An Introduction to Data-Flow Testing

Janvi Badlaney Rohit Ghatol Romit Jadhwani
Department of Computer Science
North Carolina State University
Raleigh, NC 27695, USA
jrbadlan@unity.ncsu.edu, rgghatol@unity.ncsu.edu, rgjadhwa@unity.ncsu.edu

Abstract

Control flow diagrams are a keystone in testing the structure of software programs. By examining the flow of control between the various components, we can design and select test cases. Data-flow testing is a control-flow testing technique which also examines the lifecycle of data variables. Use of data-flow testing leads to a richer test suite concentrating on improper use of data due to coding errors. The main goal of this paper is to discuss the concept of data-flow testing and apply it to a running example.

Keywords: *Data-flow testing, control-flow graph, Data-flow anomaly.*

1. Introduction

Software testing is "*The process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items*" [9]. The main goals of software testing are to reveal bugs and to ensure that the system being developed complies with the customer's requirements. To make testing effective, it is recommended that test planning/development begin at the onset of the project. Software testing techniques can be divided into 2 kinds: black box and white box techniques. Black box testing is mainly a validation technique that checks to see if the product meets the customer requirements. However, white box testing is a verification technique which uses the source code to guide the selection of test data.

Data-flow testing is a white box testing technique that can be used to detect improper use of data values due to coding errors [6]. Errors are inadvertently introduced in a program by programmers. For instance, a software programmer might use a variable without defining it. Additionally, he/she may define a variable, but not initialize it and then use that variable in a predicate [6].

e.g.
```
int x ;
if (x ==100) {};
```

In data-flow testing, the first step is to model the program as a control flow graph. This helps to identify the control flow information in the program. In step 2, the associations between the definitions and uses of the variables that is needed to be covered in a given coverage criterion is established. In step 3, the test suite is created using a finite number of paths from step 2.

In this paper, we have discussed the concept of data-flow testing. The next section covers the data-flow testing criteria and data-flow anomalies. A billing application is considered and the corresponding control-flow graphs are presented and annotated to explain the concept of data-flow testing. Section 3 presents the test cases created for this application. Section 4 summarizes the concepts presented in this paper and concludes the paper.

2. Literature survey/Case Study

This section discusses data-flow testing concepts, data-flow anomalies and data-flow testing strategies. Throughout this section, data-flow testing techniques are illustrated using an example of a billing application.

Data-flow testing monitors the lifecycle of a piece of data and looks out for inappropriate usage of data during definition, use in predicates, computations and termination (killing). It identifies potential bugs by examining the patterns in which that piece of data is used. For example, A pattern which indicates usage of data in a calculation after it has been killed is certainly a bug which needs to be addressed.

To examine the patterns, we need to construct a control flow graph of the code. A control flow graph is a directed graph where the nodes represent the processing statements like definition, computation and predicates while the edges represent the flow of control between processing statements. Since data-flow testing closely examines the state of the data in the control flow graph, it results in a richer test suite than the one obtained from traditional control flow graph testing strategies like all branch coverage, all statement coverage, etc [3].

2.1. Data-flow Anomalies

Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.

The notation for representing the patterns is [2][5]:

- 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 1 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 1: Testing anomalies [1][6]

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.

2.2. Static Data-flow Testing

With static analysis, the source code is analyzed without executing it [6]. Let us consider an example of an application to calculate the bill of a cellular service customer depending upon on his/her usage. The following calculates 'Bill' as per 'Usage' with the following rules applicable. If 'Bill' is more than $100, 10% discount is given.

Table 2: Billing rules

Usage(min)	Bill ($)
<100	40.0
101-200	50 cents for every additional minute.
>200	10 cents for every additional minute.

The source code for the above application is:

```
public static double calculateBill (int Usage)
{
 double Bill = 0;

 if(Usage > 0)
 {
       Bill = 40;
 }

 if(Usage > 100)
 {
       if(Usage <= 200)
       {
       Bill = Bill + (Usage - 100) * 0.5;
       }
       else
       {
       Bill = Bill + 50 + (Usage - 200) * 0.1;

              if(Bill >= 100)
              {
                     Bill = Bill * 0.9;
              }
       }

 }
 return Bill;
}
```

The control flow diagram is given in Figure 1 and the annotated control flow diagram with define-use-kill information for each variable is given in Figure 2.

For variable '*Usage*', the define-use-kill patterns are

- ~ define : normal case
- define-use : normal case
- use-use : normal case
- use-kill : normal case

For variable '*Bill*', the define-use-kill patterns are

- ~ define : normal case
- define-define: suspicious
- define-use : normal case
- use-define : acceptable
- use-use : normal case
- use-kill : normal case

The static data-flow testing for the given application discovered the following anomaly:

Bill: define – define

Why Static Data-flow testing is not enough?

Static Data-flow testing will fail in situations where the state of a data variable cannot be determined by just analyzing the code. This is possible when the data variable is used as an index for a collection of data elements. For example, in case of arrays, the index might be generated dynamically during execution hence we can't guarantee what the state of the array element is which is referenced by that index. Moreover, the static data-flow testing might denote a certain piece of code to be anomalous which is never executed and hence not completely anomalous [7].

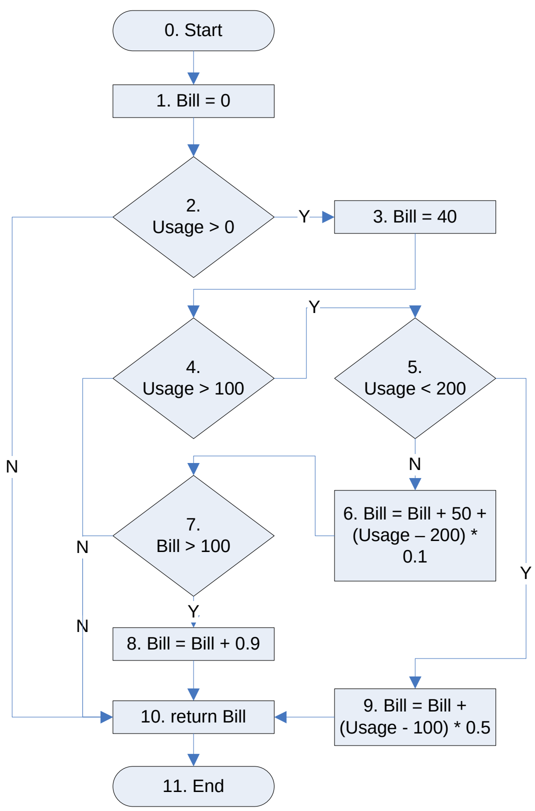

Figure 1. Control flow Diagram

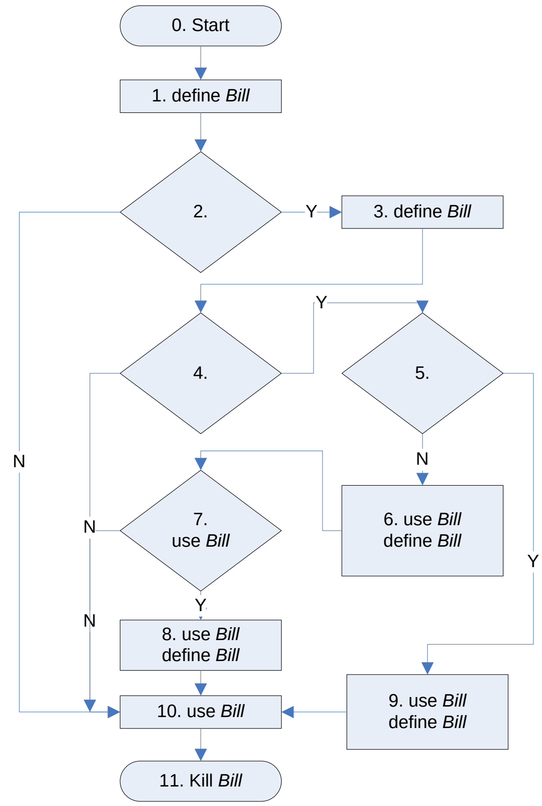

Figure 2. Annotated control flow diagram for *'Bill'*

Table 3: Static analysis for variable *'Bill'*

Anomaly		Explanation
~d	0-1	Allowed. Normal case
dd	0-1-2-3	Potential bug. Double definition.
du	3-4-5-6	Allowed. Normal case.
ud	6	Allowed. Data is used and then redefined.
uk	10-11	Allowed.
dd	1-2-3	Potential bug. Double definition.
uu	7-8	Allowed. Normal case.
k~	11	Allowed. Normal case.

Referring to Table 3, we observe that static data-flow testing for variable *'Bill'* discovered the following usage 0-1-2-3 as a potential bug.

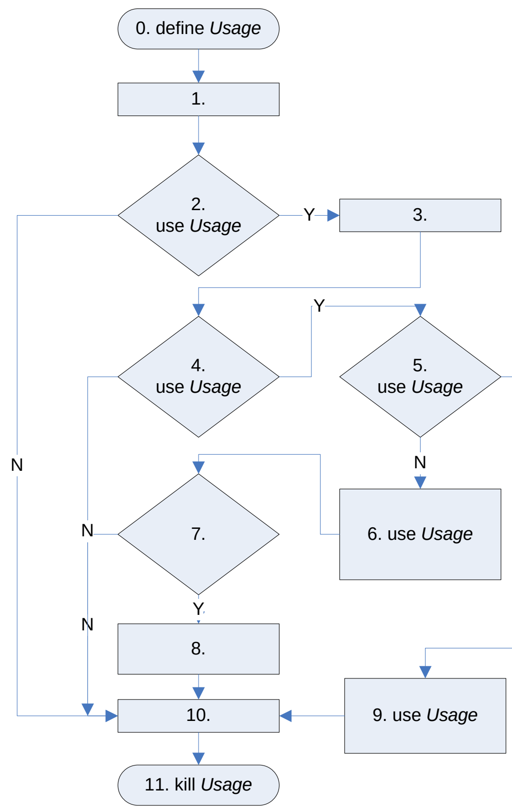

Figure 3. Annotated control flow diagram for *'Usage'*

Table 4: Static analysis for variable *'Usage'*

Anomaly		Explanation
~d	0	Allowed.
du	0-1-2	Allowed. Normal case.
uk	9-10-11	Allowed.
uu	5-6	Allowed. Normal case.
k~	11	Allowed. Normal case.

Referring to Table 4, we observe that static data-flow testing for variable *'Usage'* did not discover any bugs.

2.3. Dynamic Data-flow Testing

The primary purpose of dynamic data-flow testing is to uncover possible bugs in data usage during the execution of the code. To achieve this, test cases are created which trace every definition to each of its use and every use is traced to each of its definition. Various strategies are employed for the creation of the test cases [4][5]. The definition of all strategies is followed by an example to explain the same.

All-du paths (ADUP)
Formal Definition
'Every du path from every definition of every variable to every use of that definition' [7].
It is the strongest data-flow testing strategy since it is a superset of all other data flow testing strategies. Moreover, this strategy requires greatest number of paths for testing.

All-Uses (AU)
Formal Definition
'At least one path from every definition of every variable to every use of that can be reached by that definition' [7].

For every use of the variable, there is a path from the definition of that variable to the use.

All-p-uses (APU)
Formal Definition
'APU Strategy is derived from APU+C by dropping the requirement of including a c-use if there are no p-use instances following the definition' [7].

In this testing strategy, for every variable, there is path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then it is dropped from contention.

All-c-uses (ACU)
Formal Definition
'ACU Strategy is derived from ACU+P by dropping the requirement of including a p-use if there are no c-use instances following the definition' [7].

In this testing strategy, for every variable, there is a path from every definition to every c-use of that definition. If there is a definition with no c-use following it, then it is dropped from contention.

All-p-uses/Some-c-uses (APU+C)
Formal Definition
'For every variable and every definition of that variable, include at least one path from the definition to every predicate use; if there are definitions of the variable that are not covered then add computational use test cases as required to cover every definition' [7].

In this testing strategy, for every variable, there is a path from every definition to every p-use of that definition. If there is a definition with no p-use following it, then a c-use of the definition is considered.

All-c-uses/Some-p-uses (ACU+P)

<u>Formal Definition</u>

'For every variable and every definition of that variable, include at least one path from the definition to every computational use; if there are definitions of the variable that are not covered then add predicate use test cases as required to cover every definition' [7].

In this testing strategy, for every variable, there is a path from every definition to every c-use of that definition. Ibf there is a definition with no c-use following it, then a p-use of the definition is considered.

All-definition (AD)

<u>Formal Definition</u>

'Every definition of every variable be covered by at least one use of that variable, be that use a computational use or a predicate use'.

In this strategy, there is path from every definition to at least one use of that definition.

Let us consider our billing application and perform dynamic data-flow testing. The control flow graph is annotated for each variable (Figure 4 and Figure 5) by removing references to all other variables and replacing the contents of the nodes with 'def' for definition, 'c-use' for computational use and 'p-use' for predicate use. The testing strategies are then applied to the annotated control flow graphs and test cases are derived. Table 5 presents the list of test paths for variables 'Bill' and 'Usage' of the billing application.

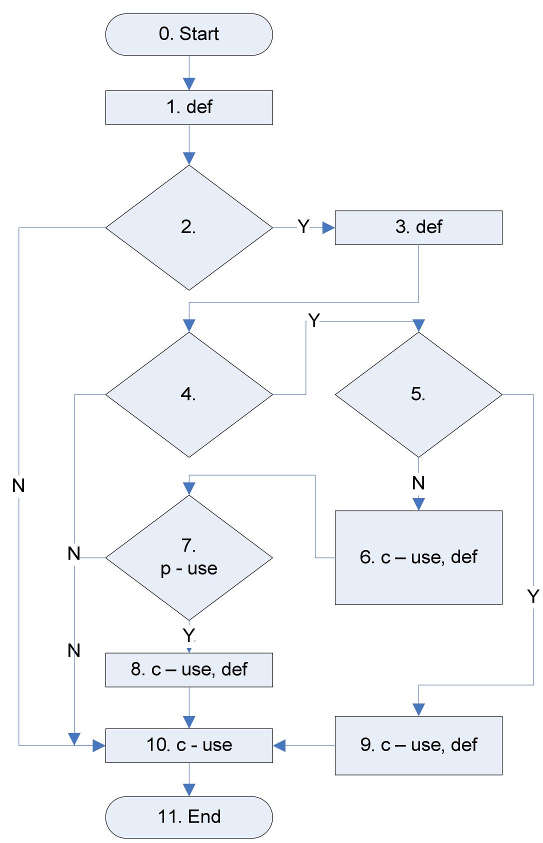

Figure 4.Annotated Control Flow diagram for variable '*Bill*'

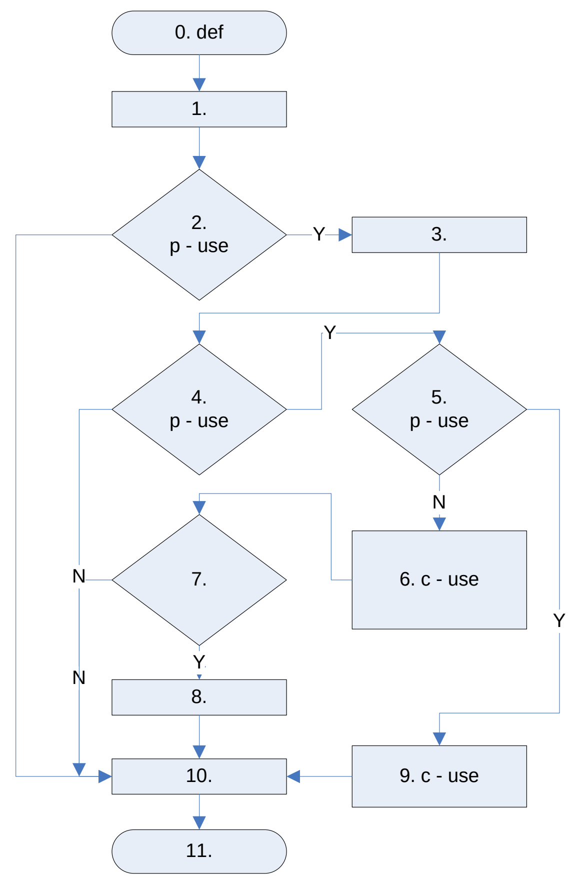

Figure 5.Annotated Control Flow diagram for variable '*Usage*'

Table 5: Data-flow testing paths for each variable

Strategy	Bill	Usage
All uses (AU)	3-4-5-6 6-7 6-7-8 8-10 3-4-5-9	0-1-2 0-1-2-3-4 0-1-2-3-4-5 0-1-2-3-4-5-6 0-1-2-3-4-5-9
All p – uses (APU)	1-2-3-4-5-6-7 3-4-5-6-7 6-7	0-1-2 0-1-2-3-4 0-1-2-3-4-5
All c – uses (ACU)	1-2-10 3-4-5-6 3-4-5-9 3-4-10 6-7-8 6-7-10 8-10 9-10	0-1-2-3-4-5-6 0-1-2-3-4-5-9
All p – use/ some c (APU + C)	1-2-3-4-5-6-7 3-4-5-6-7 6-7 8-10 9-10	0-1-2 0-1-2-3-4 0-1-2-3-4-5
All c – use/ some p (ACU + P)	1-2-10 3-4-5-6 3-4-5-9 6-7-8 8-10 9-10	0-1-2-3-4-5-6 0-1-2-3-4-5-9
All du (ADUP)	(ACU+P) + (APU+C)	(ACU+P) + (APU+C)
All definition (AD)	1-2-10 3-4-5-6 6-7 8-10 9-10	0-1-2

In Table 5 for variable *'Bill'*:
Path 1-2-10 depicts an all-definition path from definition in 1 to c-use in 10.
For All c-uses, we trace a path for every definition of Bill to at least one c-use and a path tracing from every c-use to its definition.
The same applies for All p-uses.

In case of All c-uses/Some p-uses, we observe that the paths are similar to All c-uses since every definition has a corresponding c-use.
In case of All p-uses/Some c-uses, definitions in 8 and 9 don't have a corresponding p-use, hence c-use in 10 is considered.
For All uses, we trace a path to all the uses (c-use and p-use).

2.4. Ordering of Strategies

For selection of test cases, we need to analyze the relative strength of the data-flow testing strategies. Figure 6 depicts the relative strength of the data-flow strategies and other control-flow testing strategies such as all-branch and all-statement. According to the figure, the strength of testing strategies reduces along the direction of the arrow. Hence ALL PATHS is the strongest testing strategy. Also note that ACU+P and APU+C run parallel hence they are comparable [7].

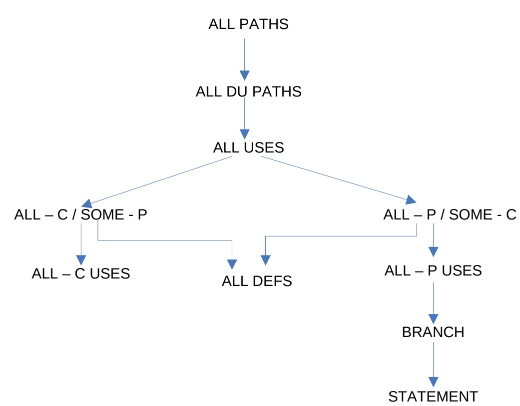

Figure 6. Relative Strength of Testing Strategies [7]

3. Test Case Creation

After obtaining the test paths, test cases are created by giving values to the input parameter (*'Usage'* for application considered). We obtain different test suites for each variable. For the application considered, we get 2 test suites for *'Bill'* and *'Usage'* respectively.

Table 6. Test Suite for variable 'Bill'

Uses	Bill	Input Usage Value	Expected Value
All definition (AD)	1-2-10 3-4-5-6 6-7 8-10 9-10	0 220 220 350 220	0.0 92.0 92.0 94.5 92.0
All c – use (ACU)	1-2-10 3-4-5-6 6-7-8 8-10 9-10	0 220 350 350 220	0.0 92.0 94.5 94.5 92.0
All p – use (APU)	1-2-3-4-5-6-7 3-4-5-6-7 6-7	220 220 220	92.0 92.0 92.0
All c - use + p (ACU + P)	1-2-10 3-4-5-6 3-4-5-9 6-7-8 8-10 9-10	0 220 170 350 350 170	0.0 92.0 75.0 94.5 94.5 75.0
All p - use + c (APU + C)	1-2-3-4-5-6-7 3-4-5-6-7 6-7 8-10 9-10	220 220 220 350 170	92.0 92.0 92.0 94.5 75.0
All uses (AU)	3-4-5-6 6-7 6-7-8 8-10 3-4-5-9	220 220 350 350 170	92.0 92.0 94.5 94.5 75.0

Table 7. Test Suite for variable 'Usage'

Uses	Usage	Input Usage Value	Expected Value
All definition (AD)	0-1-2	0	0.0
All c – use (ACU)	0-1-2-3-4-5-6 0-1-2-3-4-5-9	220 170	92.0 75.0
All p – use (APU)	0-1-2 0-1-2-3-4 0-1-2-3-4-5	0 170 170	0.0 75.0 75.0
All c - use + p (ACU + P)	0-1-2-3-4-5-6 0-1-2-3-4-5-9	220 170	92.0 75.0
All p - use + c (APU + C)	0-1-2 0-1-2-3-4 0-1-2-3-4-5	0 170 170	0.0 75.0 75.0
All uses (AU)	0-1-2 0-1-2-3-4 0-1-2-3-4-5 0-1-2-3-4-5-6 0-1-2-3-4-5-9	0 170 170 220 170	0.0 75.0 75.0 92.0 75.0

4. Conclusion

We have presented a literature survey of data-flow testing concentrating on data-flow testing criteria and data-flow testing anomalies. We have presented the concept with the help of a billing application that bills mobile customers as per their usage. Finally, we provided the test cases created after performing data-flow testing on the considered application.

5. References

[1] Tsai, B.-Y.; Stobart, S.; Parrington, N.; "Employing data flow testing on object-oriented classes", Software, IEEE Proceedings, USA, April 2001, p 56-64-87.

[2] 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, December 1994, pp 154 - 163.

[3] Chang Liu; "Teaching "Data Flow Testing" in an Software Engineering Course", School of Electrical Engineering and Computer Science, Russ College of Engineering and Technology, Ohio University.

[4] Rapps, Sandra, Elaine J. Weyuker; "Data Flow Analysis Techniques for Test Data Selection", Sixth International Conference on Software Engineering, Tokyo, Japan, September 13-16, 1982.

[5] Parrish, A.S.; Zweben, S.H.; "On the relationships among the all-uses, all-DU-paths, and all-edges testing criteria", Software Engineering, IEEE Transactions, 1995, p 1006-1009.

[6] Lee Copeland, "A Practitioner's Guide to Software Test Design", STQE Publishing, 2004.

[7] Boris Beizer, "Software Testing Techniques", International Thomson Computer Press, 1990.

[8] Robert V. Binder, "Testing Object-Oriented Systems Models, Patterns and Tools", Addison-Wesley, 2000.

[9] IEEE Standard 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminolog*y, Technical Report, IEEE, 1990

6. Appendices

Appendix A: Glossary

Black-box testing – The use of specifications of a unit, subsystem, or system to design tests. It is synonymous with specification-based testing or functional testing. [8]

Branch coverage – Branch coverage is achieved when every path from a control flow graph node has been executed at least once by a test suite. [8]

Data-Flow Anomaly – A data-flow anomaly is denoted by a two-character sequence of actions [7].

Data-Flow Testing – It selects paths through the program's control flow in order to explore sequences of events related to the status of data objects [7].

State - The state of an object can be defined as a set of instance variable value combinations that share some property of interest. [8]

Statement coverage – Coverage achieved when all the statements in a method have been executed at least once. [8]

Test case – A set of inputs, execution conditions, and expected results developed for a particular objective [8].

Testing – The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component [8].

Test suite – A related collection of test cases [8].

White-box testing – The use of source-code analysis to develop test cases. It is synonymous with program-based testing or structural testing [8].