Test Case Generation of a C Program Using CFG

Thesis

submitted in partial fulfillment of the requirements

for the award of degree of

Master of Technology
in
Computer Science and Applications

Submitted By

Jasjeet Singh
(601103009)

Under the supervision of

Mrs. Sunita Garhwal
Assistant Professor, SMCA

SCHOOL OF MATHEMATICS AND COMPUTER APPLICATIONS
THAPAR UNIVERSITY
PATIALA – 147004

July 2013

Certificate

I hereby certify that the work which is being presented in the thesis entitled, **"Test Case Generation of a C Program Using CFG"**, in partial fulfillment of the requirements for the award of degree of Master of Technology in Computer Science and Applications submitted in School of Mathematics and Computer Applications, Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of **Mrs. Sunita Garhwal** and refers other researcher's work which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for award of any other degree of this or any other University.

Jasjeet Singh

Jasjeet Singh
(601103009)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

Sunita 15/7/13

(Mrs. Sunita Garhwal)
Assistant Professor
SMCA

Countersigned by:

(Dr. Rajesh Kumar)
Head
School of Mathematics and Computer Applications
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First and foremost, I would like to express my sincere gratitude to my supervisor **Mrs. Sunita Garhwal**, School of Mathematics and Computer Applications, for her immense help, guidance, stimulating suggestions and for encouraging me in my thesis work. She always provided me a motivating and enthusiastic atmosphere to work with, it was a great pleasure to do this thesis under her supervision.

I am highly thankful to **Dr. Rajesh Kumar**, Head, School of Mathematics and Computer Applications, for his moral support, cheering concern and his optimistic attitude for doing things.

I would also like to express my appreciation to my colleagues, room mates, and **Er. Manjeet Singh.** I am also very thankful to the entire faculty and staff members of School of Mathematics and Computer Applications for their direct and indirect help, cooperation, love and affection which made my stay at Thapar University memorable. The most valuable thing I acquired from the two years study in Thapar University is to protect, survive and endure myself in this world with increased confidence and additional faith in my abilities to achieve. Finally, my special thanks go to my family for their never-ending love and support and Almighty for giving me such a capability to do this.

Jasjeet Singh

Jasjeet Singh

(601103009)

Abstract

Software testing is an important phase of software development life cycle. It ensures the quality of software. Software testing takes place throughout the software development life cycle. Based on structure of the program data flow based testing can be applied.

Data Flow Testing is a form of structural testing which focus on the definition and usage of variables but not relate to the structure of the program. A test case selection criterion is based on an analysis of the ways in which values are allied with variables, techniques similar to those used in compiler optimization. This analysis focuses on the occurrences of variables within the program.

Data flow testing are of two different types: def/use testing, in which a set of paths are derived, test cases are designed to test these paths; and the concept of splitting a program into small pieces(called slices), is based on its variables, to ease the testing of huge software systems.

The proposed approach automates the data flow testing to large extent. An C program is given as a input to software, it will generate Control Flow Graph (CFG) of the given program and depict variables with nodes. An algorithm has been designed to find Complete path in a graph. It also gives path for the selected (criteria) data flow testing criteria and also generates test cases for that path with Random Testing Technique.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This chapter describes the commonly used software quality attributes and currently available metrics for measuring software quality. It also provides an overview of software testing procedures and describes the impact of inadequate software testing.

1.1. Software Quality Attributes

Quality is defined as the bundle of attributes present in a commodity and, where appropriate, the level of the attribute for which the consumer holds a positive value. Different community understands different from quality like specific to requirements, appropriate for the purpose and satisfaction. In other words, quality from user view must deal with installation, convenient to use. Software quality is commonly recognizes as error free program. Attributes can be classified into categories: operation and revision. These attributes are shown in Table 1.1.

- *Product operation* captures how fast and effective the software is accomplishing a specific task. It includes reliability, integrity, correctness, and usability.
- *Product revision* relates to how easily it can be change, maintain, or update the software product.

1.2. Software Testing

Software testing is a critical element of software quality assurance and software development life cycle (SDLC). Testing represents the ultimate review of code, design, and specification. Program testing is the procedure of finding errors and bugs, to distinguish the differences among current and required output of the program [7].

Table 1.1. Software Quality Attributes [10]

Attribute	**Description**
Product Operation	
Correctness	How well the program performs its required function and meets customers' needs
Reliability	How well the program can be perform its function with required precision
Integrity	How well accidental and intentional attacks on the program can be withstood
Usability	How easy it is to learn, operate, prepare input of, and interpret output of the program
Efficiency	Amount of resources required by the program to perform its function
Product Revision	
Maintainability	How easy it is to detect and correct an error in the software
Flexibility	How easy to change the software with need
Testability	How easy it is to define the software performs its intended function correctly
Product Transition	
Interoperability	How easy it is to combine existing system into another
Reusability	How easy it is to use the program or its parts in other applications
Portability	How easy it is to run the software from one platform to another

The goal of testing is to ensure that the program being developed meets with the customer's necessities and work properly. Software testing is the dynamic execution of software and the compare the results of that execution against a set of pre-defined criteria. Execution is the process of running the program on a computer with or without any form of software testing tools. What the software output compared against the predefined criteria, whether the software behaved correctly or not. Every time the software is executed, the customer tests it. Therefore, program should be tested before it gets to the customer with the intent of finding and removing all errors.

Software is tested from two different ways [14]:

- Program logic is exercised using “white box” testing techniques.
- Requirements are exercised using “black box” testing techniques.

In both cases, the intent is to find the maximum numbers of errors with the fewer amounts of time and effort.

1.3. Need for Testing

Testing is an important component of software development. It plays an important role i.e. everything is tested first before use. Software testing itself can have any different purposes (quality assurance, validation, performance etc). Testing is needed to verify and validate that the software that has been built to meet these specifications [22]. Once source code has been generated, code must be tested to detect and debug errors. The goal is to design a series of test cases that have a high probability of finding errors. Cost of software failure is motivating force for well-planned and thorough testing. Testing does not reveal error if the expected and generates outcomes are same. It enhances the integrity of a system by detecting deviations in design and errors in the system. It also adds value to the product by conforming to the user requirements.

Some basic terminology used in Software Testing:

- Error: difference between Output and predefined output.
- Fault: condition that causes the program to fail to perform its function.
- Failure: inability of a system to perform its function according to its requirement.
- Test case: inputs, execution conditions and expected results designed for testing a program.
- Test suite: collection of test cases, related to a testing goal.
- Driver: utility program that applies test cases to a program under testing.
- Stub: by which missing components are replaced. It give a output to complete the testing.

1.4. Testing Objectives

Main objective is to design test suit, which would tend to uncover errors in the program. Because, with these as the guide map, the program will be functioning properly and give correct output. Besides, there is no way possible to support that software has no errors. According to the

objectives stated previously, if testing is conducted successfully it will uncover errors in the software program. The main rules that can serve well as testing objectives are [7]:

- Testing is a activity in which executing a program with test cases to find error.
- A successful test case is one that has a high probability of finding an as-yet undiscovered error.

It is almost impossible to notice all errors. In addition, results of testing provide a good indication of software program reliability and some indication of software quality as a whole.

1.5. Testability

Software testability [32] is simply how easily a program can be tested. In ideal circumstances, a software engineer designs a computer program with *testability* in mind. This enables the individuals to design effective test cases more easily for testing. Sometimes programmers are making checklist of possible design points, features, etc., can be useful in negotiating with them. There are certain measures that could be used to identify testability. Testability is also used by the military to mean how easily software can be checked and repaired in the field. The checklist characteristics that lead to testable software program are:

- Operability**:** More efficiently it can be tested if it works better.
- Observability: What you see is what you test, externally measured the internal state variables of the system.
- Controllability: Refers to the ability of software, controller arbitrarily alter the functionality of the system. Software is better controlled when testing can be more automated and optimized."
- Decomposability: Decomposing a software problem into a small number of less complex sub problems. By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.
- Simplicity: The less there is to test, the more quickly it can test.
    - Functional simplicity
    - Code simplicity
- Stability: The fewer the changes, the fewer the disruptions to testing.
- Understandability: More observations about the software, smarter the software will be tested.

1.6. Testing Process

Testing Process in each phase of the development life cycle process will have a specific input and a specific output. The phases of the development life cycle of project can be divided into the following:

- Software requirements phase.
- Software Design
- Implementation
- Testing
- Maintenance

In the whole development life cycle process, testing consumes largest amount of time. But most of the developers neglect testing phase. As a result, erroneous software is released. The testing team should be involved right from the requirements stage itself. The various phases involved in testing, with regard to the software development life cycle are:

- Requirements stage: All the requirements should be documented properly for further use and this document is called Software Requirements Specifications.
- Test Plan: A test plan document should be prepared after the requirements of the project are confirmed.
- Code Reviews: Once the code is ready for release, the tester should be ready to do unit testing for the code.
- Test Execution and Bugs Reporting: The test reports should be documented properly and the bugs have to be reported to the developer after the testing is completed.
- Release to Production: Before releasing to production, another round of top-level testing is done.

1.7. Testing Strategy

A strategy [32] for software testing must accommodate low-level tests that are necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements. A strategy must provide guidance or steps. These steps of the test strategy occur at a time when deadline pressure begins to rise, progress must be measurable and problems must surface as early as possible.

Software testing is often referred to as *verification and validation* (V&V). *Verification* refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

Testing from which quality can be assessed and, errors can be removed. Quality is included into software throughout the whole process of software development life cycle. Software testing is quality assurance. Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test. The main levels during the development process as defined by *unit, integration,* and *system testing* that are distinguished by the test target without implying a specific process model. Other test levels are classified by the testing objective.

Table 1.2. Strategies Implementation

STRATEGIES	**IMPLEMENTED AS**
Unit testing	Code
Integration testing	Design
Validation testing	Requirements
System testing	System engineering

- Unit Testing: It is done at the lowest level. It tests the basic unit of software, which is the smallest testable piece of software, and is often called "unit", "module", or "component" interchangeably.
- Integration Testing: It is performed when two or more tested units are combined into a larger structure. The test is often done on both the interfaces between the components and the larger structure being constructed, if its quality property cannot be assessed from its components.
- System Testing: It tends to affirm the end-to-end quality of the entire system. System test is often based on the functional/requirement specification of the system. Non-functional quality attributes, such as reliability, security, and maintainability, are also checked.

1.8. Software Testing Techniques

These techniques gives systematic guidance for designing tests that to exercise the internal logic and requirements of software components, and defining the input and output domains of the program to uncover errors in program performance, function, and behavior. There are two types of Software Testing Techniques as follows:

- Static Testing: It refers to testing the software requirement specification (SRS), software design specification (SDS) and other non-executable items through requirement analysis, audits, desk checks, inspections, walk-through etc. [1]. Static testing is employed to verify the correctness of requirements, designs and code before execution of test cases. Static testing will also audit newly developed or reused code for adherence to established standards. Static software sometimes reveals error which is even not detected by dynamic testing.
- Dynamic Testing: It involves the development of test cases, test procedures, execution of test cases, structure of test logs and anomaly or incident reports. Black box and white box testing techniques are two types of dynamic testing. Both techniques require a set of well-developed and well-structured test cases. We can't say software product is absolutely correct unless we perform exhaustive testing [4]. Exhaustive test need a set of test cases that will guarantees the explicitly exercises every possible, module path and every possible combination of paths with every possible module input and every possible combination of module inputs, but exhaustive testing is not possible[4].

If black box testing criterion is chosen, test cases should be written addressing the functionality of the application. If it is white box, then the test case should be written for the internal structure of the system.

1.8.1. Black-Box Testing

The black-box approach is a testing method in which test data are derived from the specified functional requirements without regard to the final program structure. It is also termed data-driven, input/output driven or requirements-based testing. Because only the functionality of the software module is of concern, black-box testing also mainly refers to functional testing -- a testing method emphasized on executing the functions and examination of their input and output data. In testing, various inputs are exercised and the outputs are compared against specification to validate the correctness. All test cases are derived from the specification. No implementation

details of the code are considered. Black box focuses on the functional requirements of the software. It attempts to find errors in the following categories [8]:

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

Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing because black-box testing purposely disregards control structure and attention is focused on the information domain.

1.8.2. White-Box Testing

Opposite to black-box testing, software is viewed as a white-box, or glass-box in white-box testing, as the structure and flow of the software under test are visible to the tester. Testing plans are made according to the details of the software implementation, such as programming language, logic, and styles. Test cases are derived from the program structure. Using white-box testing methods, the software engineer can derive test cases that [1]:

- 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.

There are many techniques available in white-box testing:

1.8.2.1. Control flow testing

Test cases are generated on the basis flow of control of a program. It includes following type of criterion [2]:

- Statement coverage: Test cases are designed so that using these test cases each statement of the modules will be exercised at least once.
- Branch coverage: Test cases are designed so that using these test cases each branch of the modules will be exercised at least once.
- Condition coverage: Test cases are designed so that using these test cases each condition in the modules are evaluated as true and false at least once.
- Branch /condition coverage: both branch and condition coverage achieved.

- Multiple conditions: Multiple conditions coverage technique states that test cases must be written such that all possible combinations of conditions in each decision are taken at least once.
- Loop coverage: Loop coverage technique states that test cases must be written to test the loop counters.

1.8.2.2. Data flow testing

Test cases are generated on the basis on data definitions and their subsequent uses. Data-flow based testing is based on following criterion [34]:

- All definition-uses (all du paths): It requires that every definition of every variable at all du paths should be exercised under the test.
- All Uses: In this test set include at least one path segment from every definition of every variable to every use (both C-use and P-use) of that definition.
- All P-uses/some C-uses: In all P-uses/some C-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition, if there are some definitions of variables that do not follow P-use, then at least one C-use for that definition is exercised.
- All C-uses/ some P-uses: In all C-uses/some P-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition, if there are some definitions of variables that do not follow this then add at least one predicate use test cases are required to cover every definition.
- All definitions: In this, test set includes every definition of every variable be covered by at least one use of that variable, so it will be computational use or predicate use.
- All P-uses: In all P-uses, test set include at least one path segment from every definition of every variable to every predicate use of that definition, if there are some definitions of variables that do not follow this, then leave them.
- All C-uses: In all C-uses, test set include at least one path segment from every definition of every variable to every computational use of that definition, if there are definitions of variables that are not covered by the above prescription then, leave them.

One reason is that all the above techniques will need some knowledge of the specification of the software under test. Another reason is that the idea of specification itself is broad -- it may contain any requirement including the structure, programming language, and programming style

as part of the specification content. The boundary between black-box approach and white-box approach is not clear-cut. Many testing strategies mentioned above, may not be safely classified into black-box testing or white-box testing.

1.9. Impact of Inadequate Testing

Software testing is an important activity in the software development life cycle. Testing software is far more difficult than exercising a program to see whether it works or not. There are a number of applications in which reliability is more important than other traditional application area [11]. Currently, there is deficiency of good available performance measures, procedures, and tools to hold up software testing. If these infra-technologies are available, the costs of performance of programs would turn down and the quality of software would increase. This would lead to better testing for products that to be developed.

The impact on the software industry due to lack of testing tools, standardized test technology are:

- increased failures due to quality issue
- increased software development costs/time
- increased time to market due to ineffective testing
- increased market transaction costs

Chapter 2

Literature survey

This chapter describes the various works that have been done in field of Data-Flow Testing and its criteria. All the terminologies require to understand data flow testing are also describing.

2.1. Control Flow Graph

Control flow diagrams are a keystone in white box testing. It shows flow of the program. It has nodes and edges, nodes correspond to block of statements or a statement and edges show the flow of program statements. Control flow graph is a directed graph. A program is a collection of statements. To generate Control Flow Graph (CFG) firstly program is divided into basic blocks. Each block has a entry and exit point. Nodes in the Control flow graph represent blocks of statements or statement. Edges are the arrow, connected component which connect the blocks and show how the flow of program. A conditional transfer statement is always the end statement of a block, and determines which block is to be executing out of two successor block. Program graphs allow the tester to view the structure of the program visually.

The program (or control flow) graph for this program is shown in figure 2.3. Each node in the graph corresponds to a statement in the program figure 2.2; however, *'A'* statement block corresponds to node *'A'*.

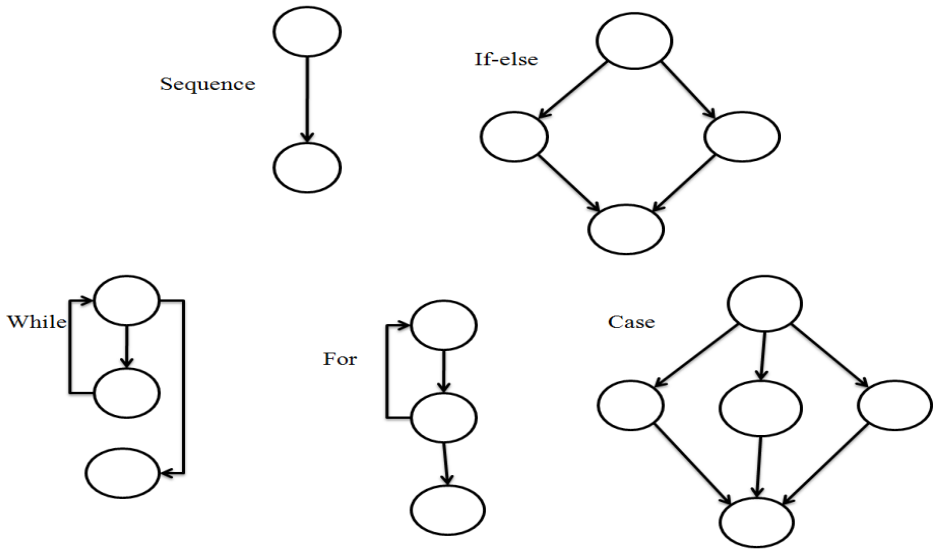

Figure 2.1. The structured constructs in flow graph [32]

Following are the flow graphs Terminologies:

- A fork(Predicate node) in a program at which point the control can diverge.
- A junction in a program where the control flow can merge.
- A process block is a sequence of program statements. A program does not jump into or out of a process. A process has one entry and one exit.
- Entry node is from where the execution of the program starts.
- Exit node is the node at where execution ends normally.
- Complete paths in which entry node and exit node are included. This is useful for testing because, it is hard to execute paths that begin at an arbitrary statement. It is hard to stop at an arbitrary statement without changing the code being tested.
- Simple path is a path in which all nodes, possibly the first and the last, are distinct.
- Loop-free path is a path in which all nodes are distinct. No repetition of nodes.

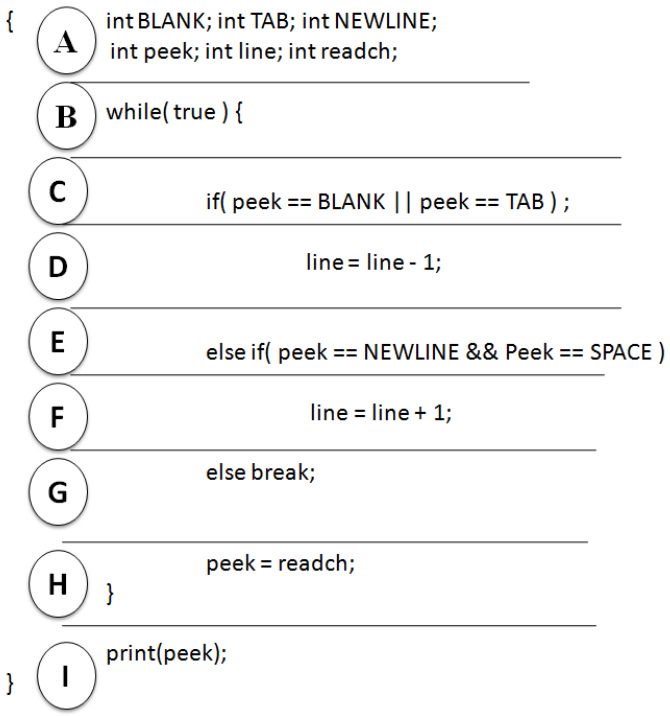

Figure 2.2. Program

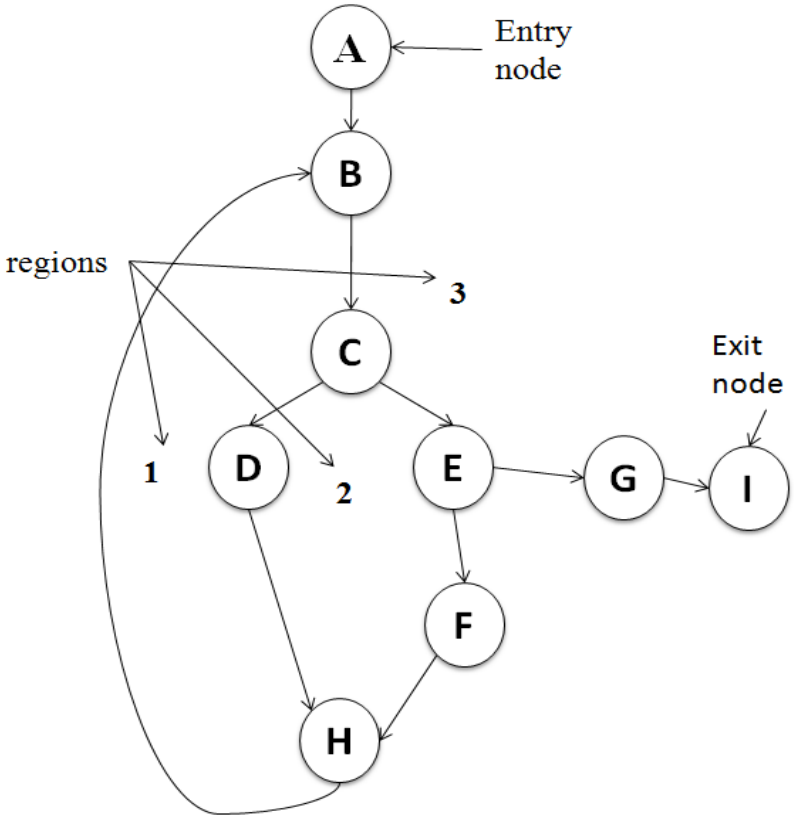

Figure 2.3. CFG

2.1.1. Cyclomatic Complexity

Cyclomatic Complexity is a software metric used to know the complexity of the program [7]. It also helps in to determine independent path for executions and estimate the hazard allied with that program. Cyclomatic Complexity of a program can be calculate with the following methods:

- The number of regions of the flow graph.
- Cyclomatic complexity V(G), for a flow graph G, is defined as

  V (G) = E - N + 2

  Where E is the number of flow graph edges, N is the number of flow graph nodes.
- Cyclomatic complexity V(G), for a flow graph G, is also defined as

  V (G) = P + 1 Where P is the number of predicate nodes contained in the flow graph G.

2.2. The Definition Use Graph

When we add variables to nodes and edges, this is called definition use graph. A test case selection criterion is based on an analysis of the ways in which values are allied with variables.

This analysis focuses on the occurrences of variables within the program. Classification of variables are as definitional (def), computation-use(c-use), or predicate-use (p-use). Table 2.1 represent nodes and variables on nodes.

Table 2.1. Nodes and variable used

Nodes	Def	C-use	P-use
A	BLANK, TAB, NEWLINE, peek, line, readch	-	-
B	-	-	-
C	-	-	Peek, NEWLINE, BLANK, TAB
D	line	line	-
E	-	-	Peek, NEWLINE, SPACE
F	line	line	-
G			
H	Peek	readch	-
I		peek	

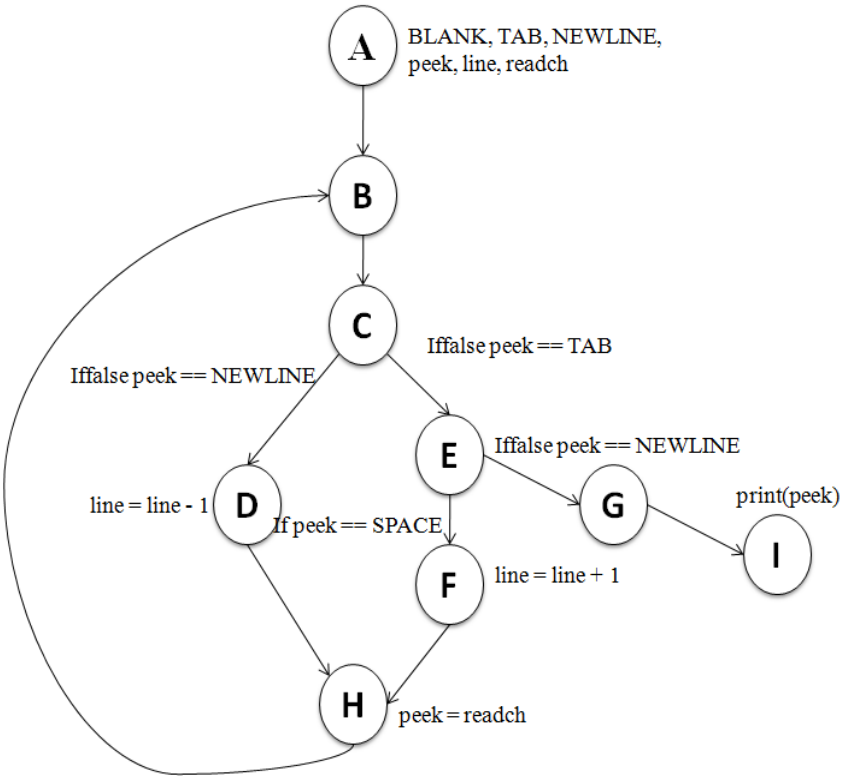

Figure 2.4. Def use graph

4.6. Data Flow Testing

Data flow testing can be considered to be a form of structural testing. Structural testing techniques require the tester to have access to details of the program's structure. Data flow testing focuses on the variables used within a program. Variables are defined and used at different points within the program; data flow testing allows the tester to chart the changing values of variables within the program.

There are two major forms of data flow testing:

- The first, called define/use testing [33], uses a number of simple rules and test coverage metrics.
- The second uses "program slices" – segments of a program.

The term static analysis refers to the fact that the tester does not have to run the program to analyze it. Static analysis allows the tester to focus on define/reference anomalies [33]:

Dynamic data testing is different: by selecting, for instance, paths through the program according to the locations and properties of references to variables within the program code, a program can be analyzed in terms of how the variables are affected, assigned and changed throughout the course of the program when running with certain test data. This complements (or maybe even replaces) the concept of selecting paths according to the structure of the program (looking at its loops, branches etc.).

2.3.1. Def/Use Testing

Def/Use testing uses paths of the program graph, linked to particular nodes of the graph that relate to variables, to generate test cases. The term "Def/ Use" refers to the two main aspects of a variable: it is either defined (a value is assigned to it) or used (the value assigned to the variable is used elsewhere – maybe when defining another variable). Def/use testing was first formalized by Sandra Rapps and Elaine Weyuker [33].

The program is referred to as *P*, and its graph as *G (P)*. The program graph has single entry and exit nodes, and there are no edges from a node to itself. The set of variables within the program is called *V*, and the set of all the paths within the program graph *P (G)* is *PATHS (P)*. Within the context of define/use testing, with respect to variables there are two types of nodes: defining nodes and usage nodes. The nodes are defined as follows:

- Defining nodes, referred to as *DEF (v, n)*: Node n in the program graph of *P* is a defining node of a variable *v* in the set *V* if and only if at *n*, *v* is defined. For example, with respect to a variable x, nodes containing statements such as *"input x"* and *"x = 2"* would both be defining nodes.
- Usage nodes, referred to as *USE (v, n)*: Node n in the program graph of P is a usage node of a variable v in the set V if and only if at n, v is used. For example, with respect to a variable x, nodes containing statements such as "print x" and "a = 2 + x" would both be usage nodes. Usage nodes can be split into a number of types, depending on how the variable is used. For instance, a variable may be used when assigning a value to another variable or it may be used when making a decision that will affect the flow of control of the program. The two major types of usage node are:
    - P-use: predicate use – the variable is used when making a decision (e.g. *if b > 6*).
    - C-use: computation use – the variable is used in a computation (for example, *b = 3 + d* – with respect to the variable *d*).

- Definition-use (du) paths: A path in the set of all paths in *P (G)* is a *du-path* for some variable *v* (in the set *V* of all variables in the program) if and only if there exist *DEF (v, m)* and *USE (v, n)* nodes such that m is the first node of the path, and n is the last node.
- Definition-clear (dc) paths: A path in the set of all paths in *P (G)* is a *dc-path* for some variable *v* (in the set *V* of all variables in the program) if and only if it is a du-path and the initial node of the path is the only defining node of *v* in the path.

Data-flow testing [1] is underneath control-flow testing technique in which the interested is in lifecycle of data variables. There are many path selection criteria describe blow:

- **All-paths** a graph containing an infinite number of paths. An all-path criterion is when we select all the path of graph to generate test cases. In figure 2.4 graph {(*A-B-C-D-H-B-C-E-G-I*),(*A-B-C-E-F-H-B-C-E-G-I*), (*A-B-C-E-G-I*)} satisfies all path criteria. Another example figure 2.5 {(*A-C*), (*A-B-E*), (*A-B-D-F*), (*A-B-D-G*)} satisfies all-path criteria.
- **All du-paths** considering figure 2.6 paths {(*A-B-D-E-G*), (*A-C-D-F-G*), (*A-B-D-F-G*)} satisfies all-du-paths criterion. Graph can contain at least one complete path for each def of a variable. There are multiple paths for a global def of a variable. Path (A-B-D-F-G) is included because, it have a def-clear path w.r.t x(x is def in node B and c-use in node G).

- **All-uses** consider the graph of figure 2.6 Paths {(*A-B-D-E-G*), (*A-C-D-F-G*)} are enough to satisfy all-uses criteria. It includes all c-use and p-use.
- **All-p-uses/some-c-uses** consider figure 2.7 paths {(*A-C*), (*A-B-D-F*), (*A-B-E-D-G*)} satisfies all-p-use/some-c-use criteria. Path {(*A-B-E-D-F*)} is not included, it has a def-clear path w.r.t to y form node E to F. If we consider this path then it will become all-use criteria.
- **All-c-uses/some-p-uses** again in graph of figure 2.7 Path {(*A-B-E-D-F*)} satisfies all-c-uses/some-p-uses criteria, but does not include all-p-use. since there is no path
- **All-defs** consider the graph of figure 2.7 Paths {(*A-B-D-F-G*)} satisfies all-defs. It includes a p-use of variable x, y, and z. In this we have to cover all the def of variables. Includes a def-clear path w.r.t. every def of variable to some c-use or p-use.

All-paths is a strongest criteria and all-def is less strong, this is in strength decreasing order.

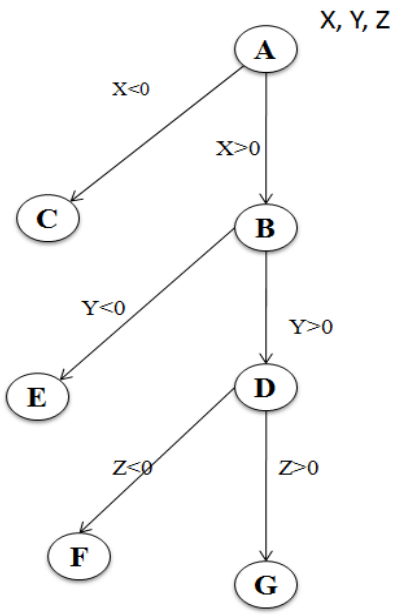

Figure 2.5. All Paths

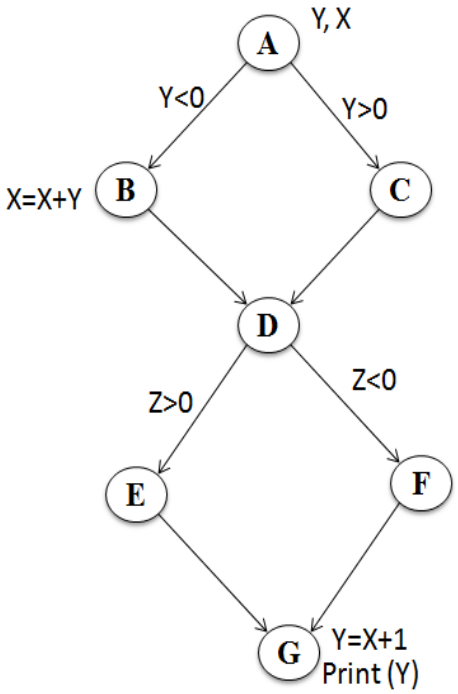

Figure 2.6. All du-path and All-use

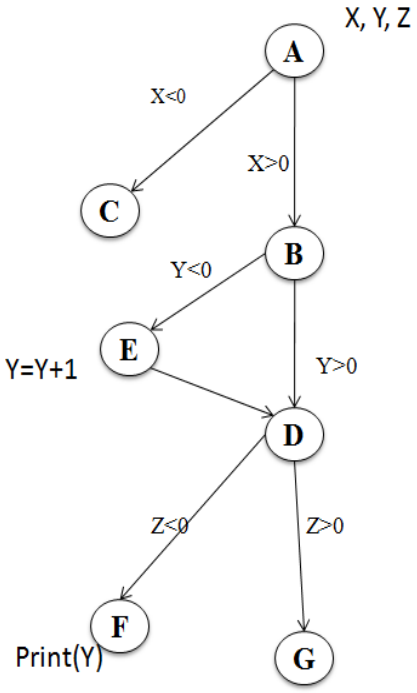

Figure 2.7. All-use, All-p-use\All-c-use and All-defs

2.3.1.1. Rapps-Weyuker Metrics

Associated with the concepts discussed in the previous section are a set of test coverage metrics, also defined by Sandra Rapps and Elaine Weyuker [33]. The metrics – a set of criteria, essentially – allow the tester to select sets of paths through the program, where "the number of paths selected is always finite, and chosen in a systematic and intelligent manner in order to uncover errors".

Rapps and Weyuker noted that there was a relationship between the different metrics: certain metrics expanded upon other metrics – that is, if a set of paths satisfied a certain metric, then it also satisfied all the other metrics below it (for example, if *All-Paths* is satisfied, then so are *All-DU-Paths* and *All-Uses*). A diagram showing the relationship between metrics is shown in figure 2.8. In the diagram (figure 2.8), the arrows show the relationship between metrics. For example, *All-Paths* subsumes (or is *stronger than*) *All-DU-Paths*. However, during the development of the metrics, they had found that *All-Defs* is not necessarily stronger than *All-Edges* and *All-Nodes*: criteria are incomparable in the sense that it is possible for a given def-use graph *G* and sets of complete paths *P1* and *P2*, that *P1* satisfies *All-edges*, but not *All-defs* for *G*, while *P2* satisfies *All-defs*, but not *All-edges*. Similarly, *All-nodes* and *All-defs* are shown to be incomparable [33].

This incomparability meant that different criteria had to be found: at this point, Rapps and Weyuker established that there were two types of usage node – computational use (*C-use*) and predicate use (*P-use*). Therefore the three criteria *All-P-Uses, All-P-Uses/Some-C-Uses, All-C-Uses/Some-P-Uses* were defined. *All-P-Uses/Some-C-Uses* provided a way to satisfy our goal which includes *All-Defs* and *All-Edges*, but requires fewer test cases, in general, than *All-Uses*. *All-C-Uses/Some-P-Uses* was defined to allow tests to be defined where the emphasis is on the flow of data rather than the flow of control (as in the case of *all-p-uses/some c-uses*) [33].

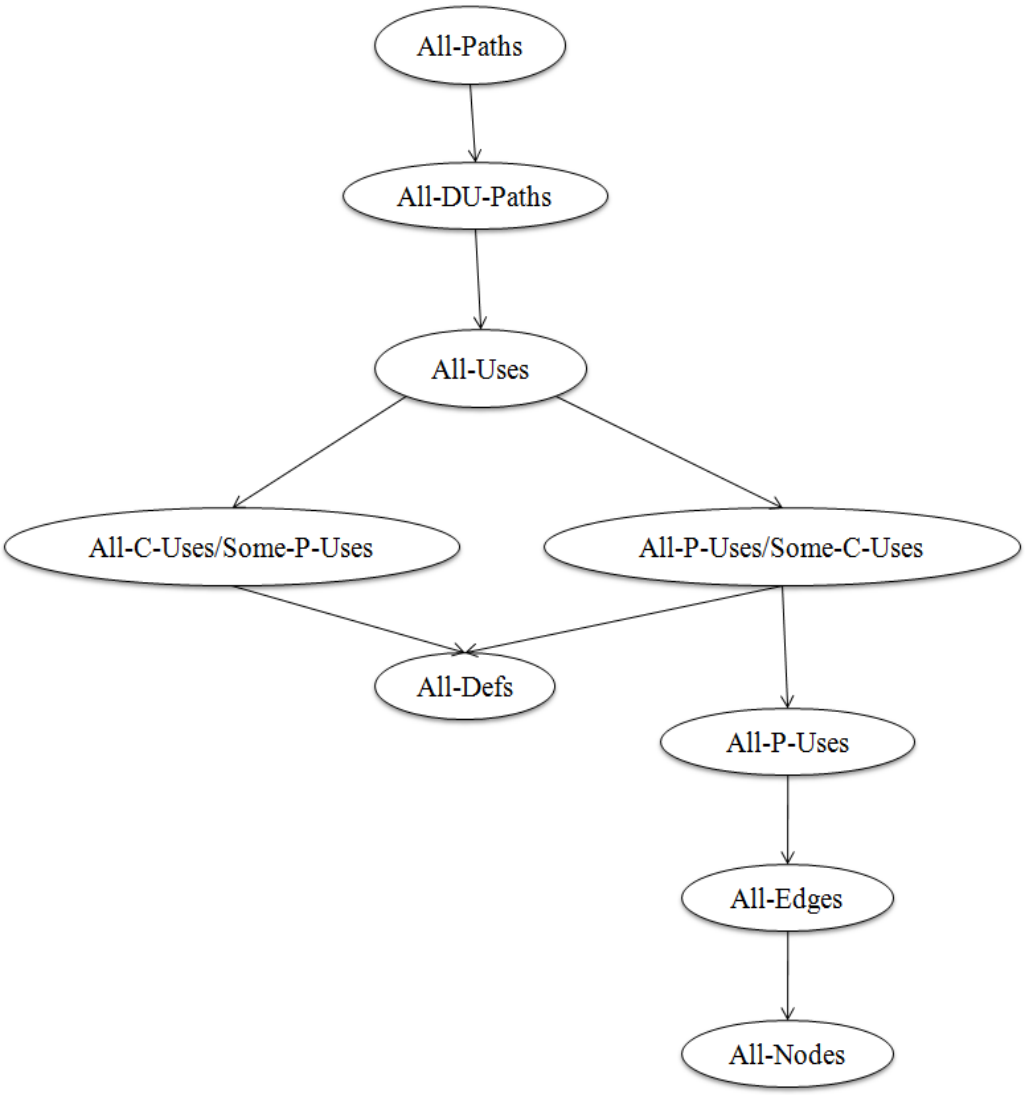

Figure 2.8. The Rapps-Weyuker Metrics [33]

2.3.1.2. Tool support

A search of academic journals has found two software tools that can automate (to a degree) the data flow testing process. The first, written by, Elaine Weyuker and Phyllis Frankl, is called ASSET. ASSET works with programs written in the Pascal programming language [29]. Another data flow coverage tool, for the C programming language, was written by J. R. Horgan and S. London [17].

2.3.2. Program Slices

The concept of program slicing was first proposed by Mark Weiser [25, 26]. According to Weiser, slicing is a source code transformation of a program [25]; this allows a subset of a program, corresponding to a particular behavior, to be looked at individually. This gives the benefit that a programmer maintaining a large, unfamiliar program does not have to understand an entire system to change only a small piece [25].

A program slice with respect to a variable at a certain point in the program, is the set of program statements from which the value of the variable at that point of the program is calculated. This definition can be amended to encompass the program graph concept: by replacing the set of program statements with nodes of the program graph. This allows the tester to find the list of usage nodes from the graph, and then generate slices with them. The relationship between slices also shows the interactions between variables in the code.

2.4. Anomalies

Anomalies are potential bugs that are identifying by examining the patterns in which data is used in the program. For example, data which is killed after that it is used for calculation is a potential bug which may crash our program.

Anomalies represent the data usage which may lead to an incorrect execution of the code.

The notation for representing the patterns is [7]:

- d – defined, created, initialized
- k – killed, terminated, undefined
- u – used
    - c – used in a computation
    - p – used in a predicate
- ~x - indicates all prior actions are not of interest to x
- x~ - indicates all post actions are not of interest to x

Table 2.2 lists the combinations of usage and their corresponding consequences. It can be observed that not all data-flow anomalies are harmful but they are all suspicious and indicate that an error can occur. For example, the usage pattern ‘ku’ indicates that a variable is used after it has been killed which is a serious defect.

Table 2.2. Anomalies

Sign	Anomaly	Explanation
~d	first define	Allowed.
du	define – use	Allowed. Normal case.
dk	define – kill	Potential bug. Data is killed without use after definition.
~u	first use	Potential bug. Data is used without definition.
ud	use – define	Allowed. Data is used and then redefined.
uk	use – kill	Allowed.
~k	first kill	Potential bug. Data is killed before definition.
ku	kill – use	Serious Defect. Data is used after being killed.
kd	kill – define	Allowed. Data is killed and then re-defined.
dd	define–define	Potential bug. Double definition.
uu	use – use	Allowed. Normal case.
kk	kill – kill	Potential bug.
d~	define last	Potential bug.
u~	use last	Allowed.
k~	kill last	Allowed. Normal case.

Chapter 3

Problem Formulation

This chapter includes the problem statement and objectives of the thesis.

3.1. Problem Statement

Most programming language paradigms utilize the concept of variables. Multiple variables can be used together to calculate the values of other variables; and variables can receive their values from other sources – such as human input via a keyboard, for instance. This increased level of complexity can result in errors within programs. The concept of Data Flow Testing allows the tester to examine variables throughout the program, helping him to ensure that none of the aforementioned errors occur.

In typical commercial, the cost of software debugging, testing, and validation activities can easily range from 50 to 75 percent of the total development cost. Decreasing the cost of software development and increase software quality are important objectives. Improved testing tools can decrease the costs of developing software.

There is a need of automaton of data-flow testing. Thus, development of testing tools and measures for software testing address some of the problems that affect the software industry. Good tools for software testing could increase value of software in a number of ways:

- Reduce the cost/time required to develop software and testing them.
- Increase the performance, reliability, and interoperability of software.

3.2. Thesis Objectives

Whole work of this thesis has been done to achieve by following objectives:

- Analyze the data flow testing and test case generation technique.
- Design an algorithm that will generate various paths in a graph.
- Implementation of the proposed algorithm in the java language.

Chapter 4

Work Done

This chapter contains the details of the work that has been done to meet all the thesis objectives.

4.1. A Compiler

An important role of the compiler is to report any errors in the source program that it detects during the translation process. A compiler is a program that can read a program in one language — the source language and translate it into an equivalent program in another language — the target language [25].Target program is an intermediate code. This intermediate code represents nodes and their linking with each other. This intermediate code is used to create the Control flow Graph (CFG). In figure 4.1 and figure 4.2 our Source program and Target program is shown.

```
{
        int BLANK; int TAB; int NEWLINE; int peek; int line; int readch;
                while( true ) {
                 if( peek == BLANK || peek == TAB ) ;
                else if( peek == NEWLINE )
                        line = line + 1;
                else break;
                peek = readch;
        }
}
```

Figure 4.1. Source Program

```
L1:L3: if peek == BLANK goto L7
       iffalse peek == TAB goto L6
L7:L5: goto L4
L6:    iffalse peek == NEWLINE goto L9
L8:    line = line + 1
       goto L4
L9:    goto L2
L4:    peek = readch
       goto L1
L2:
```

Figure 4.2. Target Program

4.2. Control Flow Graph

From our figure 4.3 target Program, we can create Control Flow Graph (CFG) of sample program as shown in figure 4.4.

In our Target Program *L1, L2, L3 … Ln* represents nodes in the graph and contents are colon (‘:’) separated as shown in figure 4.3. By using regular expression separate nodes and links and create a table like table 4.1. Now this table is further used to draw Control Flow Graph.

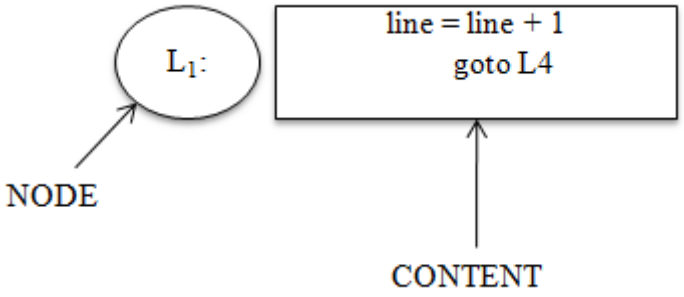

Figure 4.3. Content on Nodes

Table 4.1. Nodes and Links

Nodes	Links
L_1	L_3
L_2	-
L_3	L_7, L_6
L_7	L_5
L_6	L_8, L_9
L_5	L_4
L_8	L_4
L_4	L_1
L_9	L_2

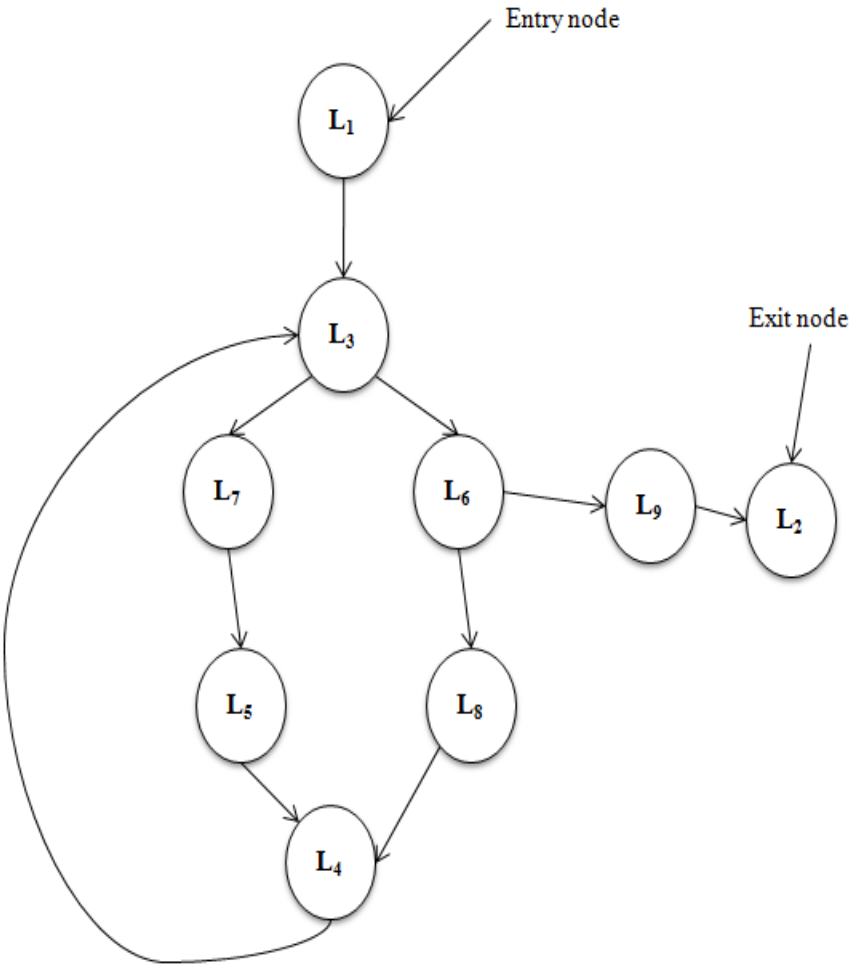

Figure 4.4. CFG of Source Program

4.3. Algorithm for Finding Paths in CFG

This algorithm has been implemented to find paths in Control Flow Graph (CFG) using Java. Algorithm detects all the independent paths in a CFG. The algorithm is also capable of finding cycles in CFG.

Nodes and links (edge) of the CFG are stored in hashtable. Nodes are the keys of hashtable and links as value, so nodes and the links act as key-value pair in the hashtable. Two stacks are used to implement this algorithm and file is created to store individual paths.

In table 4.1 nodes and their links are shown, which represents graph of figure 4.4.

Input: CFG (table 4.1) **Output:** Paths of CFG **Algorithm***:* find paths of Control Flow Graph (CFG)
STEP 1:- Traverse the graph, when a predicate node is encountered then pushes the links to stack and also push path into other stack. STEP 2:- Pop a node and a corresponding path from stacks and start path traversal from this node. If predicate node is encounter then go to Step 1, else repeat STEP 2 until the stack is empty. But stop when a node in the path is repeated. STEP 3:- At the end we have a list which has only one complete path and various intermediate paths. Take the complete path and compare it with other intermediate paths to get a complete independent path.

Example: In figure 4.4 Node L_3 is predicate node, so push the corresponding links into stack *A* and their path (through which we reach these nodes) into stack *B*.(note – these paths are identical because to reach L_7 and $L_{6,}$ common path followed is *L1-L3*.) as shown in figure. 4.5.

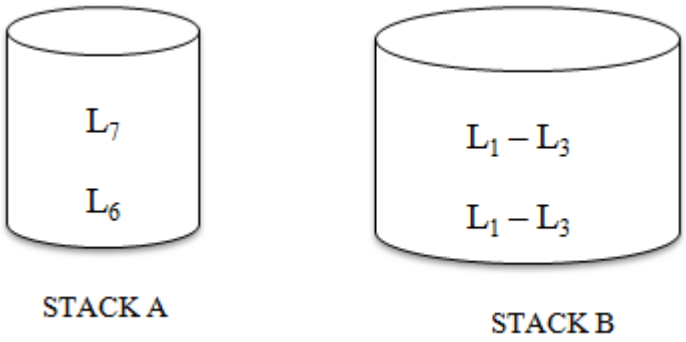

Figure 4.5. Stack A and Stack B

As shown in figure 4.6 *L1* is repeated and the path is terminated right there. This path is stored in list shown in figure 4.7.

$L_1 – L_3 – L_7 – L_5 – L_4 – L_1$

Figure 4.6. Intermediate Path

$L_1 – L_3 – L_6 – L_8 – L_4 – L_1$
$L_1 – L_3 – L_6 – L_9 – L_2$
$L_1 – L_3 – L_7 – L_5 – L_4$

Figure 4.7. Intermediate Paths

Figure 4.7 shows the content of list and path $L_1 – L_3 – L_6 – L_9 – L_2$ is a *complete path*.

Consider the intermediate path $L_1 – L_3 – L_6 – L_8 – L_4 – L_1$ compare the last link (L_1) with all the links of the complete path. Where ever L_1 is encountered in the complete path replace all the link starting from first link till L_1 with the intermediate path and form a new *complete path* which is $L_1 – L_3 – L_6 – L_8 – L_4 – L_1 – L_3 – L_6 – L_9 – L_2$. Now consider the second intermediate path $L_1 – L_3 – L_7 – L_5 – L_4$ and search for link L_4 in newly constructed *complete path* $L_1 – L_3 – L_6 – L_8 – L_4 – L_1 – L_3 – L_6 – L_9 – L_2$. Where ever L_4 is encountered in the complete path replace all the link starting from first link till L_4 with the intermediate path and form a new *complete*

path which is $L_1 – L_3 – L_7 – L_5 – L_4 – L_1 – L_3 – L_6 – L_9 – L_2$. Repeat this technique for the rest of intermediate path. These are the paths which are present in fig. 4.4 CFG and stored in file like shown in figure 4.8.

$L_1 – L_3 – L_6 – L_9 – L_2$
$L_1 – L_3 – L_6 – L_8 – L_4 – L_1 – L_3 – L_6 – L_9 – L_2$
$L_1 – L_3 – L_7 – L_5 – L_4 – L_1 – L_3 – L_6 – L_9 – L_2$

Figure 4.8. Paths in CFG

4.4. Generate Test Cases

Random Testing Technique is used in this automatic test case generation tool to select values randomly from the input domain of the program input variables. These randomly generated values are assigned to the variable and then evaluate the predicate expressions for all decision making nodes in a path. Then, randomly generated values are used to satisfy the predicate expressions. The values which will be able to satisfy the conjunctive clause for a path become the test data for that path to execute.

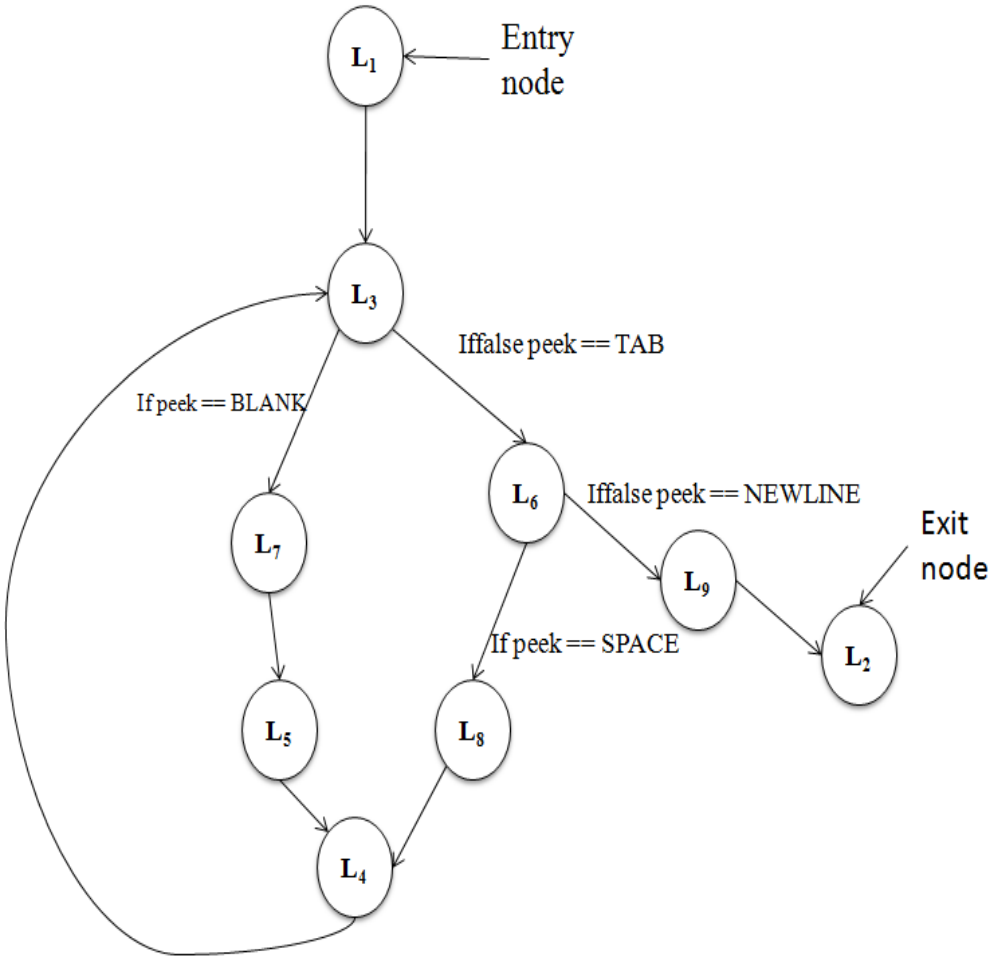

Figure 4.9. Predicate Nodes in CFG

To execute path $L_1 - L_3 - L_6 - L_9 - L_2$ in fig. 4.9 condition *iffalse peek ==TAB* on node L_3 and condition *iffalse peek = =NEWLINE* on node L_6 is to be true. We randomly choose values and assign to variable i.e. *peek, TAB, NEWLINE*

Example:

When,

peek = 0

TAB = 1

NEWLINE = 1

BLANK = 1

Path $L_1 - L_3 - L_6 - L_9 - L_2$ is executed for these values.

4.5. Screen Shorts and Project Flow

When tool is started, file selection dialog shows where the required file can be selected. Browser is shown in figure 4.10.

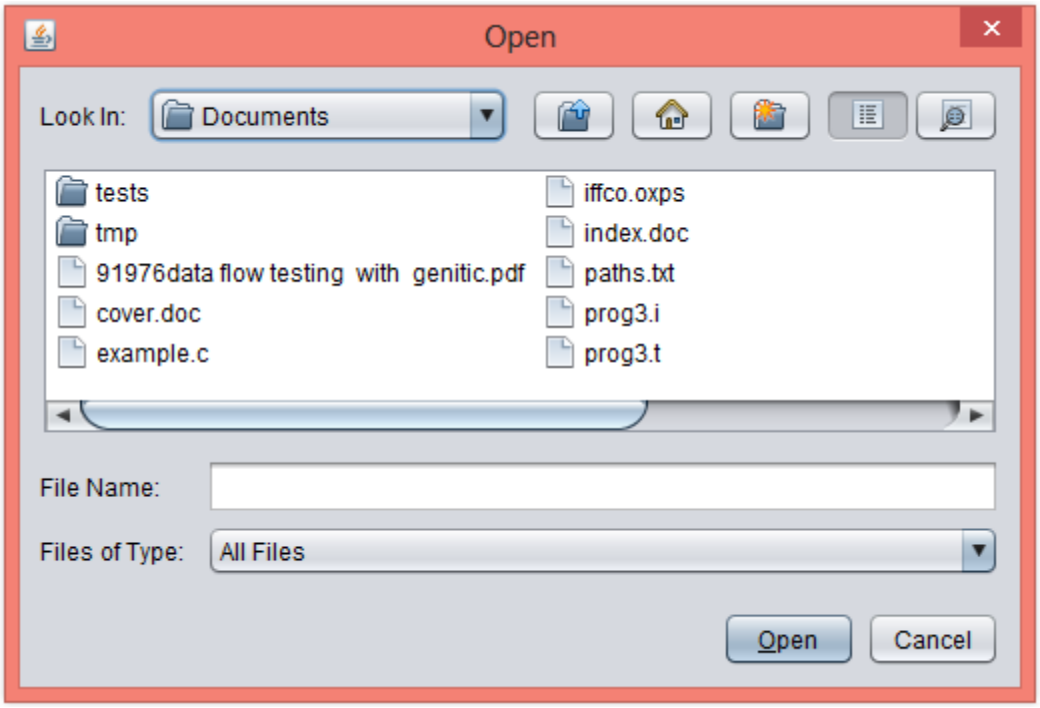

Figure 4.10. File Browser

Figure 4.11 the final dialog where Clicking on generate button create CFG of program.

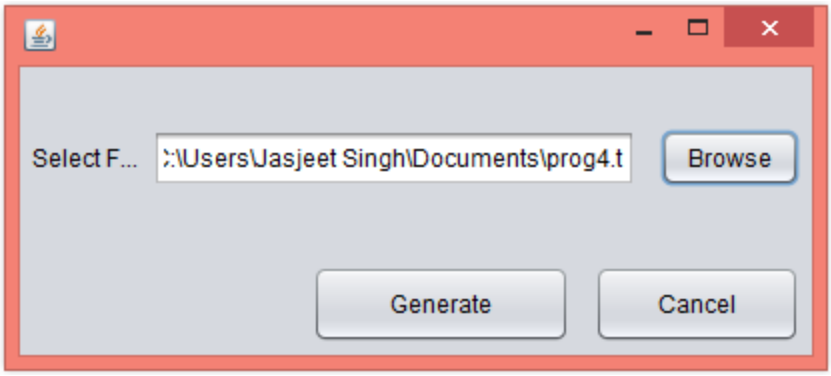

Figure 4.11. Input Dialog

Figure 4.12 shows CFG of program.

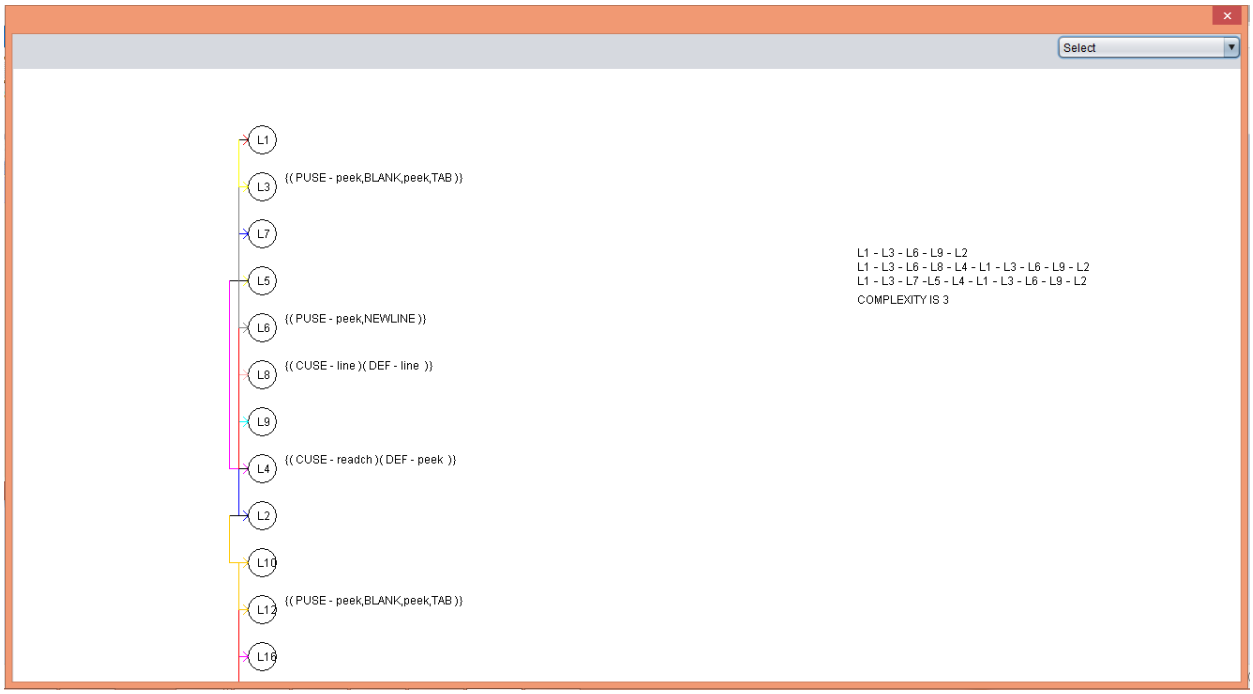

Figure 4.12. CFG Canvas

Refer figure 4.13, black circle shows where Criteria has to be selected for Test Case Generation.

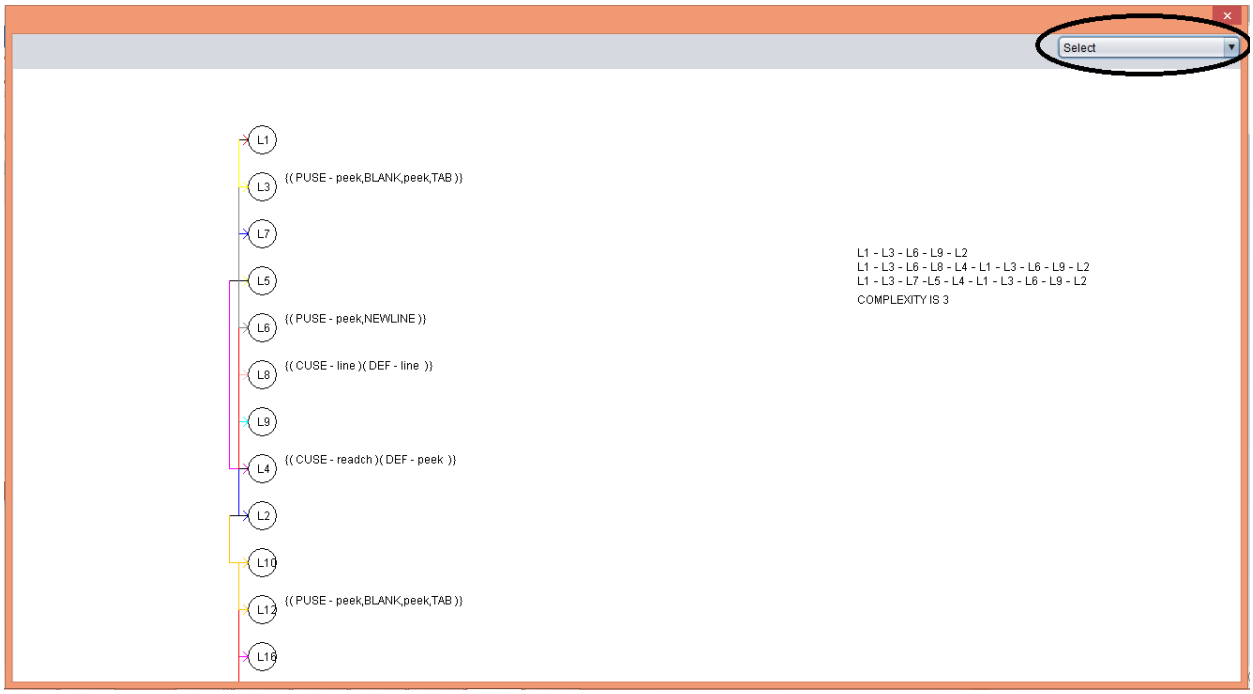

Figure 4.13. Selecting Criteria

Figure 4.14 shows the path and its Test Case.

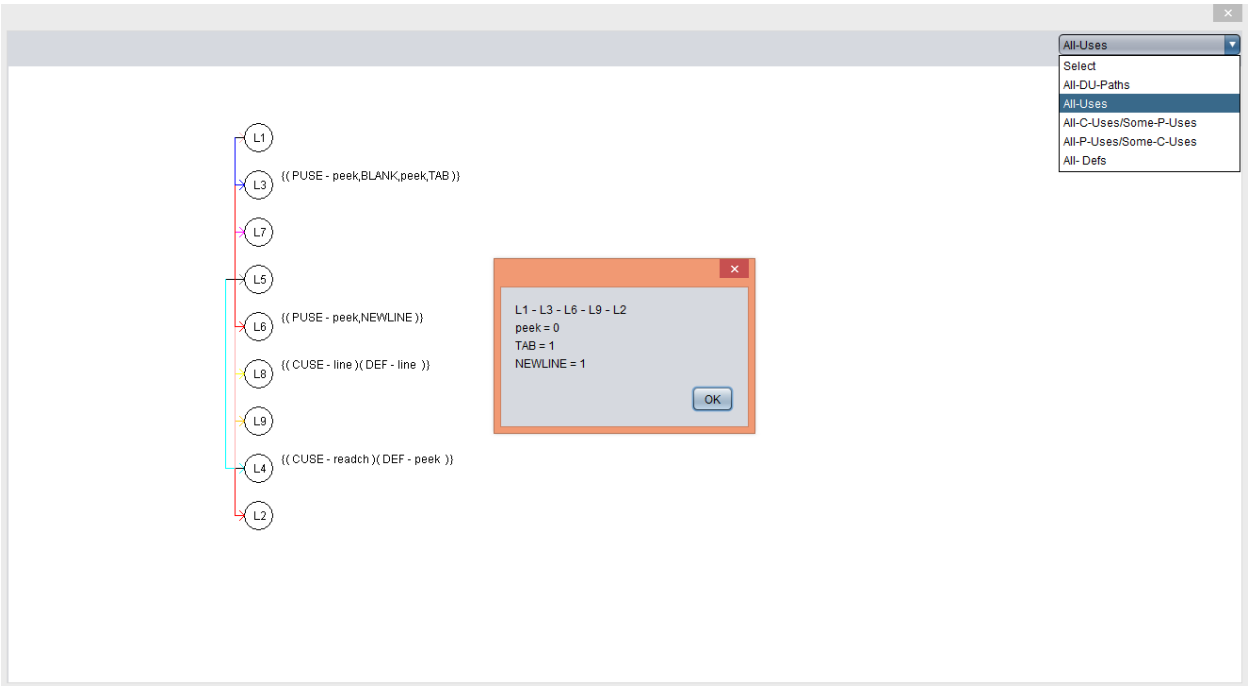

Figure 4.14. Test Case for Path

4.6. Results

Table 4.2. Results

Example	Paths	Test Case
1	$L_1 – L_3 – L_6 – L_9 – L_2$	*peek* = 0, *TAB* = 1, *NEWLINE* = 1, BLANK =1
	$L_1 – L_3 – L_6 – L_8 – L_4 – L_1 – L_3 – L_6 – L_9 – L_2$	*peek* = 0, *TAB* = 1, *NEWLINE* = 0, SPACE =0
	$L_1 – L_3 – L_7 – L_5 – L_4 – L_1 – L_3 – L_6 – L_9 – L_2$	*peek* = 0, *TAB* = 1, *NEWLINE* = 1,SPACE = 1, BLANK = 0

Table 4.2. Results (Continued)

Example	**Paths**	**Test Case**
2	$L_1 – L_3 – L_4 – L_6 – L_5 – L_2$	$i = 21$
	$L_1 – L_3 – L_4 – L_6 – L_5 – L_4 – L_6 – L_5 – L_2$	$i = 0$

Table 4.2 describes all the independent paths. These paths were generated using the implemented algorithm. These results validate the fusibility and credibility of the implemented algorithm.

Chapter 5

Conclusion and Future Scope

5.1. Conclusion

Data flow based testing is based on data definition, usage and killing of data items. In this thesis work, a review of data-flow based testing is carried out in procedure oriented languages. This "Automated Data Flow Testing" approach provides an easy and efficient way for the user to test his/her software written in C-Language automatically. This tool can be used to find the Control Flow Graph (CFG) of the given program and depict variables with nodes. An algorithm has been implemented to find paths in a graph. It also generates test cases for that path with Random Testing Technique. Automated DFT approach is developed to save the time and manual efforts in a large scale and it can be easily extended to develop other software testing techniques. Since, software testing is very intensive and expensive and accounts for a significant portion of software system development cost. If the testing process could be automated, the cost of developing software could be significantly reduced. Thus our tool deals with reducing manual effort, time and cost. In the testing framework, this research identifies various problems and issues and provides solutions to overcome these problems and to handle the related issues.

5.2. Future Scope

Following are the enhancement in which our work that can be carried out:

- Efficiency of time complexity issues of the proposed algorithm can be one of the future perspective.
- There is a scope of reducing the testing efforts using Artificial Intelligence and genetic algorithm in the proposed algorithm.
- There is scope of considering rest of Define/Use testing path selection criteria.

Bibliography

[1] B. Beizer: "Software *Testing Techniques,*" John Wiley and Dreamtech, 2002.

[2] Beizer, B.: "*Software System Testing and Quality Assurance,"* New York: Van Nostrand Reinhold Company, Inc, 1984.

[3] Boehm, B.W. :"*Software Engineering,*" IEEE Transactions on Computer SE-1(4):1226-1241, 1976.

[4] Boehm, B.W.: "*Characteristics of Software Quality,"* New York: American Elsevier, 1978.

[5] B.Y.Tsai, S.Stobart and N.Parrington: "*Employing data flow testing on object-oriented classes,*" IEEE Proceedings, 2001.

[6] Cheon, Y., Kim, M.Y., Perumandla, A.: "*A Complete Automation of Unit Testing for Java Programs,*" The 2005 International Conference on Software Engineering Research and Practice (SERP), Las Vegas, Nevada, USA, June 2005.

[7] Chilarege and Ram: "*Software Testing Best Practices,*" Center for Software Engineering, IBM Research, 1999.

[8] Elaine J. Weyuker: "*More Experience with Data Flow Testing,*" IEEE transaction on Software Eng., vol. 19, NO. 9, SEPTEMBER 1993.

[9] G. Myers: "*The Art of Software Testing,*" second edition, John Wiley & Son, 2004.

[10] Gregory Tassey: "*The Economic Impacts of Inadequate Infrastructure for Software Testing,"* Research Triangle Park, NC 27709, may 2002.

[11] Harsh Kumar Dubey, Prashant Kumar, Rahul Singh, Santosh K Yadav and Rama Shankar Yadav: "*Automated Data Flow Testing,*" in IEEE, 2012.

[12] Harrold M Jean.; Rothermel Gregg.: "*Performing data flow testing on classes,*" ACM SIGSOFT Software Engineering Notes , Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering SIGSOFT'94, pp 154 – 163, December 1994.

[13] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky and Hasan Ural: "*Data Flow Testing as Model Checking,*" 2002.

[14] Jain Deepak: "*Software Engineering Principle and Practices,*" First edition by Oxford University Press, 2009.

[15] J. A. Whittaker: "*What is Software Testing? And Why Is It So Hard?,"* IEEE Software, pp. 70-79, January 2000.

[16] J. C. Huang: "*An Approach to Program Testing,"* ACM Computing Surveys, pp.113-128, September 1975.

[17] J. R. Horgan and S. London: "*Data flow coverage and the C language,*" in Symposium on Testing, Analysis, and Verification, pp. 87–97, 1991.

[18] K. B. Gallagher and J. R. Lyle: "*Using program slicing in software maintenance,*" IEEE Trans. Software Eng., vol. 17, no. 8, pp. 751–761, 1991.

[19] K. K. Aggarwal and Yogesh Singh: "*Software Engineering,*" New Age International Publishers.

[20] Kropp, N P Koopman, P J Siewiorek: "*Automated Robustness Testing of the-Shelf Software Component,*" 28th Annual International Symposium on Fault- Tolerant Computing, 1995.

[21] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil: "*A formal evaluation of data flow path selection criteria,*" IEEE Trans. Software Eng., vol. 15, no. 11, pp. 1318–1332, 1989.

[22] Lee, C.: "*A Practitioner's Guide to Software Test Design*," STQE Publishing, 2004.

[23] McConnell, S.: "*Best Practices: Daily Build and Smoke Test,*" IEEE Software, vol. 13, no. 4, 143–144, July 1996.

[24] Moheb R. Girgis: "*Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm,"* 2005.

[25] M. Weiser*: "Program slicing,"* ICSE, pp. 439–449, 1981.

[26] M. Weiser: *"Program slicing,"* IEEE Trans. Software Eng., vol. 10, no. 4, pp. 352–357, 1984.

[27] Myers, J.P.: "*The Complexity of Software Testing,*" Software Engineering Journal, PP. 13-24, Jan 1992.

[28] P. C. Jorgensen: "*Software Testing: A Craftsman's Approach,*" CRC Press, 2nd ed., 2002.

[29] P. G. Frankl and E. J. Weyuker: "*An applicable family of data flow testing criteria,*" IEEE Trans. Software Eng., vol. 14, no. 10, pp. 1483–1498, 1988.

[30] Rakesh Shukla, David Carrington and Paul Strooper: "*A Passive Test Oracle Using a Component's API,"* Proceedings of the 12th Asia-Pacific Software Engineering.

[31] R. Ferguson and B. Korel: "*The changing approach for software test data generation*," ACM Transactions on Software Engineering Methodology, pp. 63–86, 1996.

[32] R. S. Pressman: "Software *Engineering: A Practitioner's Approach*," 3rd Edition, McGraw Hill, New York, pp. 559, 1992.

[33] Sandra Rapps and Elaine J. Weyuker: "*Data Flow Analysis Techniques for Test Data Selection*," 1982.

[34] S. Rapps and E.J. Weyuker: "*Selecting software test data using data flow information*," IEEE Transactions on Software Engineering, pp. 367-375, 1985.

[35] Ullman: "*Compilers: Principals, Techniques, and Tools*," 2nd Edition, Pearson, June 2006.

[36] Wallace, D.R. and R.U. Fujii: "*Software Verification and Validation: An Overview*," *IEEE Software*, pp. 10–17, May 1989.

List of Publication

Published

[1] Singh, J., and Garhwal, S., "*Systematic Review: Data Flow Testing*," International Journal of Advance Research in Computer Science and Software Engineering (IJARCSSE), Vol. 3, Issue. 6, July, 2013.

Communicated

[2] Singh, J., and Garhwal, S., "*Algorithm for Finding Paths in Control Flow Graph*," International Journal of Advance Research in Computer Science and Software Engineering (IJARCSSE), Vol. 3, Issue. 6, July, 2013.