures to ensure their validity

The major objective of white-box testing is to focus on internal program structure, and discover all internal program errors [28]. Black-box testing and white-box testing uncover a different class of errors and hence are complementary to each other. Black-box testing is unlikely to uncover numerous sorts of defects in the program. These defects can be of the following nature:

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement functions, conditions or controls that are out of the program.
- The logical flow of the program is sometimes counterintuitive, meaning that our unconscious assumptions about flow of control and data may lead to design errors that are uncovered only when path testing starts.
- Typographical errors are random, some of which will be uncovered by syntax checking mechanisms but others will go undetected until testing begins.

Each of these reasons provides an argument for conducting white-box testing. Black-box testing, no matter how thorough, may miss the kinds of errors noted here. White- box testing is far more likely to uncover them. The white-box testing mainly focuses on program structures and branches, various kinds of program paths, programs internal logic and data structures, program internal behaviors and states.

But, the total number of paths even in a small program may be very large resulting in a very large number of test cases. So, exhaustive white-box testing even for simple program module is, practically speaking, impossible [4]. White-box testing is performed early in the testing process whereas black-box testing is applied during the later stages of testing. White-box testing is mostly used at unit level testing whereas black-box testing is used at integration or system level testing. Some of the white-box testing methods include:

i) Control Flow Testing

Control flow testing is a white-box testing technique that tests the flow of control in a program[41]. The variations of control structure testing that can broaden test coverage and improve quality of white-box testing are:

- **Basis Path Testing:** It is a white-box testing technique for control flow testing [31].It was first proposed by Thomas McCabe in the mid 1970s. This technique uses the control flow graph of a program module to generate a set of independent paths known as basis set of paths. Test cases derived to exercise the basis set are guaranteed to execute all statements in the program at least once during testing and conditional statements in the program are also guaranteed to execute on its true and false sides [31]. Basis path testing enables the designer to derive a logical complexity measure, known as cyclomatic complexity, of the procedural design that is used as a guideline for defining a basis set of execution paths. Basis path testing is a hybrid between path testing and branch testing. Although basis path testing is simple and highly effective, but it is not sufficient in itself.

- **Condition Testing:** It is a test case design method that exercises the logical conditions contained in the program module. It focuses on testing each condition in the program. A condition can be made up of relational (<, >, =, !=, <= or >=) or Boolean (AND, OR, NOT) operators. Furthermore, a condition can be simple (only one operator) or compound (two or more operators). A number of condition testing strategies exist:

- **Branch Testing:** It is the testing designed to execute each outcome of each decision point in a computer program [2].It is probably the simplest condition testing strategy. For a compound condition C, the true and false branches of C and every simple condition in C need to be executed at least once.
- **Domain Testing:** It requires three or four tests to be derived for a relational expression of the form E <relational-operator> F. The three tests are required to make the value of E greater than, equal to or less than that of F. For a Boolean expression with n variables, all of 2^n possible tests are required (n > 0). This strategy is practical only if n is small.

Condition testing strategies have two advantages: measurement of test coverage of a condition is simple and it provides guidance for generating additional tests for the program. Therefore, the purpose of condition testing is to detect errors both in the conditions of a program and in the program itself.

ii) Data Flow Testing

Data flow testing method selects test paths of a program according to the location of definitions and subsequent uses of variables in the program. For the data flow testing approach, each statement in a program is assigned a unique statement number. For a statement with S as its statement number,

DEF(S) = {X/statement S contains a definition of X}
USE(S) = {X/statement S contains a use of X}

For the statement S:a=b+c; DEF(S) = {a} and USE(S) = {b,c}. The definition of variable X at statement S is said to be live at statement S1, if there exists a path from statement S to statement S1 which does not contain any definition of X.The definition-use chain (or DU chain) of a variable X is of form [X, S, S1], where S and S1 are statement numbers, such that X is in DEF(S) and USE (S1), and the definition of X in the statement S is live at statement S1.

A number of data flow testing strategies exist One simple data flow testing strategy is to require that every DU chain be covered at least once. Data flow testing strategies are useful for selecting test paths of a program containing nested if and loop statements [31].

iii) Loop Testing

Loop testing focuses exclusively on the validity of loop constructs [31]. Loops are a major part of all algorithms implemented in the software. Four different classes of loops can be defined:

- **Simple Loop:** It consists of a single loop with n iterations where n is the maximum number of allowable passes through the loop.
- **Nested Loop:** It consist of two or more simple loops with each loop having a number of iterations. The number of tests would grow geometrically as the level of nesting increases.
- **Concatenated Loop:** It consist of two or more simple loops that can be dependent or independent of each other.
- **Unstructured Loop:** It is highly complex and should be redesigned to fit the above cases.

iv) Mutation Testing

In mutation testing, the software is first tested by using an initial test suite built up from the different white box testing strategies. After the initial testing is complete, mutation testing is taken up. The idea behind mutation testing is to make few arbitrary changes to a program at a time[24][27][29]. Each time the program is changed, it is called as a mutated program and the change effected is called as a mutant. A mutated program is tested against the full test suite of the program. If there exists at least one test case in the test suite for which a mutant gives an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test data is enhanced to kill the mutant.

The process of generation and killing of mutants can be automated by predefining a set of primitive changes that can be applied to the program. These primitive changes can be alterations such as changing an arithmetic operator, changing the value of a constant, changing a data type, etc.

A major disadvantage of the mutation-based testing approach is that it is computationally very expensive, since a large number of possible mutants can be generated. Since mutation testing generates a large number of mutants and requires us to check each mutant with the full test suite, it is not suitable for manual testing. Mutation testing should be used in conjunction of some testing tool which would run all the test cases automatically.

In summary, test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons or improper control flow. Basis path and loop testing are effective techniques for uncovering a broad array of path errors. Test cases that exercise control flow and data values just below, at or above maxima and minima are very likely to uncover errors.

CHAPTER 3
RESEARCH METHODOLOGY

Several white-box testing strategies have been proposed in the literature: basis path testing, condition testing, data flow testing, loop testing etc. Each strategy is based on generating a set of test cases for a given program that exercise a specific set of paths for the corresponding control flow graph.

It is the objective of this research to design and develop a software tool for basis path testing. Basis path testing methodology can be applied to a program's pseudocode. The basis path testing strategy is based on generating a set of maximal linearly independent paths from the control flow graph of a given program (each path starts with start node and ends with the stop node)[6].

The following series of steps can be followed to achieve the research goal.

a) Using program's pseudocode as a foundation, draw the corresponding control flow graph.
b) Determine the cyclomatic complexity of the resultant graph.
c) Determine a basis set of linearly independent paths.

The above hierarchy will be followed in the methodology phase and described each of them in detail.

3.1 Drawing Control Flow Graph (CFG)

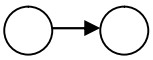
Basis path testing uses a control flow graph (or flow graph or program flow graph) to depict the logical control flow of the program. Control flow graph is a directed graph with a set of nodes (procedure and predicate nodes) and a set of edges. Each node represented by a circle represents one or more procedural statements. A node can be a:

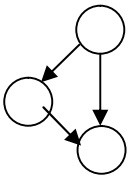
- **Procedure Node**: Procedure node is a node that represents one or more sequential statements. It is characterized by one edge emanating from it.
- **Predicate Node**: Predicate node represents a decision or condition.It is characterized by two or more edges emanating from it.

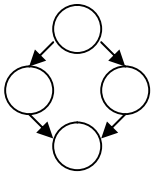
The edges (also called arcs or links) in the flow graph represent the flow of control in the program. An edge must terminate at a node, even if the node doesn't represent any procedural statements. Area bounded by edges and nodes are called regions. The area outside the graph is also counted as a region

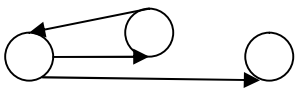
[31] suggests the following sequence of activities for drawing control flow graph: Pseudocode ⟶ Flow Chart ⟶ Control Flow Graph. Our program can construct the control flow graph directly from the pseudocode of a program. The intermediate step of drawing the flow chart as discussed by [31] can be eliminated as our program handles the logic of converting a flow chart into a control flow graph.

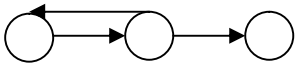
The CFG of a given program can be built from the basic/prime control flow graph notations. Programming constructs can be uniquely represented by the prime control flow graph notations. The prime control flow graph notations given by [31] and [34] are given in figure 3.1:

Sequence 

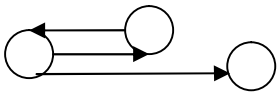
If-then 

If-then-else 

While 

Do-while 

For



Case

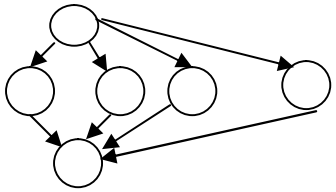


Figure 3.1 : Control flow graph notations

Each circle in the above figure represents one or more non-branching source code statements. Our program implements the following programming constructs: sequence (statements), if-then statement, if-then-else statement, while loop, do-while loop and for loop. The Boolean constructs (AND, OR and NOT) are also dealt with. It works on the following keywords in pseudocode i.e. if, then, else, elseif, endif, do, while, enddo, endwhile, for, endfor, and, or and end.

When compound conditions are encountered in a procedural design, the generation of control flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (e.g. logical AND, OR, NOT) are present in a conditional statement. Our program constructs a separate node for each of the compound conditions.

The symbols used for drawing control flow graph have been stated above. It is recommended to draw the control flow graph where the logical control structure of a module is complex.

3.1.1 Importance of Control Flow Graph

Graphs of programs have been used for many years in several areas of computer science, including software testing and compiler optimization. In these areas of application, any program or program fragment written in some procedural language can be translated into a control flow graph.Visualization of control flow graph of a program can provide better understanding of the programming process and is useful in helping programmers develop and maintain their programs[9][32]. It can improve testing and validation by making a formal link between algorithm implementation and specification. It can reduce maintenance by showing the complexity of the control flow. It can help establish norms and standards in quality

control programs. Furthermore, it can also provide better understanding and definition of the cohesion and coupling of software units. Some software metrics are based on the topology of such graphs.

Hence, control flow graph allows tracing of program paths more readily.

3.1.2 Example of Control Flow Graph

As an example, consider the pseudocode in figure 3.2, which implements Euclid's algorithm for finding greatest common divisors. Each statement is numbered for reference purposes and is represented as node in the accompanying control flow graph. The nodes are numbered from 1 to 14 and the edges are shown by the arrow line connecting the nodes.

```
// Assuming m and n both greater than 0 and return their greatest common divisor.
     Euclid (int m, int n)
1    int r;
2     if (n>m) then
3         r=m;
4         m=n;
5         n=r;
6     endif
7     r=mod (m, n)
8      while (r! = 0)
9          m=n;
10         n=r;
11         r=mod (m, n)
12    endwhile
13   return n;
14   end
```

Figure 3.2: Pseudocode for finding greatest common divisor

The corresponding control flow graph for the above pseudocode is shown in the figure 3.3.In the given control flow graph, node 2 represents the decision of the “if” statement with the true outcome at node 3 and the false outcome at the node 6. Similarly the decision of the

"while" loop is represented by node 8, and the upward flow of control to the next iteration is shown by the curved line from node 11 to node 8. Suppose the module is executed with parameters 4 and 2, as in "Euclid (4, 2)." Then execution begins at node 1, the beginning of the module, and proceeds to node 2, the decision node for the "if" statement. Since the test at node 2 is false, execution transfers directly to node 6 and proceeds to node 7. At node 7, the value of "r" is calculated to be 0, and execution proceeds to node 8, the decision node for the "while" statement. Since the test at node 8 is false, execution transfers out of the loop directly to node 12, then proceeds to node 13, returning the result of 2. Node 14 represents the end of the module.

.

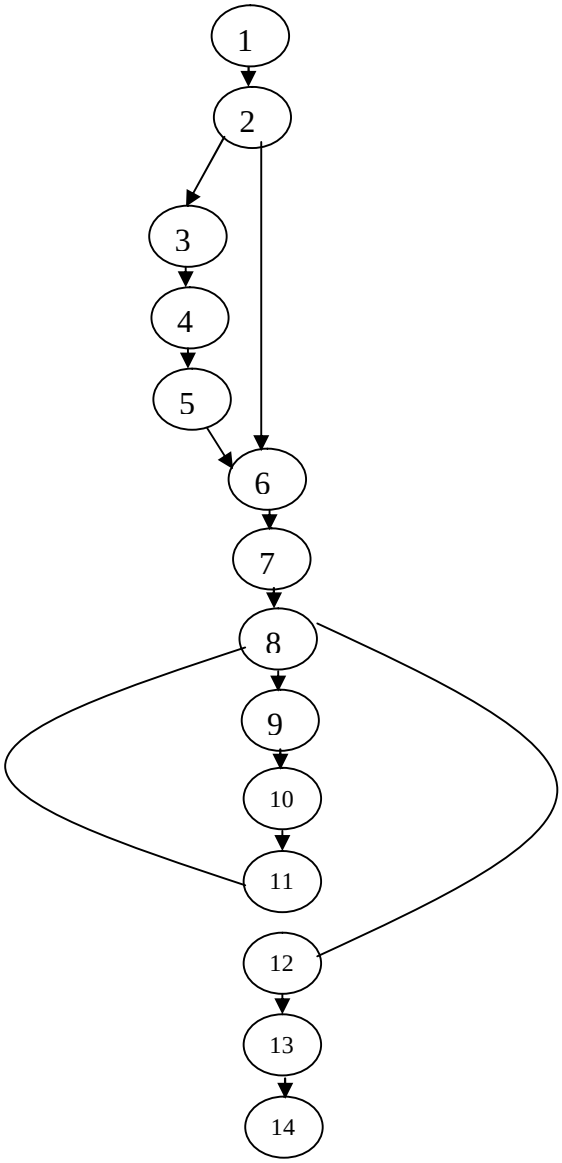


Figure 3.3: Control flow graph of pseudocode in figure 3.2

3.1.3 Matrix Representation of Control Flow Graph

Any program module may be represented by a graph called the flow graph, showing decision points and the possible logic paths through the program. Control flow graphs have been used extensively in static analysis of software and for the study of program-based structural test adequacy criteria [8].

For the purpose of finding independent paths, the control flow graph for a program module is represented as a labelled graph with its edges labelled with the symbols from an alphabet. The set of all paths from nodes i to j is indicated as C_{ij} . For obtaining the entire set of paths from the beginning to the end of the control flow graph, the control flow graph is treated as a finite automaton[22].Then the regular expression describing the strings accepted by the automaton gives the required set of paths. The graph has single initial and final node. Given in figure 3.4 are the flow graphs for common branching constructs.

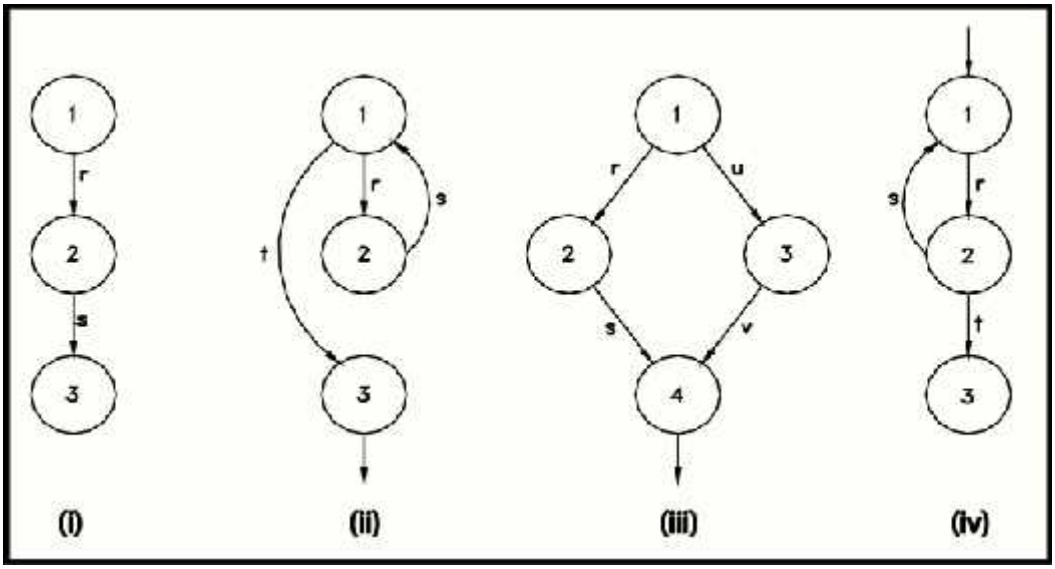


Figure 3.4: Labelled control flow graphs of different programming constructs [8]

In figure 3.4(i), we see that two sequential edges r and s have to be traversed to travel from node 1 to 3. Hence C_{13} = r**.**s where r.s = rs and '**.**' is the concatenation operator for strings. Similarly in figure 3.4(iii), the set of all paths from node 1 to 4 is {rs, uv}. Since there is a choice (or) between the paths we write the string C_{14} as, C_{14} = rs + uv. In figure 3.4(ii) and (iv), where there are more than one paths between any two nodes. For example, the paths from node 1 to 3 in figure 3.4(ii) can be written as zero or more occurrences of (rs) followed by t.

C_{13}= t + (rs) t + (rs) (rs) t + (rs) (rs) (rs) t +……..= $(rs)^{*}$t

Similarly, for figure 3.4(iv) C_{13} = (rs)*rt

The definition of operators '+', '.' and '*' are the same as in the context of regular expressions. The symbol 1 here has been used in place of ℵ (empty string) of regular expressions. This denotes the empty edge or a path of length 0 which has no symbols along it and which exists from every node to itself. Similarly, the symbol 0 is used in place of ←(null string) of regular expressions. In a flow graph, this represents an edge, which does not exist. We represent these control flow graphs as matrices. We write the adjacency matrix of a control flow graph except that for every edge between node i to j we write its label at the ij^{th} position in the matrix. Such a matrix is called a graph matrix. For example, element a_{13} = 0 in the graph matrix of figure 3.4(i) because there is no edge from node 1 to node 3 in the flow graph. The ii^{th} element of a graph matrix contains 1 for the empty edge because a path of length 0, which has no symbols along it, exists from every node to itself.
For example, the graph matrix of the graph in figure 3.4(i) is,

$$A = \begin{pmatrix} 1 & r & 0 \\ 0 & 1 & s \\ 0 & 0 & 1 \end{pmatrix}$$

3.2 Determining Cyclomatic Complexity

Cyclomatic complexity is software metric that provides a quantitative measure of the logical complexity of a program [31].When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program. This provides us with an upper bound for the number of tests that must be generated and executed to ensure that all statements in a given program have been executed at least once. Cyclomatic complexity provides us with an answer to the number of paths to test in a given program. Hence, it is related to the branch coverage criterion of white-box testing[17]. Cyclomatic complexity, according to McCabe, is computed in one of the three ways:

Method 1:

Cyclomatic complexity, V(G), for a control flow graph G, is defined as

$$V(G) = E - N + 2$$

Where E is the number of edges in the control flow graph and N is the number of control flow graph nodes.

Thus, for the CFG example in figure 3.3, the cyclomatic complexity is 3 (15 edges minus 14 nodes plus 2).

More importantly, the above method gives the number of independent paths through strongly connected directed graphs. A strongly connected graph is one in which each node can be reached from any other node by following directed edges in the graph. The cyclomatic number in graph theory is defined as e - n + 1. Program control flow graphs are not strongly connected, but they become strongly connected when a "virtual edge" is added connecting the exit node to the entry node. So, the cyclomatic complexity definition for program control flow graphs is derived from the cyclomatic number formula by simply adding one to represent the contribution of the virtual edge. This definition makes the cyclomatic complexity equal the number of independent paths through the standard control flow graph model, and avoids explicit mention of the virtual edge.

Method 2:

Cyclomatic complexity, V(G), for a control flow graph G, is also defined as

$$V(G) = P + 1$$

Where P is the number of predicate nodes contained in the control flow graph G.Predicate node represents a node with decision or condition. A simple condition is a logical expression without 'AND' or 'OR' operators. If the program includes compound conditions, which are logical expressions including 'AND' or 'OR' operators, then we count the number of simple conditions in the compound conditions when calculating the cyclomatic complexity.

Therefore, if there are six if-statement and a while loop and all conditional expressions are simple, the cyclomatic complexity is 8.If one conditional expression is a compound expression such as 'if A AND B OR C', then we count this as three simple conditions. The cyclomatic complexity is therefore 10.

For the CFG example in figure 3.3, node 2 represents the decision of the “if” statement with the true outcome at node 3 and the false outcome at the node 6.Hence node 2 represents the predicate node. Similarly, node 8 which is a “while” loop also represents the predicate node. So, the cyclomatic complexity is 2+1=3

The cyclomatic complexity, V(G), of the resultant control flow graph in this thesis is determined by applying the method 2 given above.

Method 3:

The number of closed regions of the control flow graph correspond to the cyclomatic complexity.

When the control flow graph is planar (no edges cross) and divides the plane into R regions (including the infinite region “outside” the graph), the simplified formula for cyclomatic complexity is just R. Thus, for a planar control flow graph, counting the regions gives a quick visual method for determining complexity. If the graph is not planar, the above formula does not work.

For the CFG example in figure 3.3, the number of closed regions is 3 (region outside the region is also counted). So, the cyclomatic complexity is 3.

3.2.1 Data Structure to Compute the Cyclomatic Complexity

Cyclomatic complexity can be also found out by using a data structure called graph matrix [31].A graph matrix, also called a connection matrix, is a square matrix whose size (i.e. number of rows and columns) is equal to the number of nodes on the control flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes. The control flow graph is used to develop a connection matrix. This matrix is then used to determine the cyclomatic complexity. The graph matrix is nothing more than a tabular representation of a control flow graph. However by adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing. The link weight is 1 if a connection exists between the nodes or 0 if a connection does not exist. The link weight also provides additional information about control flow.

To illustrate we use the simplest weighting to indicate connections (0 or 1). The graph matrix for control flow graph of figure 3.3 is as follow.

Node \ Connected to node	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Connections
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1-1=0
2	0	0	1	0	0	1	0	0	0	0	0	0	0	0	2-1=1
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1-1=0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1-1=0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1-1=0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1-1=0
7	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1-1=0
8	0	0	0	0	0	0	0	0	1	0	0	1	0	0	2-1=1
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1-1=0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1-1=0
11	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1-1=0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1-1=0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1-1=0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-
															Cyclomatic Complexity = 2+1=3

Table 1.1: Graph matrix to compute the cyclomatic complexity

Therefore performing the arithmetic shown to the right of the connection matrix provides us with a method for determining cyclomatic complexity. In the connection matrix, each row with two or more entries represents a predicate node.

The procedure for creating the connection matrix (graph matrix) from a control flow graph can be stated as:

1) A two dimensional array is declared. Its size is equal to the number of nodes in the control flow graph.

2) Construct the adjacency matrix $A_{ij} = 1$ if there is an edge from node i to node j and $A_{ij} = 0$ if there is no edge.

The complexity can be found out from this adjacency matrix using the procedure below.

1. For each row, find out the number of 1's in it. Subtract 1 from this. Save the result.
2. After doing step 1 on all rows, sum the results and add 1. This will give the cyclomatic complexity.

Linked list data structure is much more powerful than the graph matrix data structure proposed by [31] to construct control flow graph and to compute the cyclomatic complexity. Our program uses the linked list data structure to compute the cyclomatic complexity of control flow graph. Advantages of linked list over graph matrix data structure is discussed in detail in section 4.2.4

3.2.2 Relationship between Cyclomatic Complexity and Testing

There is a strong connection between complexity and testing. This is true in both an abstract and a concrete sense. In the abstract sense, complexity beyond a certain point defeats the human mind's ability to perform accurate manipulations, and errors result. In the concrete sense, numerous studies and general industry experience have shown that the cyclomatic complexity measure correlates with errors in software modules. The more complex a module is, the more likely it is to contain errors. Complex modules are more prone to error, harder to understand, harder to test, and are harder to modify.

Also, beyond a certain threshold of complexity, the likelihood that a module contains errors increases sharply. Given this information, many organizations limit the cyclomatic complexity of their software modules in an attempt to increase overall reliability. There are many good reasons to limit cyclomatic complexity. Complexity can be used to predict critical information about reliability and maintainability of software systems from the automatic analysis of source code. Complexity metrics also provide feedback during the software project to help control the design activity. During testing and maintenance, they provide detailed information about software modules to help pinpoint areas of potential instability. Complexity metric can be used as an indicator of how complex a module design is and this determines how easy it will be to test the specific module.

Many organizations have successfully implemented complexity limits as part of their software programs. The precise number to use as a limit, however, remains somewhat controversial. The original limit of 10 as proposed by McCabe appears to be a practical upper limit for module size, but limits as high as 15 have been used successfully as well. A number of people have recommended that a value of 10 is taken as the default level against which to compare cyclomatic complexity matrices of modules. Any module with a metric value higher than 10 is considered to be too complex and redesign or remodularization should be considered. When the cyclomatic complexity is 1-10, the program module is considered to be simple and without risk. If the value of CC (Cyclomatic Complexity) is from 11 to 20, the program module is more complex. For the value of CC from 21 to 50, the program module is complex and with high risk. Finally, for the value of CC above 50, the module is with very high risk and not testable.

In fact, cyclomatic complexity doesn't account for algorithmic complexity of a program but it is primarily measure of structural complexity of a program [4].

3.3 Determining a Basis Set of Linearly Independent Paths

A path through a program is a node and edge sequence from the starting node to a terminal node of the control flow graph of a program. A path through the program is just a sequence of instructions or statements that begins at a starting node and ends at a terminal node. A path may go through several processes or decisions, one or more times [5].

An independent path is a path through the program (from the entry node to the exit node) that introduces at least one new set of processing statements or a new condition [31]. When stated in terms of a control flow graph, an independent path moves along at least one edge that has not been traversed before the path is defined. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent. This is because; any path having a new node automatically implies that it has a new edge. A path that is subpath of another path is not considered to be a linearly independent path.

For example, a set of independent paths for the control flow graph illustrated in figure 3.3 are as follows:

Path 1: 1-2-6-7-8-12-13-14 (when n>m is F and r! =0 is F)

Path 2: 1-2-6-7-8-9-10-11-8-12-13-14 (when n>m is F, r!=0 is T and r!=0 is F)

Path 3: 1-2-3-4-5-6-7-8-12-13-14 (when n>m is Tand r! =0 is F)

Here, each new path introduces a new edge.

Path 4: 1-2-3-4-5-6-7-8-9-10-11-8-12-13-14 (when n>m is T, r! =0 is Tand r! =0 is F)

is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges. Path 4 can be expressed as a linear combination of the basis path 1 through path 3. For example path 4 is equal to path3-path 1+path 2.

A basis set of path is a set of paths, and every path in this set should satisfy the following three conditions [16]:

- Every path should be an independent path.
- Every path that does not contain in a basis set of path can be constructed by the linear combination among the paths in this set.
- All edges in a CFG should be covered by all paths in the basis set.

Paths 1, 2 and 3 constitute a basis set for the control flow graph in figure 3.3.

A set of test cases, written only for the set of all independent paths of a program is said to be a basis test set. This name comes from the fact that the paths taken by the test cases in the basis test set form a 'basis' for the set of all possible paths through the program. If test cases are designed to force the execution of paths in the basis set, then every statement in the program would be guaranteed to execute at least one time and every condition would be executed on its true and false sides. However, the basis set of paths is not unique [31]. A number of different basis sets can be derived for a given procedural design.

The value of cyclomatic complexity, V(G), provides the number of linearly independent paths through a program control flow graph. The predicate nodes are identified to aid in the derivation of cyclomatic complexity.

3.3.1 Methods to Generate the Basis Set of Paths

Basis set of paths is consisted of some of the program's paths. The paths that are to be tested must be generated or determined before testing. There has been a lot of research works dealt with the generation of basis set of paths for white- box testing. Different researchers have proposed different approaches to generate the basis set of paths. Some of these approaches have been discussed below.

3.3.1.1 Joseph Poole Approach

Joseph Poole [30] has discussed a basis set of path generation method using DFS (Depth First Search) algorithm in the CFG.The search starts at the source node and recursively descends down all possible outgoing paths. If the node visited has never been visited before, a default outgoing edge is picked, then the current path is split into new paths that traverse each outgoing edge, going down the default edge first. The default edge is any edge which is not a back edge or which later causes a node to have two incoming edges. For example, in the test condition of a pre-test loop, the default edge would be the edge which exits from the loop. If the edge that traversed the body of the loop was chosen, then a back edge from the last node in the body to the test condition node would have to be traversed later. If the node visited is a sink (no exit edges), then a path in the basis set has been found. Otherwise, the path traverses the default edge.

A pseudo-code outline of this method is as follows:

FindBasis(node)

if this node is a sink then print out this path as a solution

else if this node has not been visited before

mark the node as visited

label a default edge

FindBasis(destination of default edge)

for all other outgoing edges, FindBasis(destination of edge)

else

FindBasis (destination of default edge).

The flow graph in figure 3.5 can be used to demonstrate this method.

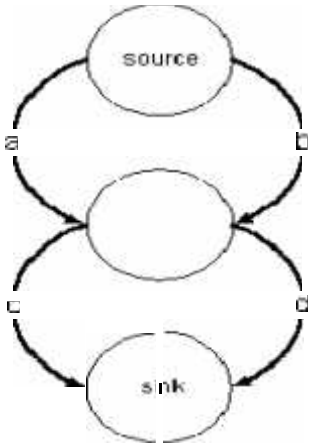


Figure 3.5: A control flow graph to illustrate Joseph approach to generate the basis set of paths [30].

A flow graph contains one source node and one sink node. A source node is a node which has no incoming edges, while a sink node is a node with no outgoing edges. The flowgraph from figure 3.5 can be used to demonstrate this method. The algorithm starts at the source node. The default edge can be either edge, let us pick a. Edge a is traversed and the intermediate node visited. Edge c is picked as the default edge for the intermediate node. Edge c is traversed and the final node visited. There are no outgoing edges, so path ac is added to the basis set. The algorithm has traversed the default edge of the intermediate node, so now d is traversed. The destination is again the sink, so ad is added to the basis set. The intermediate node has had all of its edges traversed; therefore b from the source node is traversed. The destination of b is the intermediate node which has been visited before. Therefore the default edge c leading to the sink is traversed and the path bc is added to the basis set. The source has had all outgoing edges traversed, so the algorithm terminates.

The limitation of this method is that the loop operation in a program is not taken into account. Another problem exist in this method is that it did not consider the method that how to choose the successor of a multiple successor node in a CFG to build a basis path during the construction of the basis set of path.

3.3.1.2 Bhattacherjee et al. Approach

Bhattacherjee et al.[7][8] have proposed a novel approach to determine the entire set of paths for any two nodes of a control flow graph of a program module by using the Symmetric

Algorithm. The elements of the graph matrix are manipulated by the Symmetric Algorithm to produce all paths between any two nodes. A simple heuristic approach is then applied to extract an independent set of paths from this set.

In this approach, the control flow graph of a program module is represented as a matrix and is called as a graph matrix. It has been also discussed in detail in the section 3.1.3. The elements of the graph matrix are manipulated by the Symmetric Algorithm to obtain matrix A*. The matrix A* so formed is called the regular matrix which have regular expression as their elements. The elements of matrix A* list out all paths between all pairs of nodes in the graph.

The notations used in the Symmetric Algorithm are:

a_{ii}	represents the ii^{th} element of the A matrix.
A	graph matrix of a labelled graph, of order n × n.
A_i	graph matrix with i^{th} row and column deleted, of order (n - 1) × (n - 1).
A_{ij}	graph matrix with i^{th} and j^{th} rows and columns deleted, of order (n - 2) × (n - 2).
r_i	i^{th} row of A with a_{ii} deleted, of order (n - 1).
s_i	i^{th} column of A with a_{ii} deleted, of order (n - 1).
r_{ji}	j^{th} row of A with a_{jj} and a_{ji} deleted, of order (n - 2).
s_{ji}	j^{th} column of A with a_{ij} and a_{jj} deleted, of order (n - 2).
p_{ij}	i^{th} row of A with a_{ij} and a_{ii} deleted, of order (n - 2).

Definitions of different terms that are used in the symmetric algorithm are as follows:

$\Leftarrow = (a_{ii} + r_i A_i^* s_i)^*$	where $\Leftarrow$ gives all the circuits starting from node i.
$\overset{i}{\Leftarrow}_j = (a_{jj} + r_{ji} A_{ij}^* s_{ji})^*$	where $\overset{i}{\Leftarrow}_j$ gives all those circuits starting from node j that do not pass through node i.
$b_{ij} = a_{ii} + p_{ij} A_{ij}^* s_{ji}$	where b_{ij} gives all paths from node i to node j excluding the circuits at i or j.

$c_{ii} = \Leftarrow_i$ where c_{ii} gives all paths from node i to itself, that is, all circuits starting from node i.

$c_{ij} = \Leftarrow_i b_{ij} \Leftarrow_j^i$ where c_{ij} gives all paths from node i to node j.

$A^* = [\ c_{ij}]$ where elements of matrix A^* list out all paths between all pairs of nodes in the graph.

The use of Symmetric Algorithm to determine independent paths is illustrated by using an example. A sample graph is shown in figure 3.6 below.

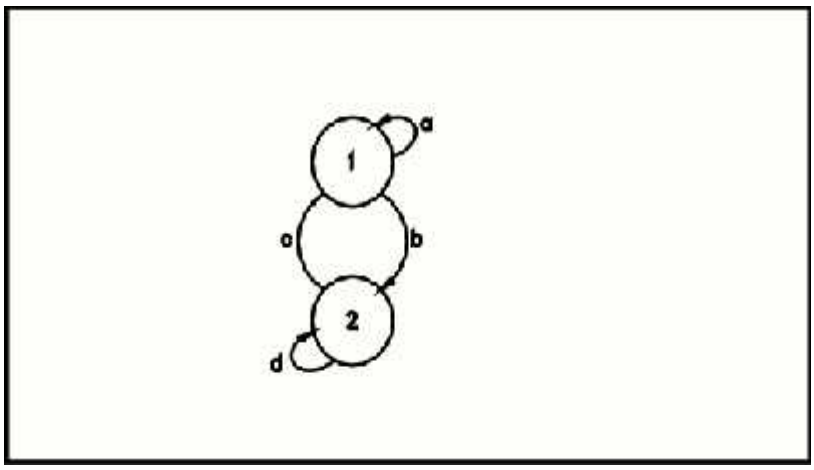


Figure 3.6: An example graph to illustrate the Symmetric Algorithm [7]

The graph matrix for the graph shown in figure 3.6 can be written as:

$$A = \begin{pmatrix} 1+a & b \\ c & 1+d \end{pmatrix}$$

By applying the Symmetric Algorithm, we get the following:

$\Leftarrow_1 = (a+ bd^*c)^*$, $\Leftarrow_2 = (d+ca^*b)^*$, $\Leftarrow_1^2 = a^*$, $\Leftarrow_2^1 = d^*$, $b_{12} = b$, $b_{21} = c$

$$A^* = \begin{pmatrix} (a+bd^*c)^* & (a+bd^*c)^*bd^* \\ (d+ca^*b)^*ca^* & (d+ca^*b)^* \end{pmatrix}$$

The entire set of paths from node 1 to 2 are given by $c_{12} = (a+bd^*c)^*bd^*$

Now, an algorithm is used to extract the set of independent paths from the entire set of paths obtained by the Symmetric Algorithm. The steps are as follows:

Setp1: Transform the regular expression by the following transformation, a* = 1+a. Hence the string expression (a + bd*c)*bd* becomes (1 + a + b (1 + d) c) b (1 + d).

Step 2: If the expression obtained is a product of sum of terms, go to step three otherwise, if a term happens to be a simple expression, call steps 3–5 recursively upon it and replace it with the set of strings so obtained.

Step 3: Strings representing independent paths are formed by picking up one symbol (or string of symbols) from every product and appending them together. Every time a new symbol is picked up from a product, it is replaced by a dashed symbol (implying that it has been used).

Step 4: If any non-dashed symbol remains in a product, pick it up, else pick up a dashed symbol. If no non-dashed symbols remain in trailing products, append all the non-dashed symbols of the current product to the previous string one by one. To complete the string pick any one dashed symbol from each of the trailing products and append.

Step 5: Repeat steps 3–4 for all products of sum of terms.

Step 6: The set of strings so obtained constitutes all the independent paths in the flow graph.
To get the basis set, choose any e-n+2 strings such that all symbols get included at least once.

Example: Consider the regular expression (a + bd*c)*bd*, which gives the entire set of paths from node 1 to node 2.

After applying step 1 to the regular expression (a + bd*c)*bd*, the string expression obtained is: (1+a+b(1+d)c)b(1+d). Now steps 3-5 are applied to the term b(1+d)c. First pass of step 3 gives sub-string b'(1'+d)c' and the first path string obtained is bc. Second pass of step 3 gives sub-string b'(1'+d')c' and second path string obtained is bdc.
Hence sub-string b(1+d)c gets replaced by bc+bdc.
Thus string expression becomes: (1+a+bc+bdc)b(1+d)
First pass of step 3 gives string (1'+a+bc+bdc)b'(1'+d) and first path string as:1.b.1=b

Second pass of step 3 gives string (1'+a+bc+bdc)b'(1'+d') and second path string: bd
Similarly other path strings thus obtained are: ab, bcb, abcb, bdcb
Hence the set obtained is {b, bd, ab, bcb,abcb, bdcb}
From this set, any (e-n+2) strings can be chosen such that all symbols get included at least once.

3.3.1.3 Guangmei et al. Approach

Guangmei et al. [16] have discussed a kind of basis set of paths generation method. It is built by searching the control flow graph of a program by depth first searching method. The generation of basis set of paths is based on the following two hypotheses.

Hypothesis 1: Every node other than the start node in the CFG can be reached from the start node, that is to say, there is at least one path from the start node to the nodes in the CFG other than the start node itself.
Hypothesis 2*:* There is at least one path from the nodes in a CFG other than the exit node to the exit node.

A Basis Set of Paths can be generated by the algorithm given below.

Step 1: Push the start node into a stack.
Step 2: If the stack is not empty, go to step 3 else go to step 7
Step 3: Try to find an unvisited edge whose tail node is the top element of the stack. If no edge is found, pop the top element of the stack from the stack and go to step 4. If an edge is found, go to step 4.
Step 4: Mark the edge with a visited flag. If the head of the edge is exit node, generate a basis path and the sub-paths from every multi-in-degree node to the exit node, go to step 3; else go to step 5.
Step 5: If the head node of this edge does not in the stack, and there is a sub-path from the node to the exit node, merge the sub-path in the stack with this sub-path and generate a new basis path go to step 3; else go to step 6.
Step 6: If the head node of this edge is in the stack, then generate a sub-path from the start node of the CFG by using the node in the stack and the head node of the new edge, else push the head node of this edge into the stack, go to step 2.

Step 7: For each the sub-path from the start node, find the corresponding sub-path to the exit node that the last node in the sub-path from the start node is the same as the first node in the sub-path to the exit node, then merge these two sub- paths to generate a basis path.

In order to reduce the searching procedure while building a basis paths , the sub-path from the multi-in- degree nodes to the end node of program and the sub-path that contains a loop is recorded during the construction of basis path.

3.3.1.4 Salloum and Salloum Approach

Salloum and Salloum [34] have presented an algorithm to generate a basis set of paths for a given control flow graph. For a given control flow graph, the algorithm generates a tree whose paths from the root to the leaves represent a basis set of paths. The tree is constructed by visiting every node and edge at least once as follows:

1. The start node is the root.

2. Repeat the following operation until every leaf node of the tree satisfies one of the three conditions: it is the stop node, has appeared twice on the same path in the tree (the path from the start to the leaf node), or has appeared as an interior node (non leaf node) of the tree.

For every node nd in the tree, create a new node of the tree for every node adjacent to nd (in the CFG) and draw a branch that connects nd to its adjacent node.

3. For every leaf node of the tree that is not the stop node, visit the nodes along any simple path (no cycle) from the leaf node to the stop node and create the necessary nodes and branches.

Our program makes use of the algorithm by [34] to generate a basis set of paths. It has been observed that the above algorithm generates a basis set of paths for a given control flow graph in O (max (n, e)) time complexity where n is the number of nodes and e is the number of edges in the control flow graph.

Salloum and Salloum approach to determine the basis set of independent paths is illustrated by using an example. A sample graph is shown in figure 3.7 below.

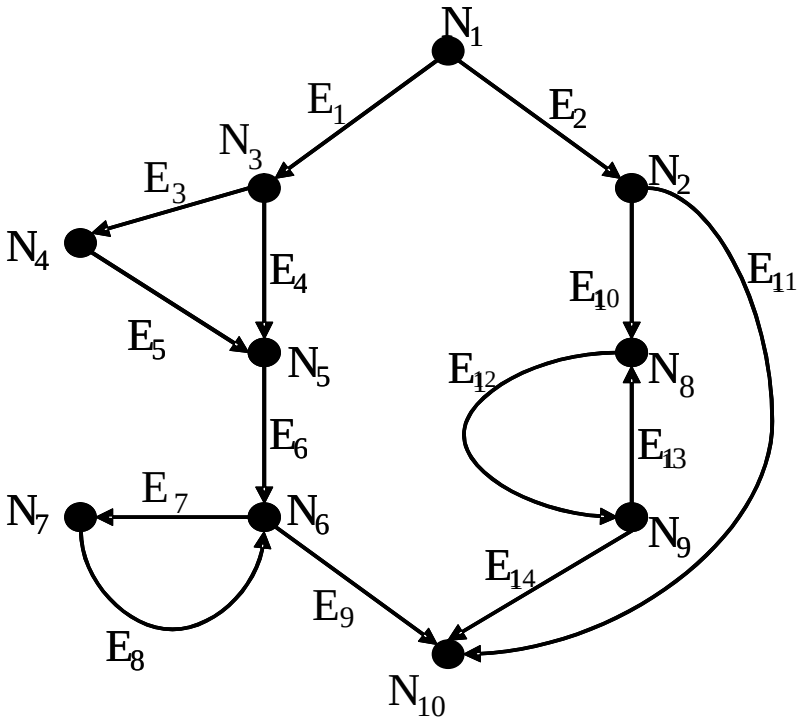


Figure 3.7: **A control flow graph to illustrate the Salloum and Salloum algorithm**

In the above graph, N_1 is the start node and N_{10} is the stop node. Figure 3.8 shows the tree generated by the algorithm.

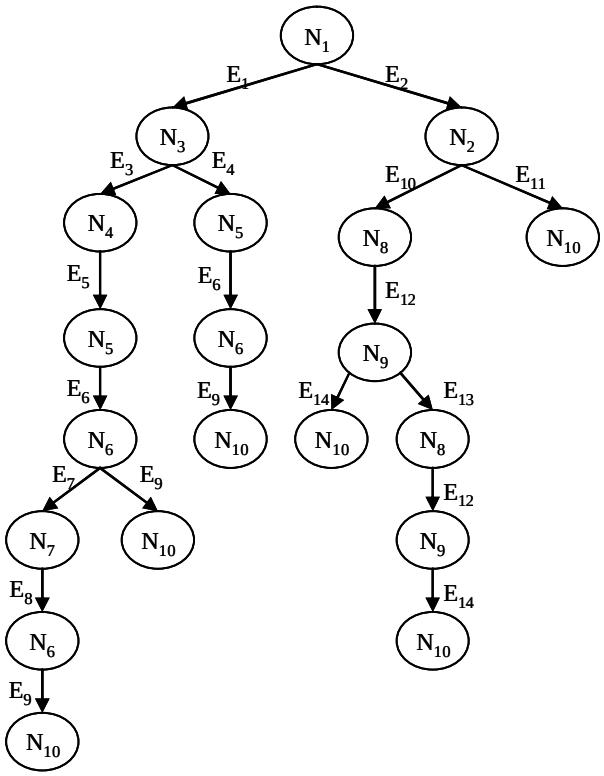


Figure 3.8: **A tree representing a basis set of paths generated by the algorithm for the CFG of figure 3.7**

A basis set of linearly independent paths for the CFG of figure 3.7 consists of the following:

$P_1 = E_1E_3E_5E_6E_7E_8E_9$

$P_2 = E_1E_3E_5E_6E_9$

$P_3 = E_1E_4E_6E_9$

$P_4 = E_2E_{10}E_{12}E_{14}$

$P_5 = E_2E_{10}E_{12}E_{13}E_{12}E_{14}$

$P_6 = E_2E_{11}$

A set of paths are linearly independent if no path in the set is a linear combination of any other paths in the sets, i.e. $P_1,P_2,\ldots P_n$ are linearly independent paths means if $\Sigma a_iP_i=0$, then all a_i's equal 0.

To verify the linearly independent property of these paths, the vector representation is used. Each path of a given CFG is represented by a vector of non-negative integers, where the i^{th} position in the vector is the number of occurrence of the edge i. The vector representations of the paths P_1 to P_6 are as follows:

P_1 = (10101111100000)

P_2 = (10101100100000)

P_3 = (10010100100000)

P_4 = (01000000010101)

P_5 = (01000000010211)

P_6 = (01000000001000)

Assume $\Sigma a_iP_i = 0$ to show that all a_i's=0. This equation implies the following identities:

$a_1 + a_2 + a_3=0$

$a_4+a_5+a_6=0$

$a_1+a_2=0$

$a_3=0$

$a_1+a_2=0$

$a_1+a_2+a_3=0$

$a_1=0$

$a_1=0$

$a_1+a_2+a_3=0$

$a_4+a_5=0$

$a_6=0$

$a_4+2a_5=0$

$a_5=0$

$a_4+a_5=0$

Obviously these equations imply all a_i's =0

Thus, this verifies that the paths generated by Salloum and Salloum algorithm are linearly independent paths.

3.3.1.5 Baseline Method

The Baseline method, proposed by McCabe [25], is a technique for identifying a basis set of test paths through the module being tested, equal in number to the cyclomatic complexity of the module. The word “baseline” comes from the first path, which is typically selected by the tester to represent the “baseline” functionality of the module. The Baseline method provides support for structured testing, since it gives a specific technique to identify an adequate test set rather than resorting to trial and error until the criterion is satisfied.

The first step is to pick a functional “baseline” path through the program. The selection of this baseline path is somewhat arbitrary. First, for every decision in the module, select an outcome with the shortest (in terms of number of other decisions encountered) path to the module exit. To generate the next path, change the outcome of the first decision along the baseline path while keeping the maximum number of other decision outcomes the same as the baseline path. That is, once the baseline path is “rejoined,” it should be followed to the module exit. Any decisions encountered that are not part of the baseline path may be taken arbitrarily, and, as with the baseline path, it is a good idea to follow a robust functional path subject to the constraint of varying just the first baseline decision. To generate the third path, begin again with the baseline but vary the second decision outcome rather than the first. When all of the decisions along the baseline have been flipped, proceed to the second path, flipping its new decisions as if it were the baseline. When every decision in the module has been flipped, the test path set is complete.

At each step, exactly one new basis edge is added, so the total number of paths generated is the cyclomatic complexity. It is sufficient to show that they are linearly independent to complete the proof that they are a basis set of paths. To see that, consider the path/edge matrix with the generated set of paths in order as the rows and the basis edges as the columns, with the columns ordered by the index of the path which first traversed each corresponding

edge. The matrix is then lower-triangular with all major diagonal entries equal to “1”, so all rows are linearly independent. Thus, the baseline method generates a basis set of paths.

CHAPTER 4

IMPLEMENTATION

4.1 Pseudocode of the Program

In chapter 3, different techniques for the generation of basis set of paths have been discussed. This chapter describes the implementation of the algorithm proposed in the previous chapter for the generation of basis set of paths. The pseudocode of the program implemented in this thesis is as follows:

```
1    While (data!= “end” or “END”)
2        Read one line at a time from the program;
3        If (data ends with semicolon)
4            Statement_procedure ();
5        If (data starts with “if” or “IF”)
6            IF_procedure ();
7        If (data starts with “then” or “THEN”)
8            THEN_procedure ();
9        If (data starts with “else” or “ELSE”)
10           ELSE_procedure ();
11       If (data starts with “else if” or “ELSE IF”)
12           ELSEIF_procedure ();
13       If (data starts with “while” or “WHILE”)
14           WHILE_procedure ();
15       If (data starts with “do” or “DO”)
16           DO_procedure ();
17       If (data starts with “for” or “FOR”)
18           FOR_procedure ();
19       If (data == “endif” or “ENDIF”)
20           ENDIF_procedure ();
21       If (data == “enddo” or “ENDDO”)
22           ENDDO_procedure ();
23       If (data == “endwhile” or “ENDWHILE”)
24           ENDWHILE_procedure ();
```

25 If (data == “endfor” or “ENDFOR”)

26 ENDFOR_procedure ();

27 endwhile

28 If (data == “end” or “END”)

29 END_procedure ();

30 Showtree();

31 Display_nodes ();

32 Cyclomatic_complexity ();

33 Testpath ();

Statement_procedure ()

1 Store the variable type, variable name and variable value in a lookup table

2 If data read immediately before this ended with semicolon, then concatenate new data with data read immediately before.

3 Make a new node in linked list

4 Store the data in the node

IF_procedure ()

1 Save “if” substring;

2 Skip any leading whitespaces

3 string = index of “if” + 2 to len – 1;

4 Find_and_or (string);

THEN_procedure ()

1 Save “then” substring;

2 Skip any leading whitespaces

3 Statement_procedure (); //gathers statements that end with semicolon in one node

ELSE_procedure ()

1 Save “else” substring;

2 Skip any leading whitespaces

3 Statement_procedure (); //gathers statements that end with semicolon in one node

ELSEIF_procedure ()

1 Save “elseif” substring;

2 Skip any leading whitespaces

3 string = index of “elseif” + 6 to len – 1;

4 Find_and_or (string); // involves the same steps as IF_procedure ()

WHILE_procedure ()

1 Save “while” substring;

2 Skip any leading whitespaces

3 string = index of “while” + 5 to len – 1;

4 Find_and_or (string); // involves the same steps as IF_procedure ()

DO_procedure ()

1 Save “do” substring;

2 Skip any leading whitespaces

3 Statement_procedure (); //gathers statements that end with semicolon in one node

FOR_procedure ()

1 Save “for” substring;

2 Skip any leading whitespaces

3 Separate for statement in 3 parts by looking for semicolons

4 Store variables in lookup array (type, name, value)

5 Make a node for the whole for statement

ENDIF_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “if”.

ENDDO_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “do”.

ENDWHILE_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “while”.

ENDFOR_procedure ()

1 Create the link in the linked list by traversing linked list backwards to look for first “for”.

END_procedure ()

1 Stop reading input data

Find_and_or (data)

1 index1 = look for index of “and” in data //returns first occurrence of and

2 index2 = look for index of “or” in data //returns first occurrence of or

3 case 1: if index1 and index2 contain nothing: //and & or not found

4 make a new node in linked list

```
5               save data in the new node
6       case 2: if only index1 returns a value:          //and found
7               make a new node in linked list
8               skip leading whitespaces
9               string1 = store data substring from index 0 to index1-1 in the new node
10              string2 = store substring index1+4 to len-1
11              Find_and_or (string2);
12      case 3: If only index2 returns a value:          //or found
13              make a new node in linked list
14              skip leading whitespaces
15              string1 = store data substring from index 0 to index2 –1 in the new node
16              string2 = store substring index2 + 3 to len – 1
17              Find_and_or (string2);
18      case 4: If both index1 and index2 return a value:  //both and & or found
19              make a new node in the linked list
20              index3 = min (index1, index2);
21              string1 = store data substring from index 0 to index3 – 1 in new node
22              string2 = store substring index3 + 3 to len – 1
23              Find_and_or (string2);
```

Showtree ()

```
1       Node current = head;
2       Node p = head;
3       while (current != null)
4               while (p != null)
5                       if (p.node_link1 != 0 or -1)
6                               Node r = head;
7                               while (r != null)
8                                       if (p.node_link1 == r.node_number)
9                                       draw arrowed line between the nodes p and r
10                                      r = r.getNext();
11                              endwhile
12                      endif
13                      if (p.node_link2 != 0 or –1)
14                              Node r = head;
```

```
15                          while (r != null)
16                                  if (p.node_link2 == r.node_number)
17                                  draw arrowed line between the nodes p and r
18                                  r = r.getNext();
19                          endwhile
20                  endif
21          p = p.getNext();
22          endwhile
23          write the node_number
24          curent = current.getNext();
25  endwhile
```

Display_nodes ()

```
1   Node current = head;
2   while (current!= null)
3           output node_number, node_link1, node_link2, marker, text;
4           current = current.getNext();
5   endwhile
```

Cyclomatic_complexity ()

```
1   Node current = head;
2   node_count=0;
3   while (current! = null)
4           if (current.nodelink1! =0 && current.nodelink2! =0)
5                   node_count++;
6           endif
7           current = current.getNext();
8   endwhile
9   cyclomatic_complexity=node_count+1;
10  print cyclomatic _complexity;
```

Testpaths()

```
1   Node current = head;
2   String t = String.valueOf(current.node_number);
```

```
3       Paths(current,t);
```

Paths(Node n, String t)

```
1       Node current = null;
2       if(n.node_number == listCount)
3               print t;
4       else
5               if(n.node_link1 != 0 && n.node_link1 != -1)
6                       current = getNode(n.node_link1);
7                       String rough = String.valueOf(current.node_number);
8                       String t1 = t.concat(rough);
9                       if(current != null && count < 3)
10                              paths(current, t1);
11              endif
12              if(n.node_link2 != 0 && n.node_link2 != -1)
13                      current = getNode(n.node_link1);
14                      String rough = String.valueOf(current.node_number);
15                      String t2 = t.concat(rough);
16                      if(current != null && count < 3)
17                              paths(current, t2);
18              endif
19      endif
```

4.2 Program Facts

Some facts related to the thesis program have been discussed in this section.

4.2.1 Control Flow Graph

Control flow graph construction is straightforward and easy to apply to small programs. Larger programs need a software tool to compute and draw the control flow graph. So, focus of our program is on constructing control flow graph from the pseudocode of a program. Our program accommodates the following programming constructs: if-then, if-then-else, do-while, while and for loop. It works on the following keywords in pseudocode i.e. if, then, else, elseif, endif, do, while, enddo, endwhile, for, endfor, and, or and end.

4.2.2 Syntax Errors

We haven't limited our program to any specific language hence no syntax errors are considered.

4.2.3 Semantic Errors

We have used pseudocode of a program in order to avoid semantic errors. Only the logical flow of function is looked at.

4.2.4 Data Structure

The procedure for deriving the control flow graph and even determining a set of basis paths can be automated. To develop a software tool that assists in basis path testing, a data structure called a linked list is used. Linked list is used to store both the predicate and procedure nodes created. Linked list can store many parameters and mathematical algorithms when applied to linked list can be used to automate the generation of the basis set of paths.

Linked list data structure is much more powerful than the graph matrix data structure proposed by [31] to construct control flow graphs and to compute the cyclomatic complexity.

Linked list is one of the fundamental data structure. It consists of a sequence of nodes, each containing arbitrary data fields of same type and one or two references (or pointers) pointing to the next and/or previous nodes. Linked list permits insertion and removal of nodes at any point in the list in constant time, but does not allow random access. On the other hand, a graph matrix is a square matrix whose size (number of rows and columns) is equal to the number of nodes in the flow graph. The matrix entries correspond to connections (an edge) between nodes. Matrix entries can be accessed randomly by specifying the index. Graph matrix is nothing more than a tabular representation of a flow graph.

The principle benefit of linked list over a graph structure is that the order of linked items may be different to the order that data items are stored in memory or on disk, allowing the list of items to be traversed in a different order. Elements can be inserted into linked lists indefinitely, while a matrix will eventually either fill up or need to be resized, an expensive operation that may not even be possible if memory is fragmented. Similarly, a matrix from

which many elements are removed may become wastefully empty or need to be made smaller.

Linked list structure is often preferred to represent sparse graphs, that is, graphs with few edges, as it has smaller memory requirements and it does not use any space to represent edges which are not present. Graph structures on the other hand provide faster access for some applications but can consume huge amounts of memory if the graph is very large. In general, linked list is beneficial for a dynamic collection, where elements are frequently being added and deleted, and the location of new elements added to the list is significant.

The complexity to insert/delete data nodes in linked list is O (1). The complexity to index elements in linked list is O (n).

The control flow graph in our program is not a fully connected graph. Hence, it justifies the use of a linked list. Statements in the pseudocode are stored systematically in the nodes of the linked list. Every node in the linked list has the following data members: text, marker, node number, next node, node link1 and node link2. Text and marker are stored in string formats. Text field holds the statements entered by the user whereas marker identifies the keyword (e.g. if, while, do etc.) used in the corresponding text field. The text field is parsed to separate the definition of variables in the program. The type, name and value of variables are stored in a lookup table. Linked list is updated as pseudocode statements are read from the file line by line. This is unlike the static graph matrix data structure that requires size declaration at the beginning. Linked list allows efficient traversal of data nodes when constructing the control flow graph in the shape of a tree.

4.2.5 Programming Language

The programming has been accomplished by using Visual C++ codes. Microsoft Visual C++ 6.0 version is used for this purpose.

4.2.6 Display

Our application reads the pseudocode of the program stored in the file with one line at a time. Data read from the file is then passed to the main program. The main program analyzes the data, breaks the data based on the language keywords, stores the fragmented data in the linked list and constructs control flow graph, calculate the cyclomatic complexity and generates the basis set of paths at the end.

Our application displays the program nodes (both procedure nodes and predicate nodes) in the form of number. The nodes are numbered according to the node number in the linked list. The node and its linked nodes are shown with an arrow head. The user can select the function by highlighting the code for the specific function and the cyclomatic complexity of that particular function is displayed. If the cyclomatic complexity is greater than 10, then the value is displayed in red, otherwise it is displayed in black.

Finally, our application generates the test paths of the given program by visiting the nodes of the tree from the root to the leaves.

Thus our application accepts pseudocode of the program as input, generates control flow graph in the form of a tree diagram, compute its cyclomatic complexity and provides program feedback in the form of test paths.

4.2.7 Program Assumptions

Some assumptions used in the program are as follows:

- All small or all capital keywords
- A simple statement ends with a semicolon.
- A simple statement can have at most one next link.
- A conditional statement can have at most 2 next links.
- Program operates on keywords of the language – if, for, and, etc.

CHAPTER 5

EXPERIMENTS, RESULTS AND EVALUATION

5.1 Experiments Design

Experimental results have been obtained by employing our tool for the unit testing of the following programs: compare, sum of squares, binary search, bubble sort, selection sort, quick sort's partition algorithm and insertion sort. We have selected seven test programs as SUTs(Software Under Test) for experimentations. Each SUT poses special characteristics which we would like to investigate the performance of our software tool against. All these programs have been taken from [36] for the experimental purpose. Each experiment is comprised of pseudocode of the program under test as an input. Then, our software tool constructs the corresponding CFG of the pseudocode, computes the cyclomatic complexity and finally generates the basis set of independent paths of the CFG.

5.2 Experimental Results

Experimental results obtained from the program under test include:

5.2.1 Compare

```
Void Compare ()
1    int i = 10;
2    int j = 100;
3    if i < j
4         then i++;
5    endif
6    end
```

Digraph G

{

1

1 ->2

2

2->3

2 ->4

3

3->4

4

4 ->5

5

}

The nodes in the tree are:

{1, 2, int i = 10;int j = 100;}

{2, 3, 4, if, i < j}

{3, 4, then, then i++;}

{4, 5, endif, endif}

{5, end, end}

Cyclomatic Complexity: 2

Test Paths:

1 2 4 5

1 2 3 4 5

5.2.2 Sum of Squares

```
Void SumSquares ()
1    int n = 10;
2    int partialsum = 0;
3    int i = 1;
4    int temp = 0;
5    while i <= n
6         temp = i * i;
7         partialsum = partialsum + temp;
8         i++;
9    endwhile
10   end
```

Digraph G

{

1

1 ->2

2

2->3

2->4

3

3->2

4

4->5

5

}

The nodes in the tree are:

{1, 2, int n = 0;int partialsum = 0;int i = 1;in temp = 0;}

{2, 3, 4, while, i <= n}

{3, 2, temp = i * i; partialsum = partialsum + temp; i++;}

{4, 5, endwhile, endwhile}

{5, end, end}

Cyclomatic Complexity: 2

Test Paths:

1 2 4 5

1 2 3 2 4 5

5.2.3 Binary Search

```
Void BinarySearch ()
1    int keys [10];
```

```
2    int low = 0;
3    int high = 9;
4    int middle = 0;
5    int x = 5;
6    while low < high
7         middle = (low + high) /2;
8         if x > keys[middle]
9              then low = middle + 1;
10             else high = middle;
11        endif
12   endwhile
13   keys[low] = x;
14   end
```

Digraph G
{
1
1 ->2
2
2->3
2->8
3
3->4
4
4->5
4->6
5
5->7
6
6->7
7
7->2
8
8->9
9

9->10

10

}

The nodes in the tree are:

{1, 2, int keys [10];int low = 0;int high = 9;int middle = 0;int x = 5;}

{2, 3, 8, while, low < high}

{3, 4, middle = low + high / 2;}

{4, 5, 6, if, x > keys [middle]}

{5, 7, then, then low = middle + 1;}

{6, 7, else, else high = middle;}

{7, 2, endif, endif}

{8, 9, endwhile, endwhile}

{9, 10, keys [low] = x;}

{10, end, end}

Cyclomatic Complexity: 3

Test Paths:

1 2 8 9 10

1 2 3 4 5 7 2 8 9 10

1 2 3 4 6 7 2 8 9 10

5.2.4 Bubble Sort

```
Void BubbleSort ()
1    int array [10];
2    int temp = 0;
3    boolean notdone = 0;
4    while notdone == 0
5         for int i = 0; i < 10; i++
6              if array [i] > array [i + 1]
7                   then temp = array [i];
8                   array [i] = array [i + 1];
9                   array [i + 1] = temp;
```

```
10                  notdone = 1;
11              endif
12          endfor
13      endwhile
14      end
```

Digraph G

{

1

1->2

2

2->3

2-> 8

3

3 ->4

3 ->7

4

4-> 5

4-> 6

5

5-> 6

6

6-> 3

7

7-> 2

8

8 ->9

9

}

The nodes in the tree are:

{1, 2, int array [10];int temp = 0;boolean notdone = 0;}

{2, 3, 8, while, notdone == 0}

{3, 4, 7, for, i < 10 ; int i = 0; i ++}

{4, 5, 6, if, array > array + 1}

{5, 6, then, then temp = array; array = array + 1; array + 1 = temp; notdone = 1 ;}

{6, 3, endif, endif}

{7, 2, endfor, endfor}

{8, 9, endwhile, endwhile}

{9, end, end}

Cyclomatic Complexity: 4

Test Paths:

1 2 8 9

1 2 3 7 2 8 9

1 2 3 4 6 3 7 2 8 9

1 2 3 4 5 6 3 7 2 8 9

5.2.5 Selection Sort

```
Void SelectionSort ()
1    int array [10];
2    int i = 9;
3    int j = 0;
4    int temp = 0;
5    while i > 0
6         j = i;
7         for int k = 0; k < i; k++
8              if array[k] > array[j]
9                   then j = k;
10
11             endif
12        endfor
13        temp = array [i];
14        array [i] = array [j]
15        array [j] = temp;
16        i--;
17   endwhile
```

```
18    end
```

Digraph G
{
	1
	1->2
	2
	2->3
	2->10
	3
	3->4
	4
	4->5
	4->8
	5
	5->6
	5->7
	6
	6->7
	7
	7->4
	8
	8->9
	9
	9->2
	10
	10->11
	11
}

The nodes in the tree are:

{1, 2, int array [10]; int i = 9; int j = 0; int temp = 0 ;}

{2, 3, 10, while, i > 0}

{3, 4,j = i;}

{4, 5, 8, for, int k=0 ; k <10; k ++}

{5, 6, 7, if, array [k] > array [j]}

{6, 7, then, then j = k;}

{7, 4, endif, endif}

{8, 9, endfor, endfor}

{9, 2, temp = array [i];array [i] = array [j];array [j] = temp; i--;}

{10, 11, endwhile, endwhile}

{11, end, end}

Cyclomatic Complexity: 4

Test Paths:

1 2 10 11

1 2 3 4 8 9 2 10 11

1 2 3 4 5 7 4 8 9 2 10 11

1 2 3 4 5 6 7 4 8 9 2 10 11

5.2.6 Quick Sort's Partition Algorithm

```
Void Partition ()
1    int array [10];
2    int i;
3    int j;
4    int pivot = 0;
5    int temp = 0;
6    int middle = 0;
7    int p = 0;
8    middle = i + j / 2;
9    pivot = array [middle];
10   array [middle] = array [i];
11   array [i] = pivot;
12   p = i;
13   for int k = i + 1; k <= j; k++
14        if array [k] > pivot
15             then temp = array [++p];
```

```
16              array [p] = array [k];
17              array [k] = temp;
18          endif
19      endfor
20      temp = array [i];
21      array [i] = array [p];
22      array [p] = temp;
23      end
```

Digraph G
{
1
1->2
2
2->3
2->6
3
3->4
3->5
4
4->5
5
5->2
6
6->7
7
7->8
8
}

The nodes in the tree are:

{1, 2, int array [10]; int i; int j; int pivot = 0; int temp = 0; int middle = 0; int p = 0; middle= i+j/2; pivot = array [middle]; array [middle] = array [i]; array [i] = pivot; p = i ;}

{2, 3, 6, for, int k = i + 1; k <= j; k++}

{3, 4, 5, if, array [k] > pivot}

{4, 5, then, then temp = array [++p]; array [p] = array [k]; array [k] = temp ;}

{5, 2, endif, endif}

{6, 7, endfor, endfor}

{7, 8, temp = array [i]; array [i] = array [p]; array [p] = temp ;}

{8, end, end}

Cyclomatic Complexity: 3

Test Paths:

1 2 6 7 8

1 2 3 4 5 2 6 7 8

1 2 3 5 2 6 7 8

5.2.7 Insertion Sort

```
Void InsertionSort ()
1    int array [10];
2    int j = 0;
3    int k = 0;
4    boolean notfinished = 0;
5    for int i = 0; i < 10; i++
6         k = array [i];
7         j = i;
8         if array [j - 1] > k
9              then notfinished = 1;
10             else notfinished = 0;
11        endif
12        while notfinished == 1
13             array [j] = array [j - 1];
14             j--;
15             if j > 0
16                  if array [j - 1] > k
17                       then notfinished = 1;
```

```
18                      else  notfinished = 0;
19                  endif
20              else notfinished = 0;
21              endif
22          endwhile
23          array [j] = k;
24    endfor
25    end
```

Digraph G
{

1

1->2

2

2->3

2->19

3

3->4

4

4->5

4->6

5

5->7

6

6->7

7

7->8

8

8->9

8->17

9

9->10

10

10->11

10->15

11

11->12

11->13

12

12->14

13

13->14

14

14->16

15

15->16

16

16->8

17

17->18

18

18->2

19

19->20

20

}

The nodes in the tree are:

{1, 2, int array [10];int k = 0;int j = 0;boolean notfinished = 0;}

{2, 3, 19, for, int i = 0; i < 10; i ++}

{3, 4, k = array [i];j = i;}

{4, 5, 6, if, array [j - 1] > k}

{5, 7, then, then notfinished = 1;}

{6, 7, else, else notfinished = 0;}

{7, 8, endif, endif}

{8, 9, 17, while, notfinished = = 1}

{9, 10, array [j] = array [j - 1]; j-- ;}

{10, 11, 15, if, j > 0}

{11, 12, 13, if, array [j - 1] > k}

{12, 14, then, then notfinished = 1;}

{13, 14, else, else notfinished = 0;}

{14, 16, endif, endif}

{15, 16, else, else notfinished = 0;}

{16, 8, endif, endif}

{17, 18, endwhile, endwhile}

{18, 2, array[j] = k; }

{19, 20, endfor, endfor }

{20, end, end}

Cyclomatic Complexity: 6

Test Paths:

1 2 18 19 20

1 2 3 4 5 7 8 9 10 11 12 14 15 16 8 17 18 19 20

1 2 3 4 7 8 9 10 11 12 14 15 16 8 17 18 19 20

1 2 3 4 5 7 8 17 18 19 20

1 2 3 4 5 7 8 9 10 11 13 14 15 16 8 17 18 19 20

1 2 3 4 5 7 8 9 10 15 16 8 17 18 19 20

5.3 Evaluation of Results

The outcome of the research has been evaluated by verifying the CFG, cyclomatic complexity and the basis set of paths generated by the tool against the result obtained manually.Executed results from the tool have been compared with the expected results (manually) to ensure whether the tool gives the satisfactory result or not.

We found that the algorithm implemented in this thesis gives acceptable results for drawing the control flow graph, computation of cyclomatic complexity and generation of basis set of paths from the pseudocode of a given program. It has been observed that control flow graph, from the pseudocode of a given program is generated in a reasonable time. The algorithm implemented in the tool for generating a basis set of paths for a given control flow graph by using approach given by [34] yields results in linear time i.e. O(max(n,e)) time where n is the number of nodes and e is the number of edges in the control flow graph.

CHAPTER 6
CONCLUSION AND FURTHER RECOMMENDATION

6.1 Conclusion

Testing involves the identification of a limited number of test cases out of nearly unlimited number of possibilities. One of the biggest problems in automation of software testing is how to determine test cases. Identifying test cases typically requires the generation of basis set of independent paths.

Unit testing of conventional or object-oriented software makes heavy use of white-box testing techniques, specifically basis path testing. Basis path testing uses the control flow graph of a program to generate a set of independent control flow paths.
This thesis focuses on generating basis set of paths for basis path testing. In this thesis, the basic steps followed are:

a) Control flow graph construction
b) Computation of cyclomatic complexity
c) Basis set of paths selection

Experiments on our tool show that the algorithms implemented for control flow graph construction, computation of cycloamtic complexity and basis set of paths selection for a given program yield results in a reasonable time. The control flow graph contains both procedure and predicate nodes and shows all the pertinent control flow information.

A software tool has been implemented and the power of the approach demonstrated. The solution presented is a simple one, and can be applied to any software following procedural programming approach i.e. conventional software.

This work deals strictly with the internal control flow of a program, which constitutes only a part of the information contained in the pseudocode. Additionally, control flow graph obscures important information present in the code, specifically feasible and infeasible paths. The testing of a set of independent paths may not be sufficient in many cases.

6.2 Further Recommendation

The techniques presented in this thesis are an initial step towards a more extensive test case generation application. Future work would be to generate the test cases for testing the conventional software and to look into the feasibility of modifying the developed tool to generate test cases for object-oriented programs as well. In this scenario, the control flow graph can be modified to accommodate several functions in one class. The predicate nodes in the control flow graph will be converted to function nodes (function calls). When this strategy is applied, then it becomes class level testing (OO paradigm) instead of unit level testing (procedural programming). So, the algorithm implemented in this thesis can be further improved and extended to apply to class level testing for object-oriented software testing.

This tool can be extended to incorporate data flow definition-use (DU) testing. Our program stores the definition of variables present in the pseudocode in a lookup table. This together with the subsequent use of variables in the control flow graph's predicate nodes can be applied to select test paths for data flow testing.

References:

[1] B. T. Abreu, E. Martins, and F.L.Sousa, *Automatic Test Data Generation for Path Testing using a New Stochastic Algorithm*, 2004
http://www.sbbd-sbes2005.ufu.br/arquivos/16-%209523.pdf

[2] ANSI/IEEE STD 610.12-1990, *Glossary of Software Engineering Terminology*, The Institute of Electrical and Electronics Engineers, Inc., February 1991

[3] A.L.Baker, J.W.Howatt and J.M. Bieman, *Criteria for finite set of paths that characterize control flow*, Proceedings of 19th Annual Hawaii International Conference on System Sciences, 1986, pp.158–163.

[4] A.Behforooz and F.J. Hudson, Software Engineering Fundamentals (Oxford University Press, Special Edition, 2004, New York)

[5] B. Beizer, Software Testing Techniques (Van Nostrand Rheinhold, Second Edition, 1990 New York)

[6] A. Bertolino and M. Marre, *Automatic Generation of Path Covers Based on the Control flow analysis of computer Programs*, IEEE Transaction on Software on software Engineering, Vol.20, No.12, 1994, pp. 885-899

[7] V.Bhattacherjee, D. Suri, and P.K. Mahanti, *Application of regular matrix theory to software testing*, European Journal of Scientific Research, Vol.12, No.1, 2005, pp. 60-70

[8] V.Bhattacherjee, D. Suri, and P.K. Mahanti , *Software testing: a graph theoretic approach*, Int. J. Information and Communication Technology, Vol. 1, No. 1, 2007, pp.14–25

[9] J.R. Bint and R. Site, *Optimizing Testing Efficiency with Error Prone Path Identification and Genetic Algorithms*, Australian Software Engineering Conference (ASWEC'04), (13-16) April 2004, Melbourne Australia, pp. 106-115

[10] P.M.S. Bueno and M. Jino, *Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data*, Proceedings of the 15th IEEE International Conference

on Automated Software Engineering (ASE '00),(11-15) September 2000, Grenoble France, pp. 209-218

[11] H. Y.Chen, T. Y.Chen and T. H. Tse, *TACCLE - a methodology for object-oriented software testing at the class and cluster levels*, ACM Transactions on Software Engineering and Methodology (TOSEM),Vol. 10, No.1, January 2001, pp. 56-109.

[12] G. A. Cheston and J. P.Tremblay, Data Structures and Software Development in an Object-Oriented Domain (Prentice Hall, Java Edition, 2002, New Delhi)

[13] W. H Deason., D. B. Brown, K.H. Chang and J. H. Cross, *A rule-based software test data generator*, IEEE Transactions on Knowledge and Data Engineering, Vol. 3, No. 1, March 1991, pp. 108-117.

[14] D. R. Graham, *Software testing tools: A new classification scheme,* Journal of Software Testing, Verification and Reliability, Vol. 1, No. 2, 1992, pp. 18-34

[15] H.G.Gross and A.Seesing, *A Genetic Programming Approach to Automated Test Generation for Object-Oriented Software*, 2006
http://www.st.ewi.tudelft.nl/~gross/Publications/Seesing_2006.pdf

[16] Z. Guangmei, C.Rui, L. Xiaowei and H.Congying, *The Automatic Generation of Basis Set of Path for Path Testing*, Proceedings of the 14th Asian Test Symposium (ATS'05),(18-21)December 2005, pp.46-51

[17] N.Gupta, A.P. Mathur and M.L.Soffa, *Generating Test Data For Branch Coverage* Proceedings of the 15th IEEE International Conference on Automated Software,(11-15) September 2000, Grenoble France

[18] J.E. Hopcroft, R.Motwani and J.D. Ullman, Introduction to Automata Theory,Languages and Computation (Pearson Education, Second Edition 2001, New Delhi)

[19] J.C.Huang, *An Approach to Program Testing*, ACM Computing Surveys, Vol. 7, No. 3, September 1975.

[20] N. Juristo, A. M. Moreno and W. Strigel, S*oftware testing practices in industry*, IEEE Software, Vol 23, No.4,August 2006, pp.19-21

[21] B.Korel, *Automated Software Test Data Generation*, IEEE Transactions on Software Engineering, Vol. 16, No. 8, August 1990, pp. 870-879

[22] R.Lämmel and J. Harm, *Test case characterization by regular path expressions*, Proceedings of Formal Approaches to Testing of Software (FATES),2001

[23] W.E. Lewis, Software Testing and Continuous Quality Improvement (Auerbach Publishers, Third Edition, 2000)

[24] J.C.Lin and P.L.Yeh, *Using genetic algorithms for test case generation in path testing,* Proceedings of the 9th Asian Test Symposium (ATS'00), (4-6) December 2000, Taipei Taiwan, pp. 241-246

[25] T.J. McCabe and A.H.Watson, *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*, National Institute of Standards and Technology Special Publication, 1996

[26] J. D. McGregor and D. A. Sykes, A Practical Guide to Testing Object-Oriented Software (Addison-Wesley, Third Edition, 2001)

[27] J. Miller, M. Reformat, and H. Zhang, *Automatic Test Data Generation Using Genetic Algorithm and Program Dependence Graphs*, Information and Software Technology, Vol. 48, 2006, pp. 586–605

[28] G.J.Myers, The Art of Software Testing (John Wiley & Sons, 1979, New York).

[29] R.P Pargas, M.J.Harrold, and R.R.Peck, *Test-Data Generation Using Genetic* algorithms, Journal of Software Testing, Verification and Reliability, Vol.9, No.4, September 1999, pp. 263-282.

[30] J.Poole, *A Method to Determine a Basis Set of Paths to Perform Program Testing* http://hissa.nist.gov//publications/nistir5737, 2004

[31] R. S. Pressman, Software Engineering: A Practitioner's Approach (Tata McGraw-Hill, Fifth Edition, 2003, New Delhi)

[32] P. N. Robillard and M. Simoneau , *Iconic control graph representation,* ACM Software Practice and Experience, Vol. 23, No.2, February 1993, pp. 223-234

[33] M.Roper, I. MacLean, A. Brooks, J. Miller and M. M.Wood, *Genetic Algorithms and the Automatic Generation of Test Data,* http://citeseer.ist.psu.edu/135258.html, 1995

[34] M.Salloum and S.Salloum, *Efficient algorithm for generating a basis set of path for white-box testing*, International Conference on Computer Science and Applications, June 2006, pp. 27-31

[35] I. Sommerville, Software Engineering (Pearson Education, Seventh Edition, 2004, New Delhi)

[36] T.A.Standish, Data Structures in Java (Addison Wesley, Second Edition, 1998, New Delhi)

[37] H.H.Sthamer, The Automatic Generation of Software Test Data Using Genetic Algorithms (PhD Dissertation, University of Glamorgan, November 1995).

[38] R.Torkar,*Towards Automated Software Testing, Techniques, Classifications, and Frameworks*, Karlskrona Blekinge Institute of Technology, 2006

[39] G.M.Weinberg, The Psychology of Computer Programming (Dorset House Publishing

Co, Silver Anniversary Edition, 1998).

[40] J.A. Whittaker, *What Is Software Testing? And Why Is It So Hard?*, IEEE Software, February 2000.

[41] J.Yan and J. Zhang, *An efficient method to generate feasible paths for basis path testing*, Information processing letters, Vol. 107, No.10, 2008, pp. 87-92

[42] H.Zhu, P.A.V Hall, and J.H.R.May, *Software unit test coverage and adequacy*, ACM Computing Surveys, Vol. 29, No. 4, December 1997, pp. 366-427