Automation of Data Flow Based Testing

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Technology
in
Computer Science and Applications

Submitted By
Ravideep Lochav
(Roll No. 601003023)

Under the supervision of:
Mrs. Sunita Garhwal
Assistant Professor, SMCA

SCHOOL OF MATHEMATICS AND COMPUTER APPLICATIONS
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

CERTIFICATE

I hereby certify that the work which is being presented in the thesis entitled, "*Automation of Data Flow Based Testing*", 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 of 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 the thesis has not been submitted for award of any other degree of this or any other University.

Signature: Ravideep Lochav

(Ravideep Lochav)

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

Sunita

(Mrs. Sunita Garhwal)

Assistant Professor
SMCA
Thapar University
Patiala

Countersigned by

(Dr. S.S.Bhatia)
Head
School of Mathematics and Computer Applications
Thapar University
Patiala

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

Acknowledgement

Words are inadequate to express my gratitude to my thesis supervisor, **Mrs. Sunita Garhwal** Assistant Professor, School of Mathematics and Computer Applications, Thapar University, who has been very concerned and has aided for all the material essential for the preparation of this thesis report. She has helped me to explore this vast topic in an organized manner and provided me with all the ideas on how to work towards a research-oriented venture.

I am highly obliged to **Dr. S. S. Bhatia**, Head of Department, SMCA and **Mr. Singara Singh**, P.G. Coordinator for the motivation and inspiration that helped me for the thesis work.

I would also like to thank the staff members and my colleagues who were always there in the need and provided with all the help and facilities, which I required, for the completion of my thesis.

Most importantly, I would like to thank my parents and the Almighty for showing me the right direction out of the blue, to help me stay calm in the oddest of the times and keep moving even at times when there was no hope.

Ravideep Lochav
(601003023)

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 based testing is based on the concept that when a data is defined, used and killed. Some combination of definition, usage and killing of data leads to data anomalies. These anomalies must be removed for high quality software. In data flow based testing a def-use graph is constructed from the given segment of code. Using various strategies of data-flow based testing test suite can be generated.

Design by contract is a methodology used for improving the quality of software. In this thesis work design by contract combined with data flow based testing is applied on stack class. This approach causes reduction in du paths.

A software is designed that will determine data anomalies in the tested modules. It will also show that when a data is defined, killed and used for a particular variable.

List of contents Page No.

List of Figures and Tables

List of Abbreviations

SRS	Software Requirement Specification
SDS	Software Design Specification
U	Use
K	Killed
C-use	Computation Use
P-use	Predicate Use
DU	Definition-use
Def	Definition
ADUP	All-du Paths
AU	All Uses Strategy
APU	All P-uses
ACU	All C-uses
AD	All Definitions

Chapter 1
Introduction

This chapter contains the basic concepts of software testing and different types of testing.

1.1 Software Testing

Software testing is an important activity in the software development life cycle. Software testing is a standard method of assuring software quality. Testing is a process of executing a program with the intent of finding an error [12]. Testing software is far more complex 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]. There is a need of effective software testing for these kinds of applications. Usually a business application needs more reliability and functionality otherwise they will lose money, sales and customers.

1.1.1 Basic Terminology used in Software Testing

In this section, some basic definitions and terminology are explained [1]:

- **Error:** Human action that causes a software fault.
- **Fault:** Missing or incorrect code that may result in a failure.
- **Failure:** Manifested inability of a system or component to perform a required function within specified limits.
- **Test case:** Set of inputs, execution conditions and expected results developed for a particular objective.
- **Test suite:** It is a collection of test cases, typically related by a testing goal or an implementation dependency.
- **Test driver:** Class or utility program that applies test cases to an implementation under test.
- **Test harness:** System of test drivers and other tools that support test execution.
- **Stub:** Partial temporary implementation of a component.

1.1.2 Need for Testing

Testing is an important activity of software development. Software testing is needed to verify and validate that the software that has been built to meet these specifications [13]. Testing does not reveal error if the expected and generates outcomes are same. Testing does not reveal presence of error but it shows absence of errors [7]. Testing enhances the integrity of a system by detecting deviations in design and errors in the system. Testing aims at detecting error-prone areas. This helps in the prevention of errors in a system. Testing also adds value to the product by conforming to the user requirements.

1.2 Objectives of Software Testing

The main objective of software testing is to prove that the software product meets a set of pre-established acceptance criteria under a prescribed set of environmental circumstances [1]. Following are the steps to achieve good quality software [1]:

1. In the first step, we check that the requirements specification used for software design is correct.
2. In second step, we check that the design and coding meets the requirements correctly. Correctness means that the software meets both functional and non-functional requirements. Non-functional requirement include timing, performance and other criterion.

Software test cases are designed with the intent to find error. System acceptance criteria usually involve hardware, procedures and operators so that acceptance tests involve more than just the software. Following are the main objectives of software testing [1]:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet undiscovered error.
3. A successful test is one that uncovers an as-yet undiscovered error.

1.3 Testing Principles

Davis [4] suggests following are the testing principles:

1. **Test suite should be traceable to customer requirements:** The objective of software testing is to uncover errors. It follows that the most severe defects (from

the customer's point of view) are those that cause the program to fail to meet its requirements.

2. **Early plan for software testing:** Test planning can begin as soon as the requirements model is complete. All tests can be planned and designed before any code has been generated.
3. **Pareto principle:** It 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.
4. **Testing should start with small modules and progress up for performing their integration:** 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.
5. **Exhaustive testing is not possible:** The number of path permutations for even a moderately sized program is exceptionally large. Thus it is impossible to execute every combination of paths during testing.
6. **Testing should be conducted by third party:** If a software developer is testing the software, his/ her intention is to prove that the software is working correctly that is designed by him/her. Thus testing should be conducted by an independent third party.

1.4 Characteristics of a Good Test

Following are the characteristics of good test cases [13]:

1. **A good test has a high probability of finding an error:** Test case should be designed by understanding the software and try to develop a picture of the software that how the system can fail.
2. **Non-redundant test cases:** Software testing is a very expensive task. For reducing the cost of software testing, every test case is generated with the intent to find different types of error. If two test cases reveal same kind of errors, we can delete one of them from our test cases.
3. **A good test should be neither too simple nor too complex:** Although it is sometimes possible to combine a series of tests into one test case, the possible

side effects associated with this approach may mask errors. In general, each test should be executed separately.

1.5 Software Testing Techniques

Following are two types of software testing techniques:

1. Static Testing
2. Dynamic Testing

1.5.1 Static Testing

Static testing 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.

Following are the objectives of the static testing [8]:

1. Validating the requirement specifications.
2. Looking for omissions, inconsistencies, redundancies in all documents and source code.
3. To ensure that the documents of design and coding conform to the specification.

1.5.2 Dynamic Testing

Dynamic testing 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].

1.6 Types of Dynamic Testing

Basically, there are two forms of dynamic software testing:

1. Functional or Black-Box Testing
2. Structural or White-Box Testing

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.

In functional testing, only observation of the output for certain input values is used as test cases. In structural testing the knowledge to the internal structure of the code is used to find the number of test cases required to guarantee a given level of test coverage. Only white-box testing is not sufficient to find out all the errors of the program. Similarly black-box testing is not sufficient to find out all the error of the program. So we need a combination of white-box and black-box testing.

1.6.1 Black-Box Testing or Functional Testing

Black-box testing is also known as behavioral testing. It focuses on the functional requirements of the software. Black-box testing is the testing of a piece of software without regard to its underlying implementation. The goal of black-box testing is to demonstrate that the software being tested does not adhere to its external specification. Black-box testing is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. Black-box testing attempts to find errors in the following categories [8]:

1. Incorrect or missing functions
2. Interface errors
3. Errors in data structures or external data base access
4. Behavior or performance errors
5. 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.

1.6.2 White-Box Testing or Structural Testing

White-box testing is also known as glass-box testing. In white-box testing, test cases are designed that uses the control structure of the procedural design to derive test cases [8]. Using white-box testing methods, the software engineer can derive test cases that [1]:

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

Therefore, white-box testing is the testing of the underlying implementation of a piece of software without regard to the specification for that piece of software. The goal of white-box testing of source code is to identify infinite loops and paths through the code which should be allowed but which cannot be executed and dead (unreachable) code.

Following are the advantages of white box testing[8]:

1. Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed.
2. Often, a logical path is not likely to be executed when, in fact, it may be executed on a regular basis.
3. Typographical errors are random.

White-box testing techniques can be future divided into control flow and data-flow based testing [1].

1.7 Control flow testing

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

1. Statement coverage: Test cases are designed so that using these test cases each statement of the modules will be exercised at least once.
2. Branch coverage: Test cases are designed so that using these test cases each branch of the modules will be exercised at least once.
3. 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.
4. Branch /condition coverage: both branch and condition coverage achieved.

5. 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.
6. Loop coverage: Loop coverage technique states that test cases must be written to test the loop counters.

1.8 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 [15]:

1. All definition-uses (all du paths): It requires that every definition of every variable at all du paths should be exercised under the test.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.

1.9 Different Categories of White-Box Testing

Acceptance Testing: Acceptance tests are created from user requirements. Acceptance tests are a form of black-box testing. Each acceptance test represents some expected result from the system. Customers are responsible for verifying the correctness of the acceptance tests and reviewing test scores to decide which failed tests are of highest priority. Acceptance tests are also used as regression tests prior to a production release [10]. A principal purpose of acceptance testing is that, once completed successfully, and provided certain additional acceptance criteria are met, the sponsors will then sign off on the system as satisfying the contract and deliver final payment.

Alpha Testing: Alpha testing takes place at developer's sites, and involves testing of the operational system by internal staff, before it is released to external customers [10]. Testing of an application when development is near completion, Minor design changes may still be made as a result of such testing. This testing is typically done by end-users or others, but not by programmers or testers.

Beta Testing: Beta testing takes place at customers sites, and involves testing by a group of customers who use the system at their own locations and provide feedback, before the system is released to other customers Testing is done when development and testing are essentially completed and to find final bugs and problems before final release[3]. It is typically done by end-users or others, not by programmers or testers.

Mutation Testing: Mutation testing is done by selecting a set of mutation operators and then applying them the source program one at a time for each applicable piece of the source code. The result of applying one mutation operator to the program is called a mutant. If the test suite is able to detect the change, then the mutant is said to be killed. Mutation testing is a powerful technique for unit-level software testing that as usually implemented, is computationally expensive [8].

Control Testing: Control is a management tool to ensure that processing is performed in accordance to what management desire or intents of management. The objective of the control testing is accurate and complete data, authorized transactions, process meeting the needs of the user. Testers should have negative approach i.e. they should determine or anticipate what can go wrong in the application system [3]. Develop risk matrix, which identifies the risks, controls, segment within application system in which control resides.

1.10 Thesis Outline

This thesis is organized into five chapters. Chapter 1 gives the description about the basic concepts of software testing and its types. Chapter 2 describes the data flow based testing. Chapter 3, describes the motivation behind the thesis, discusses the problem statement and its objectives. Chapter 4, describe the review of data flow based testing, data contract and automaton of data-flow based testing. Chapter 5 summarizes the conclusions drawn from the thesis along with the directions regarding the future work.

Chapter 2

Data Flow Based Testing

This chapter describes the various works that have been done in field of Data-Flow Testing.

2.1 Data Flow Testing

Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects [2]. For example, paths are chosen to assure that every data object has been initialized prior to use or that all defined objects have been used for something. Data-flow testing uses the control flow graph to explore the unreasonable things that can leads to data anomalies.

2.1.1 Data Object and Their Usage

Data objects can be created, used and killed. Data can be used in computation (C-use) or in control flow predicates (P-use).

Following are the notation used with data flow based testing:

1. **Data Definition:** It includes data definition, creation, initialization etc. Data is defined explicitly, when it appears in a data declaration or implicitly when it appears on the left hand side of the assignment [10]. File pointer is defined while it is initialized during opening of file. It can be used for opening a file.
2. **Killed or undefined (k):** It includes data killing, undefined, released etc. An object is killed or undefined, when it is released or otherwise made unavailable, when its contents are no longer known with certitude, release of dynamically allocated objects back to the availability pool, return of records [10]. An assignment statement can kill and redefine immediately. For example: If variable a is defined previously and assignment statement takes places as a=17 cause the old value of a is killed and a new value is defined for a.
3. **Usage (u):** Data can be used as C-use or P-use where C stands for computations and P stands for predicate. A variable is used for computation when it appears on

the right hand side of an assignment statement or a file record is read or written. A variable is used as P-use, if it appears directly in a predicate statement.

Example 2.1: Consider following program segment.

Table 2.1: Sample program for C-use and P-use [14]

Statement no.	Statement
1.	read (x, y);
2.	z = x + 2;
3.	if (z < y)
4.	w = x + 1;
5.	Else
6.	y = y + 1;
7.	print (x, y, w, z);

C-use and P-use of each definition of the above program is given in table 2.2.

Table 2.2: C-Use and P-use [14]

Statement	**Def**	**C-use**	**P-use**
1	x, y	-	-
2	z	x	-
3	-	-	z, y
4	w	x	-
6	y	y	-
7	-	x, y, w, z	-

2.2 Data Flow Anomalies

Data anomaly is represented by a two-character sequence of actions. Following are the combination that occur with definition, use and killed [1].

1. **dd:** When a definition of a data appear after its definition without any use. It is harmless but suspicious anomalies. There is no need to define a data twice without using it.

2. **dk:** When an object is defined and subsequent killed without its use. It is probably a bug.
3. **du:** When an object is defined and subsequently used. It will be a normal situation and not considered as anomalies.
4. **kd:** When an object is killed and then redefined. It is a normal situation.
5. **kk:** When an object is killed twice without any re-definition or usage then it is harmless.
6. **ku:** When an object is killed and subsequent used without re-definition. It is a bug because the object does not exist.
7. **ud:** It occurs when an object is used and subsequently redefinition occurs. This normally occurs in a programming language and it is not considered as anomalies.
8. **uk**: When an object is killed after its usage. It is a normal situation.
9. **uu:** When an object is use and in subsequent statement same definition is again used. It is a normal situation.

2.3 Data Flow Anomaly State Graph

Data flow anomaly model prescribes that an object can be in one of four distinct states [1]:

1. **K**: undefined, previously killed, does not exist
2. **D**: defined but not yet used for anything
3. **U**: has been used for computation or in predicate
4. **A**: anomalous

These capital letters (K, D, U, A) denote the state of the variable and should not be confused with the program action, denoted by lower case letters.

2.3.1 Forgiving Data Flow Anomaly Graph

In unforgiving data flow anomaly graph, once a variable becomes anomalous it can never return to a state of grace [1, 7].

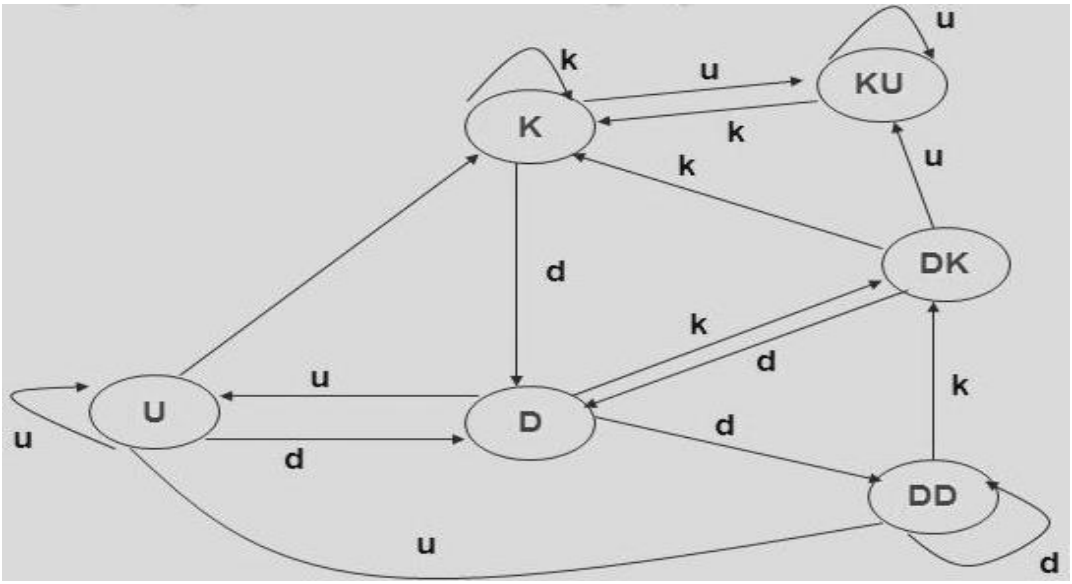

Figure 2.1: Forgiving data flow anomaly graph [7]

2.3.2 Unforgiving Data Flow Anomaly Graph

Forgiving data flow anomaly graph is an alternate model where redemption (recover) from anomalous state is possible [3, 7].

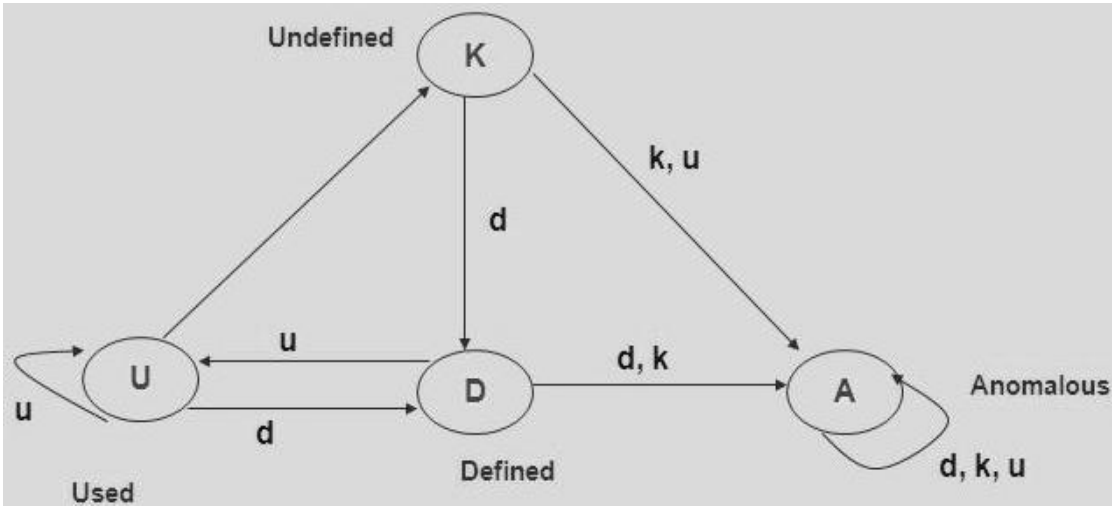

Figure 2.2: Unforgiving data flow anomaly graph [7]

2.4 Strategies of Data Flow Testing

Following are the definitions used in data flow base testing:

1. **Definition-clear path:** Definition-clear path with respect to a variable X, is a connected sequence of links in def-use graph such that variable X is defined on the first link and not redefined or killed on any subsequent link of that path segment[1].

2. **Loop-free path:** It is a path segment for which every node in it is visited at most once. It is either loop free or if there is a loop, only one node is involved [1].
3. **Simple Path Segment:** It is a path segment in which at most one node is visited twice. A simple path segment is either loop free or if there is a loop, only one node is involved [1].
4. **Du path:** A du path from node i to k is a path segment such that if the last link has a computational use of X, then the path is simple and definition clear, if the second last node is j that is the path and link has a predicate uses then the path from i to j is both loop free and definition clear [1].

Data-flow testing strategies are structural strategies. Data-flow strategies require data-flow link weights (d, k, u, c, and p). Data flow testing strategies are based on selecting test path segments (also called sub-paths) that satisfy some characteristic of data flows for all data objects. Various types of data flow testing strategies [7] in decreasing order of their effectiveness are:

1. **All du Paths (ADUP):** It requires that every du path from every definition of every variable to every use of that definition be exercised under some test. All du paths strategy is a strong criterion, but it does not take as many tests as it might seem at first because any one test simultaneously satisfies the criterion for several definitions and uses of several different variables.
2. **All Uses Strategy (AU):** All uses strategy is that at least one definition clear path from every definition of every variable to every use of that definition be exercised under some test.
3. **All P-uses/some C-uses strategy (APU+C) :** For every variable and every definition of that variable, include at least one definition free path from the definition to every predicate use; if there are definitions of the variables that are not covered by the above prescription, then add computational use test cases as required to cover every definition.
4. **All C-uses/some P-uses strategy (ACU+P) :** All C-uses/some P-uses strategy (ACU+P) is to first ensure coverage by computational use cases and if any definition is not covered by the previously selected paths, add such predicate use cases as are needed to assure that every definition is included in some test.

5. **All Definitions Strategy (AD):** All definitions strategy asks only every definition of every variable is covered by at least one use of that variable, be that use a computational use or a predicate use.
6. **All Predicate Uses (APU), All Computational Uses (ACU) Strategies:** All predicate uses strategy is derived from APU+C strategy by dropping the requirement that we include a C-use for the variable if there is no P-use for the variable. All computational uses strategy is derived from ACU+P strategy by dropping the requirement that we include a p-use for the variable if there are no c-uses for the variable.

2.5 Comparisons of different criterion of data-flow based testing

Among different criterion of data-flow based testing some criterion are comparable with each other as shown in figure 2.3. Arrows denote that the strategy at the arrow's tail is stronger than the strategy at the arrow's head. The right-hand side of this graph, along the path from "all paths" to "all statements" is the more interesting hierarchy for practical applications.

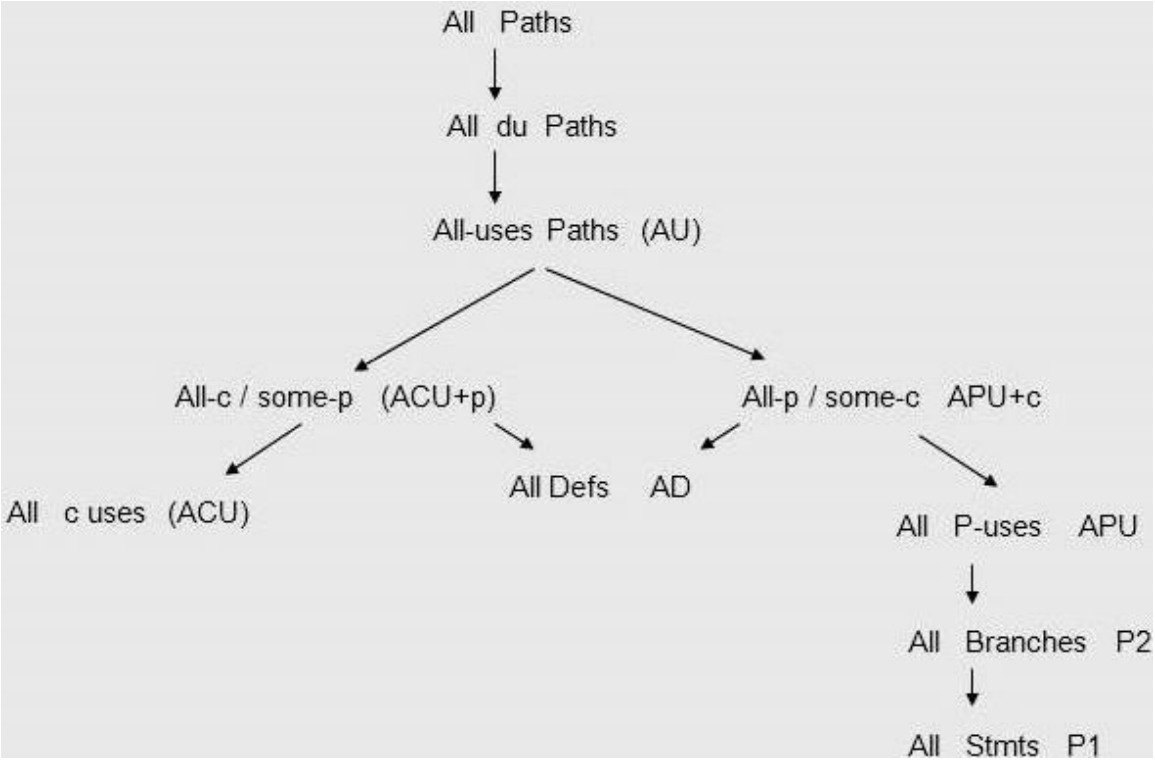

Figure 2.3: Ordering of the data-flow testing strategies [1]

In the following section comparison between different criterions is explained.

1. **All du Paths→ All uses**: All du paths are a stronger testing criterion than all uses. For example in figure 2.4 {(l, 2, 4, 5, 7), (1, 3, 4, 6, 7)} satisfies all-uses, but not all du paths, since it does not include the path (1, 2, 4, 6, 7), which includes a def clear path with respect to y from node 2 to node 7. This example shows that all du paths is a stronger criterion than all uses. All du paths require that test cases included such that it cause certain combinations of path segments to be traversed, whereas all uses would not require these combinations.
2. **All uses→ All P-uses/Some C-uses**: In def-use graph of figure 2.5 {(l, 2, 3, 4), (1, 3, 5)} satisfies all P-uses/some C-uses, but not all-uses, since there is no def clear path with respect to y from the def of y in node 2 to the C-use of y in node 5. The example makes clear the advantage of the all uses criterion over all the P-uses/some C-uses. By not requiring that every C-use be exercised, we might well miss an error involving an incorrect treatment of positive even values for x.

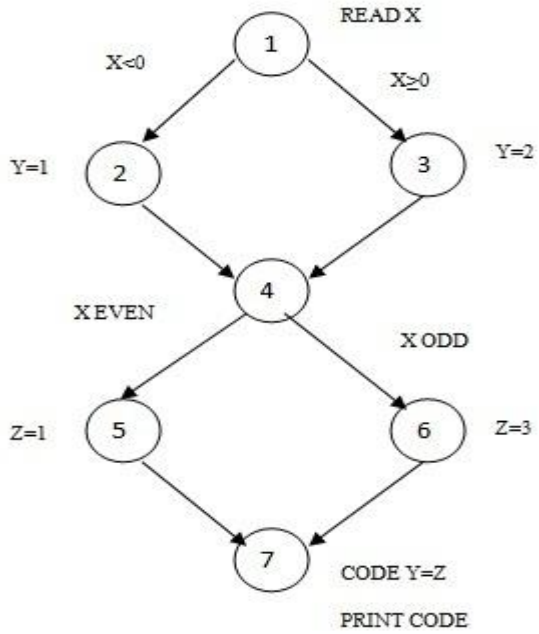

Figure 2.4: Comparisons between all du paths and all uses [21]

3. **All uses→ All C-uses/Some P-uses**: Consider the def-use graph of figure 2.6 {(l, 3)} satisfies all C-uses/some P-uses, but does not satisfy all-uses since there is no path containing the P-use of x in edge (1, 2). This is example of the problem of not

requiring that test data be included which cause every node and edge to be traversed. Any problems in node 2 or edge (1, 2) would go undetected.

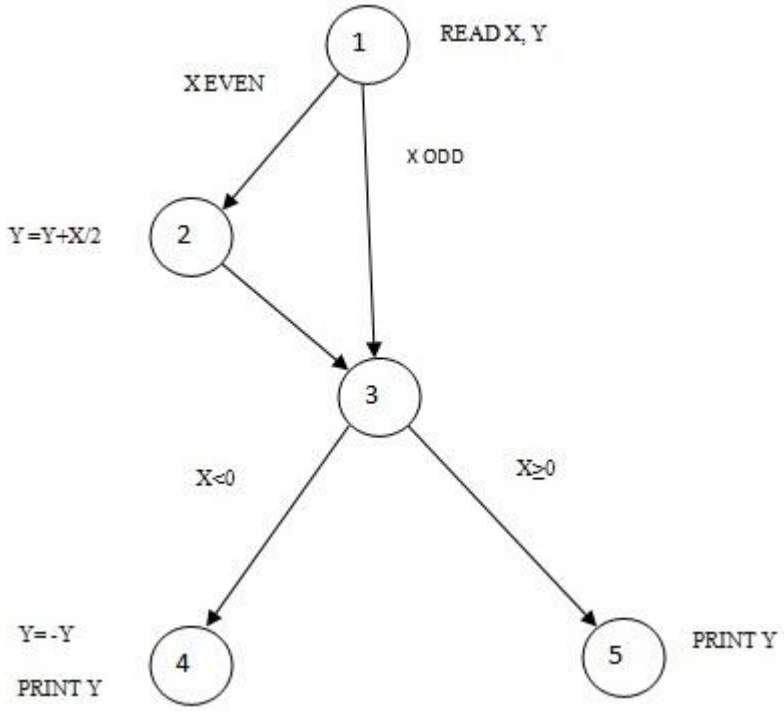

Figure 2.5: Comparisons between all uses and all P-uses/some C-uses [15]

4. **All C-uses/Some P-uses→ All defs and All P-uses/Some C-uses→ All defs:** Consider the def-use graph of Figure 2.7 {(1, 2)} satisfies all defs, but does not satisfy all C-uses/some P-uses or all P-uses/some C-uses. Again, as in the previous case, problems which occur in unexecuted nodes or un-traversed edges would go undetected.

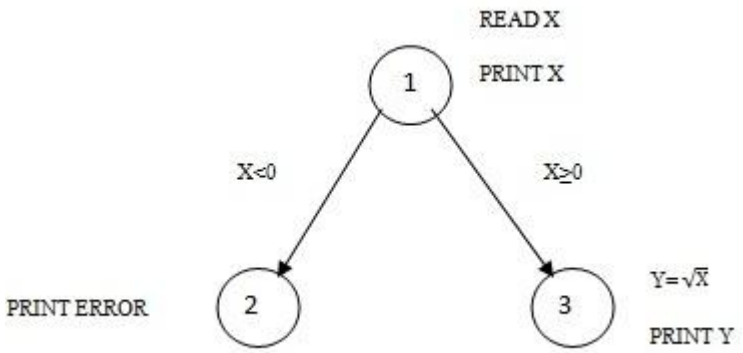

Figure 2.6: Comparisons between all uses and all C-uses/some P-uses [15]

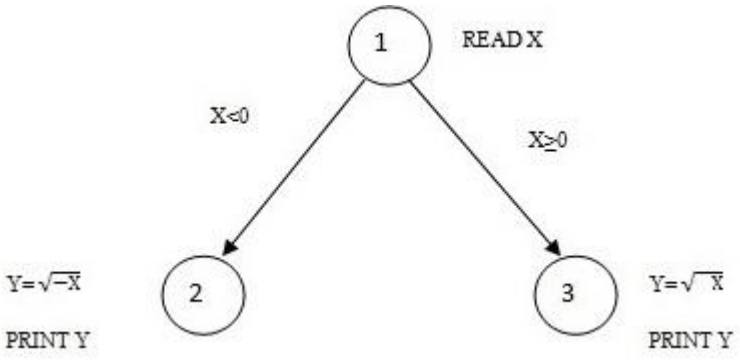

Figure 2.7: Comparisons between all def and all P-uses/some C-uses [15]

5. **All P-uses/Some-C-uses→ All P-uses:** Consider the def use graph of figure 2.8 {(l,2,3,5), (1,3,4)} satisfies all P-uses, but not all P-uses/some C-uses, since dpu(y, 2) is empty, but there is no def-clear path w.r.t. y from the definition of y in node 2 to the C-use of y in node 4. This example illustrates the original problem we identified: it is possible to traverse all edges and still have a definition remain unused. In this case, an incorrect assignment to y in node 2 would presumably go undetected if the all P-uses criterion was chosen.

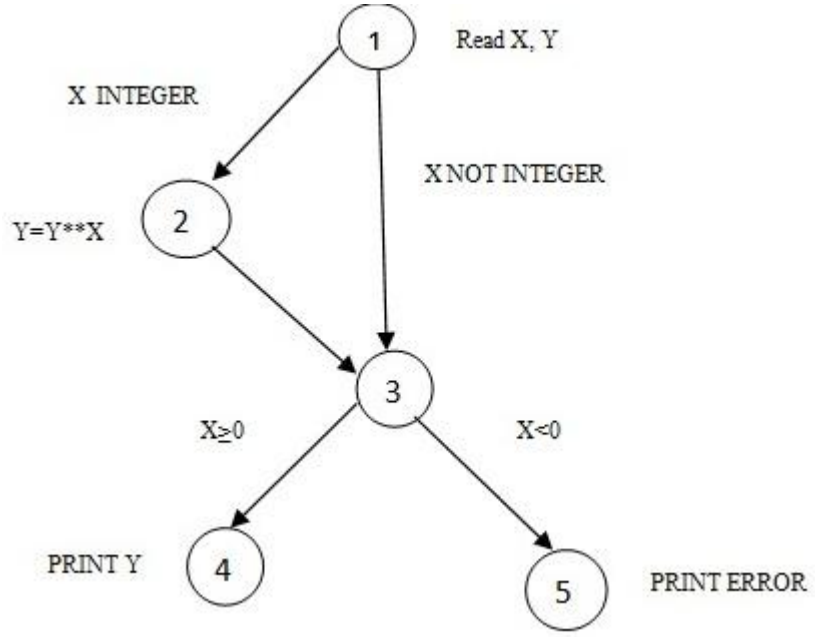

Figure 2.8: Comparisons between All P-uses/Some C-uses [15]

6. All P-uses→ All Edges: Consider the graph of Fig. 2.9 {(l, 2, 3, 2, 4)} satisfies all edges

but not all P-uses, as it does not include a def-clear path w.r.t. x from the def of x in node

1 to edge (2, 4).

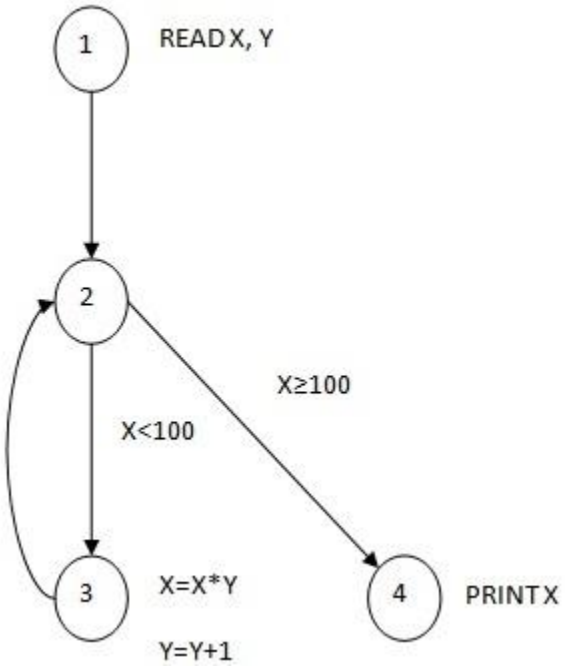

Figure 2.9: All P-uses→ All edges [15]

7. **All Edges→All Nodes:** Consider the graph of Fig. 2.10 {(l, 2, 3)} satisfies all nodes but not **all** edges.

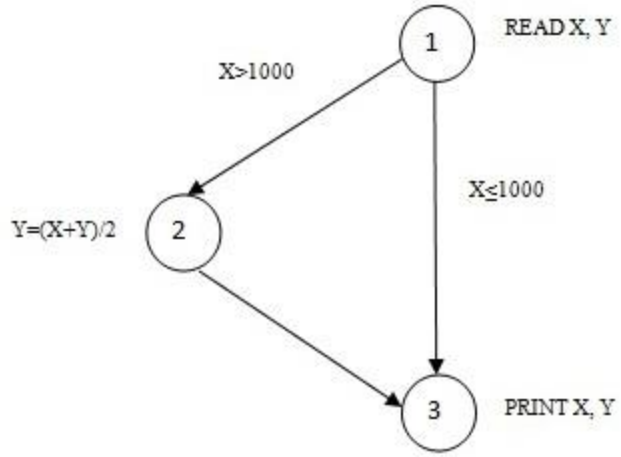

Figure 2.10: All Edges→ All Nodes [15]

8. **All C-uses/Some P-uses and All P-uses/Some C-uses are incomparable:** The graph of Fig. 2.4 demonstrates that all P-uses/some C-uses do not include all C-uses/some P-uses. {(I, 2, 3, 4), (1, 3, 5)} satisfies all P-uses/some C-uses, but not all C-uses/some P-uses, since there is no def-clear path w.r.t. y from the def of y in node 2 to the C-use of y in node 5. Similarly, the graph of Fig. 2.6 demonstrates that all C-

uses/some P-uses does not include all P-uses/some C-uses. {(l, 3)} satisfies all C-uses/some P-uses but not all P-uses/some C-uses, since there is no path containing the P-use of x in edge (1, 2).

Following criterions are not comparable:

1. All defs and All P-uses are incomparable.
2. All defs and All Edges are incomparable.
3. All defs and All nodes are incomparable
4. All C-uses/Some P-uses and All P-uses are incomparable.
5. All C-uses/Some P-uses and All Edges are incomparable.
6. All C-uses/Some P-uses and all nodes are incomparable.

Note that although ACU+P are stronger than ACU, both are incomparable to the predicate-biased strategies. Note also that "all definitions" is not comparable to ACU or APU.

The genetic algorithm technique is guided by the data flow dependencies in the program to search for test data to fulfill the all-uses criterion. The approach can be used in test data generation for programs with/without loops and procedures. Moheb R. Girgis[10] proposed genetic algorithm which accepts as input an instrumented version of the program to be tested, the list of def use paths to be covered, the number of input variables, and the domain and precision of each input variable. It accepts the genetic algorithm parameters as population size, maximum number of generations, and probabilities of the crossover and mutation. The algorithm produces a set of test cases, the set of def-use paths covered by each test case, and a list of uncovered def-use paths, if any.

Hyoung Seok Hong et al. [7] investigate a model checking-based approach to data flow testing and characterize data flow oriented coverage criteria in temporal logic such that the problem of test generation is reduced to the problem of finding witnesses for a set of temporal logic formulas.

Program testing is the most commonly used method for demonstrating that a program actually accomplishes its intended purpose. The testing procedure consists of selecting elements from the program's input domain, executing the program on these test cases and

comparing the actual output with the expected output. Sandra Rapps and Elaine J. Weyuker [15] examines a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. It is argued that currently used path selection criteria which examine only the control flow of a program are inadequate.

Harsh Kumar Dubey et al. [6] developed an "Automated Data Flow Testing" tool which 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 anomalies in the program and it can be performed white box testing in an efficient manner. Automated DFT tool 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 automated data-flow testing tool deals with reducing manual effort, time and cost.

Chapter 3

Problem Statement and Objectives

This chapter includes the thesis problem statement and its objectives.

3.1 Problem Statement

Software testing is an important phase of software development life cycle. Testing is a process of executing a program with the intent of finding an error. Testing can be manual, automated, or a combination of both. Manual testing of the software is inefficient and costly. If the testing process could be automated, the cost of developing software could be significantly reduced.

Data flow based testing is a white box testing. Data flow testing has been applied to generate test cases for testing classes using data flow criteria, but this is a difficult task. Moreover, some of data flow test cases generated may be unworkable. Whenever we are using data flow based testing, several anomalies can occurs like variable is used but never defined, variable is defined but not used or referenced or variable is defined twice without any use of first. These anomalies are results of error in the program and cannot be detected by dynamic testing techniques. These anomalies will be identified by static analysis of code.

Basic unit of object oriented testing is a class. Dataflow testing is a code based testing technique that uses the dataflow relations in a program to guide the selection of tests. Existing dataflow testing techniques can be applied both to individual methods in a class and to methods in a class that interact through messages. Existing techniques do not consider the dataflow interactions that arise when users of a class invoke sequences of methods in an arbitrary order. There is a need to work on design by contract and data flow based testing techniques.

There is a need of automaton of data-flow testing which will helps us in generating the test cases and it will finds the anomalies of the program. A tool is designed which help us in improvement of quality of a software and causes reduction in cost. This tool focuses on where a variable receives input, where it is used and finds out if there is any data anomalies presented or not.

3.2 Thesis Objectives

Following are the objectives of the thesis:

1. Investigation of the data flow based testing and their different criterion in procedural language.
2. Use of design by contract methodology in data flow based testing.
3. Design of a tool that will help in data flow based testing.

Chapter 4

Implementation and Experimental Results

This chapter describes how data flow testing is applied on the program. How data contract and data flow based testing used to generate du paths.

4.1 Review of Data Flow Based Testing

We have considered reverse floyd triangles program and generated test cases or test path based on different criterion of data flow based testing.

```
int i,j,k;
i=7;
for (j=3;j≥0;j--)
{
for (k=0;k≤j;k++)
{
printf (“%d”,(i+k));
}
i = i-j;
printf(“\n”);
```

Figure 4.1: Reverse Floyd Triangle

For the program given in figure 4.1, first we generate def-use graph as shown in figure 4.2. First we identify that for variable definitions. Corresponding to each variable definition their computational and predicate uses are identified as shown in table 4.1.

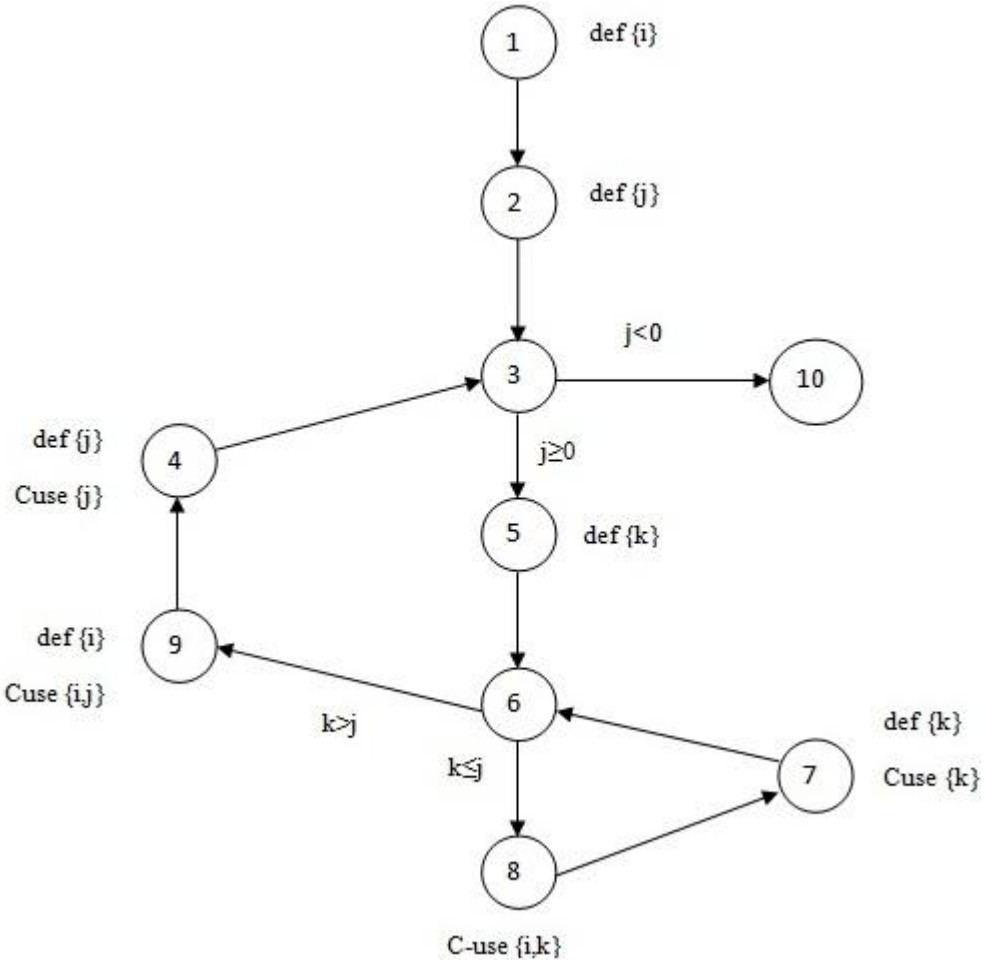

Figure 4.2: definition-use Graph

dcu stands for definition computation use and it includes the nodes at which a particular definition is used. dpu stands for definition predicate use and it includes the edge which is going to execute by considering a particular definition. dcu and dpu are computed using table 4.1.

Computation of dcu and dpu:

dcu(i,1) = {8,9}

dcu(j,2) = {4,9}

dcu(j,4) ={4,9}

dcu(k,5) = {7,8}

dcu(k,7) = {7,8}

dcu(i,9) = {8,9}

dpu(i,1) = { }

dpu(j,2) = {(3,10),(3,5),(6,9),(6,8)}

dpu(j,4) = {(3,10),(3,5),(6,9),(6,8)}

dpu(k,5) = {(6,9),(6,8)}

dpu(k,7) = {(6,9),(6,8)}

dpu(i,9) = { }

Table 4.1: Nodes with definition and their uses [6]

Node	C-use	def	Edge	P-use
1	--	{i}	(3, 10)	{j}
2	--	{j}	(3, 5)	{j}
3	--	--	(6, 8)	{k ,j}
4	{j}	{j}	(6, 9)	{k, j}
5	--	{k}	-	-
6	--	--	-	-
7	{k}	{k}	-	-
8	{i, k}	--	-	-
9	{i, j}	{i}	-	-
10	--	--	-	-

Test path based on the path testing criterion:

1. **All Nodes:** In all nodes criterion, test cases are generated to ensure that each node of the def-use graph is to be executed.
   Test case for all nodes: T_1= {1,2,3,5,6,8,7,9,4,3,10}
2. **All Edges:** In all edges criterion, test cases are generated to ensure that each edge of the def-use graph is to be executed.
   Test case for all edges : T_1= {1,2,3,5,6,8,7,9,4,3,10}
3. **All def:** In all def. criterions, test cases are generated to ensure that corresponding to each definition of each variable at least one C-use or P-use will be covered.
   Test case for all def. : T_2= {1,2,3,5,6,8,7,6,9,4,3,5,6,8,7,6,9,4,3,10}

4. **All P-use:** In all P-use criterions, test cases are generated to ensure that corresponding to each definition of each variable each P-use of the definition will be covered.

   Test case for all P-use:

   T_1={1,2,3,4,5,6,7,8,9,10}

   T_2={1,2,3,4,5,6,8,7,6,9,4,3,5,6,8,7,6,9,4,3,10}

   T_3={1,2,3,10}

   T_4={1,2,3,5,6,9,4,3,10}

5. **All P-use/Some C-use:** In all P-use/some C-use criterions, test cases are generated to ensure that corresponding to each definition of each variable each P-use of the definition will be covered. If any definition of the variable does not have P-use then at least one c-use of that will be covered.

   Test case for all P-use/some C-use: T_1, T_2, T_3 and T_4 test cases are required to ensure all P-use and some C-use.

6. **All C-use/Some P-use:** In all c-use/some p-use criterions, test cases are generated to ensure that corresponding to each definition of each variable each c-use of the definition will be covered. If any definition of the variable does not have c-use then at least one p-use of that will be covered.

   Test case for all C-use/some P-use: T_1 and T_2 test cases are required to ensure all P-use and some C-use.

7. **All Uses:** In all-uses criterion, corresponding to each definition of each variable every C-use and P-use will be covered.

   Test case for all use: T_1, T_2, T_3 and T_4 test cases are required to ensure all uses.

8. **All Path:** Following are test cases required to ensure the all-path criterion:

   {1, 2, 3, 10}

   {1, 2, 3, 5, 6, 9, 4, 10}

   {1,2,3,5,6,9,4,3,5,6,9,4,10}

   {1,2,3,5,6,9,4,3,5,6,8,7,6,9,4,10}

   {1,2,3,5,6,9,4,3,5,6,8,7,6,8,7,6,…,9,4,10}

We can iterate on the 2 loops as many times as possible to cover all path criteria. Another program is consider as input and using data flow based testing test path are generated based on different criterion.

```
int  i=1, j=1, k= 1;
for ( i=1;i≤3;i++)
{
for(j=1;j≤3;j++)
{
for(k=1;k≤3;k++)
printf("%d%d%d",i,j,k);
```

Figure 4.3: Sample program to generate test cases using data flow based testing Def./use graph is shown in figure 4.4. Table 4.2 represent, corresponding to each node what node is defined.

Table 4.2: Node, C-Use, def, Edge and P-use [6]

Node	C-use	def	Edge	P-use
1	--	{i,j,k}	(3,5)	{i}
2	--	{i}	(3,12)	{i}
3	--	--	(6,8)	{j}
4	{i}	{i}	(6,4)	{j}
5	--	{j}	(9,11)	{k}
6	--	--	(9,7)	{k}
7	{j}	{j}	--	--
8	--	{k}	--	--
9	--	--	--	--
10	{k}	{k}	--	--
11	{i,j,k}	--	--	--
12	--	--	--	--

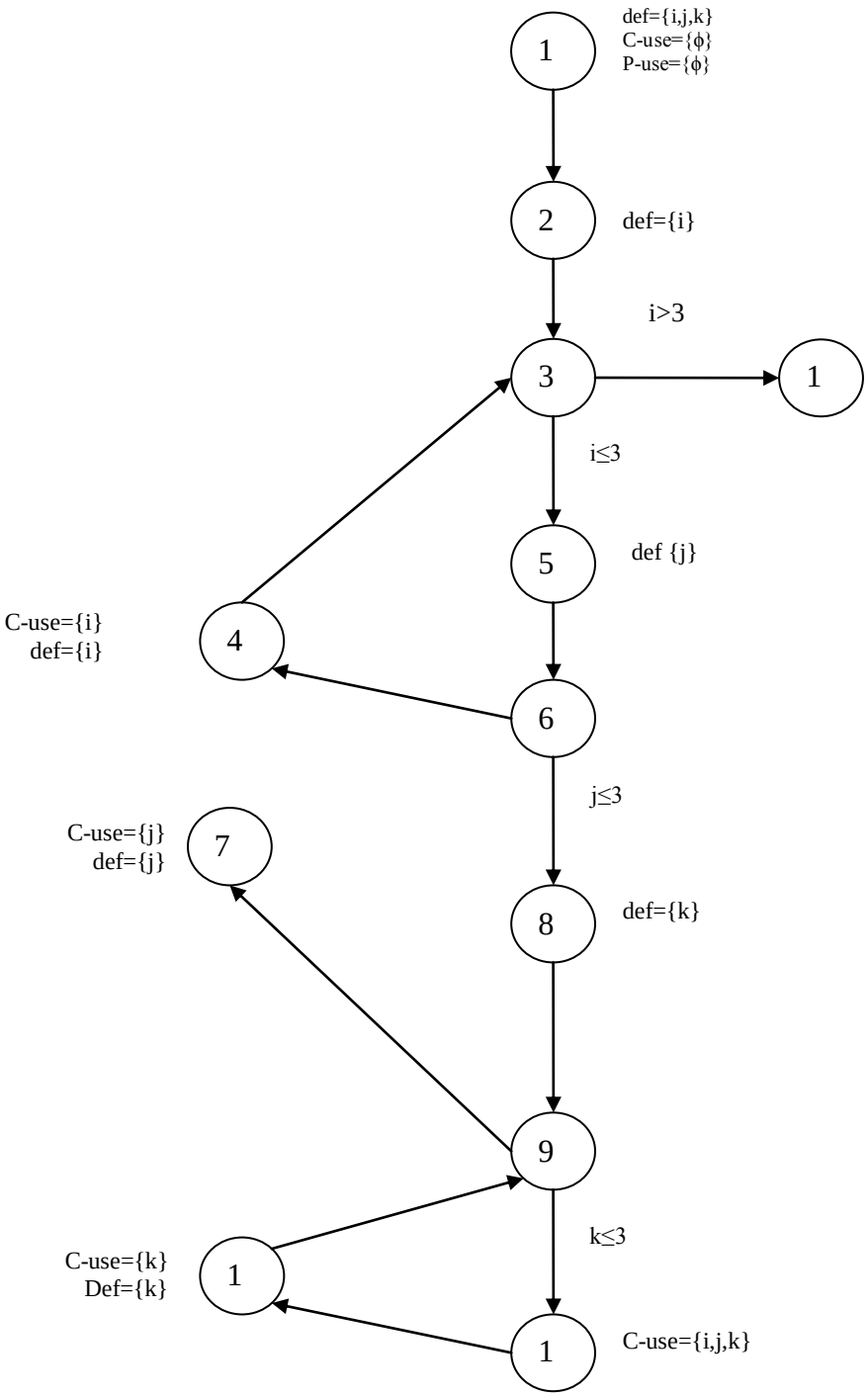

Figure 4.4: Definition-Use Graph

Computation of dcu and dpu

dcu (i,1) ={ }

dcu (j,1) ={ }

dcu (k,1) ={ }

dcu (i,2) = {11,4}

dcu (j,5) = {11,7}

dcu (k,8) = {11,10}

dcu (k,10) = {11,10}

dcu (i,4) = {11,7}

dpu (i,1) = { }

dpu (j,1) = { }

dpu (k,1) = { }

dpu (i,2) = {(3,5)}

dpu (j,5) = {(6,8)}

dpu (k,8) = {(9,11)}

dpu (k,10) = {(9,11),(9,7)}

dpu (i,4) = {(3,5),(3,12)}

dpu (j,7) = {(6,8),(6,4)}

Test path generated on basis of different criterion are as follow:

1. **All Nodes:** In all nodes criterion, test cases are generated to ensure that each node of the def-use graph is to be executed.

   Test case for all nodes: T_1= {1,2,3,5,6,8,9,11,10,9,7,6,4,3,12}

2. **All Edges:** In all edges criterion, test cases are generated to ensure that each edge of the def-use graph is to be executed.

   Test case for all edge: T_1= {1,2,3,5,6,8,9,11,10,9,7,6,4,3,12}

3. **All def:** In all def criterions, test cases are generated to ensure that corresponding to each definition of each variable at least one C-use or P-use will be covered.

   Test case for all def: T_1= {1,2,3,5,6,8,9,11,10,9,7,6,4,3,12}

4. **All P-use:** In all P-use criterions, test cases are generated to ensure that corresponding to each definition of each variable each P-use of the definition will be covered.

   Test case for all P-use: T_1= {1,2,3,5,6,8,9,11,10,9,7,6,4,3,12}

5. **All P-use/Some C-use:** In all P-use/some C-use criterions, test cases are generated to ensure that corresponding to each definition of each variable each P-use of the

definition will be covered. If any definition of the variable does not have p-use then at least one C-use of that will be covered.

Test case for all P-use/some C-use: T_1= {1,2,3,5,6,8,9,11,10,9,7,6,4,3,12}

6. **All C-use/Some P-use:** In all c-use/some P-use criterions, test cases are generated to ensure that corresponding to each definition of each variable each C-use of the definition will be covered. If any definition of the variable does not have C-use then at least one P-use of that will be covered.

   Test case for all C-use/some P-use: T_1= {1,2,3,5,6,8,9,11,10,9,7,6,4,3,12}

7. **All Uses:** In all-uses criterion, corresponding to each definition of each variable every C-use and P-use will be covered.

   Test case for all use: T_1= {1,2,3,5,6,8,9,11,10,9,7,6,4,3,12}

8. **All Path:** Test case for all path criterion:

   {1, 2, 3, 12}

   {1, 2, 3, 5, 6, 4, 3, 12}

   {1,2,3,5,6,8,9,7,6,4,3,12,…}so on.

4.2 Design by Contract in Data Flow Based Testing

We have used design by contract methodology on a stack class to generate test cases using data flow based testing. This technique was known as data flow based testing using contract [16]. The contract specification of class include precondition, post-condition of method and class invariant. Main benefits of this approach are that using this approach less number of du pairs will be generated.

```
class stack
{
  int stk[SIZE];
  int top;
 public:
   stack( );
   void push(int data);
   int pop( );
};
stack:: stack( )
```

```
    {
     pre(true);
     top= -1;
     post(top=-1);
     }
void stack::push ( int data)
{
    pre(top<size-1);
    top=top+1;
    stk[top]=data;
    post(stk[top-1)=data;
  }
 int stack::pop( )
 {
   pre(top!=-1);
   int data= stk[top];
   top=top-1;
   post(!is_full( ));
    return data;
  }
//Stack class invariant is
bool stack_invariant( ){return top>=-1 && top<=size-1;}
```

Figure 4.5: Stack class using design by contract

Class flow diagram constructed from contract specification is shown in figure 4.6

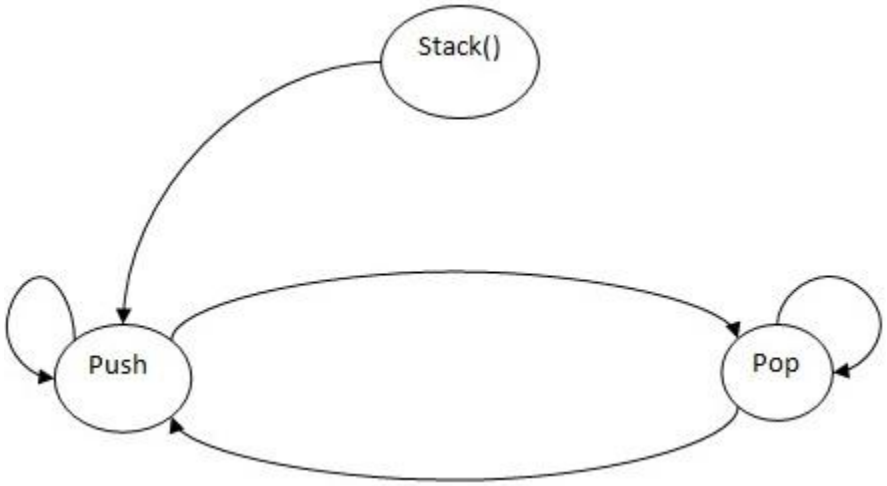

Figure 4.6: Class flow graph for class stack

Table 4.3, represents the various data item their definition and uses.

Table 4.3: Definitions and uses of data members of class stack

Data member	top	stk
Def	Stack, push, pop	push
c-use	push, pop	pop

Table 4.4 represents various feasible du paths and infeasible du paths.

Table 4.4: Feasible and infeasible du pairs of class stack

Feasible du pairs	Infeasible du pairs
(pop, pop) (stack, push) (push, push) (push, pop)	(stack, pop)

Some of the test cases of stack class are as below:

1. Stack, push, push, pop, pop
2. Stack, push, pop, push, pop
3. Stack, push, push, pop
4. Stack, push, pop, push, push, pop

As (stack, pop) is not a feasible path, hence test cases are not generated for the same.

In conventional programming, number of feasible paths are more. Pre-conditions in conventional class become design by contract class in object oriented programming scheme used here.

Code for conventional program is as below:

```
class stack
{
int stk[SIZE];
int top;
public:
 stack()
{
top=0;
}
void push (int a)
{
if (isfull())
{
cout<<"stack is full";
}
stk[top]=a;
top++;
}
int remove()
{
if(isempty())
{
cout<<"stack empty";
}
return (stk[--top]);
}
int stack::isempty()
```

```
{
if (top<0)
return 1;
else  return 0;
}
int stack::isfull()
{
if(top= =SIZE)
return 1;
else  return 0;
}
};
```

Figure 4.7: Conventional stack class

Table 4.5 represents the data definition, c-use and p-use of conventional stack class.

Table 4.5: Definitions and uses of data members of conventional stack class

Data member	**Top**	**Stk**
Def	stack, push, pop	Push
c-use	push, pop	Pop
p-use	push, pop	--

Table 4.6 represents various feasible du paths and infeasible du paths.

Table 4.6: Feasible and infeasible du pairs of conventional stack class

Feasible du pairs	**Infeasible du pairs**
(stack, push) (stack, pop) (push, push) (push, pop) (pop, push) (pop, pop) (stack, push) (stack, pop) (push, push) (push, pop) (pop, push) (pop, pop)	(stack, pop)

Clearly conventional programming is having more du feasible paths as compared to design by contract.

4.3 Automation of Data Flow Based Testing

A software tool is designed in ANSI C language that will help in data flow based testing. It will accept any c program file (like fact.c). For any variable used in the input c program, it will check whether there are any data anomalies is presented or not. It will also display variable definition and both computation and predicate use and at last killing of variable which will be useful in making test cases based on any criterion of data flow based testing. A various figures of working of the tool in ANSI C are given below

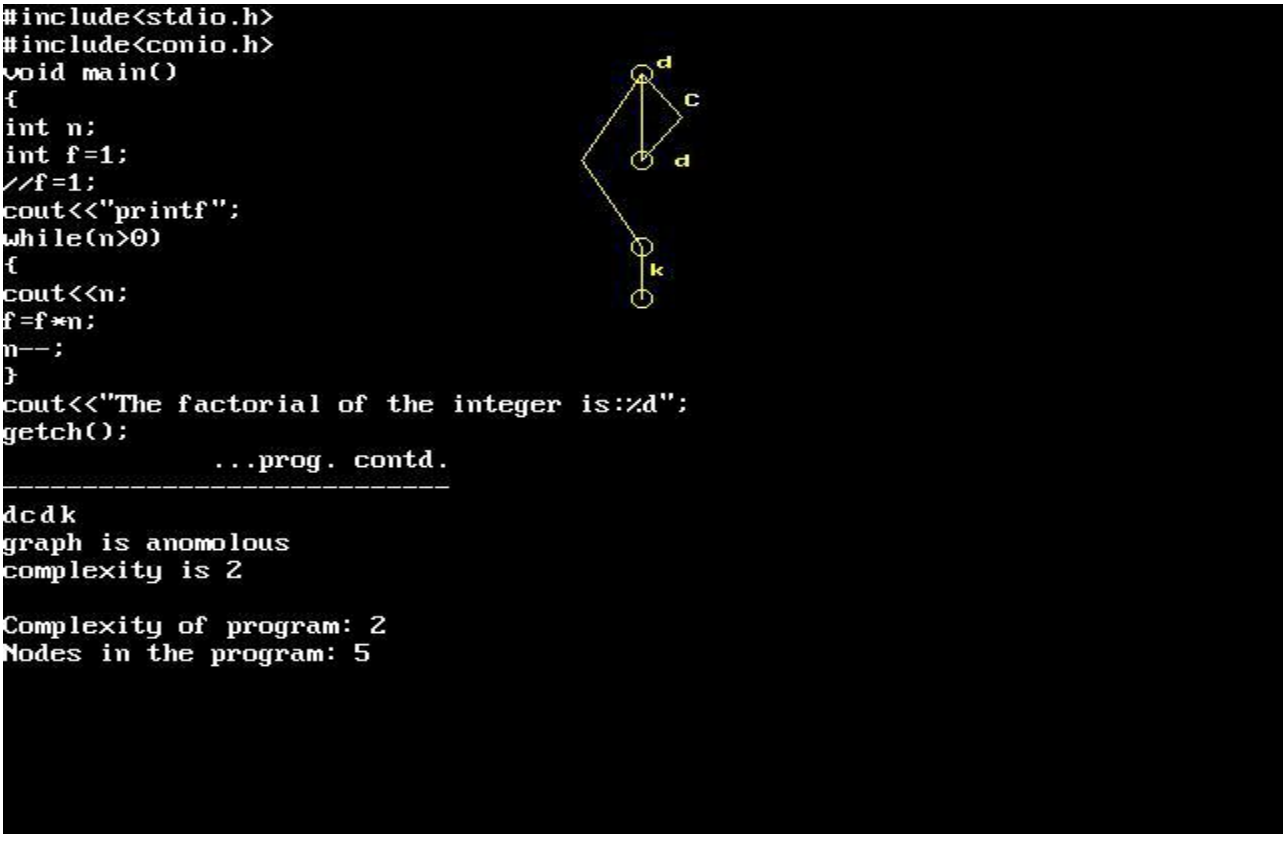

Figure 4.8: Snap-shot representing automaton of data-flow based testing

Figure 4.8, represents various data uses of variable instance f and the cyclomatic complexity of the program. dk represents data-killed anomalies as shown in figure.

Chapter 5

Conclusion and Future Scope

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. Using data flow testing design by contract technique is applied on stack class. Clearly design by contact data flow based testing cause reduction in number of du paths. Hence the cost of software testing can be reduced. Software is designed for checking data anomalies in the tested modules. It will display a def-use graph for the tested modules. For each variable, we can check when a variable is defined, used and killed. This can help in automaton of data flow based testing.

Following are the future direction on which work can be carried out:

1. This software can be extended so that corresponding to each variable definition dcu and dpu can be displayed and based on any strategies software will helps in designing of test cases.
2. Design by contract with data flow based testing can be used for inter-class testing.

References

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

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

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

[4] D. Goldberg: "*Genetic Algorithms*", Addison Wesley, 1988.

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

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

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

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

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

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

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

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

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

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

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

[16] Yogesh Singh and Anju Saha: "Enhancing Data Flow Testing of Classes through design by Contract", Seventh IEEE/ACIS International Conference on Computer and Information Science, 2008.

Publications

COMMUNICATED

Ravideep Lochav, Sunita Garhwal, "Review paper on data-flow based testing", Communicated to International journal of Advanced Research in Computer science.