Comparative Analysis of Flow Graph and Dependence Graphs

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

Master of Engineering
in
Software Engineering

Submitted By:
Shveta Verma
(Roll No. 801031030)

Under the supervision of:
Mr. Vinay Arora
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Comparative Analysis of Flow Graph and Dependence Graphs,*" in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Vinay Arora* 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.

Shveta Verma.
(Shveta Verma)

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

Vinay Arora.
(Mr. Vinay Arora)
Assistant Professor
CSED, Thapar University
Patiala

Countersigned by

03/07/2011.
(Dr. Maninder Singh)
Head of Department
CSED, Thapar University
Patiala

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

Acknowledgement

First of all, I am thankful to God for all the blessings and showing me the right direction. Due to the mercy of God, it has been made possible for me to reach so far.

I wish to express my deep gratitude to Mr. VinayArora, Assistant Professor, Computer Science and Engineering Department, Thapar University, Patiala for providing his support throughout the span of my thesis. This work would not have been possible without his encouragement and valuable guidance.

I am also thankful to Dr. Maninder Singh, Head, Computer Science and Engineering Department for his kind help and cooperation. I express my gratitude to all the staff members of Computer Science and Engineering Department for providing me all the facilities required for the completion of my thesis work.

I would like to say thanks to all my friends especially Mehak Jindal for her support. I want to express my appreciation to every person who contributed with either inspirational or actual work to this thesis.

Last but not the least I am highly grateful to all my family members for their inspiration and ever encouraging moral support, which enables me to purse my studies.

ShvetaVerma

Abstract

Software Testing is the process of executing a software system to determine whether it matches its specifications when executed in its intended environment. It is an important process of accessing the software to determine its quality. Test case generation is a method to generate test cases to make comparative analysis of actual outcome to predicted outcome. The graphical methods provide the structural representation of the system that facilitates testing the logical development of the program. There are many graphs such as Control Flow Graph (CFG), Program Dependence Graph (PDG), Class Dependence Graphs(ClDG), System Dependence Graph(SDG), *etc*that can represent various features like single/multiple procedures, flow/context sensitivity, inheritance, polymorphism and so on.

Each of these graphs represents some unique features/parameters like CFG represents flow of control between statements of a program, SDG represents method calls between multiple procedures and so on. The comparative analysis of these graphical techniques can be done on the basis of these features.

Table of Content

Certificate…… i

Acknowledgement……..... ii

Abstract…….. iii

Table of Content……. iv

List of Figures……. vi

List of Tables……..... viii

Chapter 1. Introduction…….. 01

1.1 Software Testing Terminologies…….. 01

1.1.1 Test Case Generation…..…… 01

1.1.2 Test Execution Automation…… 03

1.1.3 Testing Activities…… 03

1.2 Object-Oriented Concepts……..…….. 04

1.2.1 Testing Object-Oriented System……. 04

1.3 Graphical Methods……..... 07

1.3.1 Control Flow Graph (CFG)…….. 07

1.3.2 Control Dependence Graph……. 10

1.3.3 Data Dependence Graph…….. 12

1.3.4 Program Dependence Graph…… 13

1.3.5 System Dependence Graph……. 15

1.4 Thesis Layout…….…….. 18

Chapter 2. Literature Survey……. 19

2.1 Graphical Methods……..... 19

2.2 Test Case Generation……..... 19

2.2.1 Using Def-Use Graph……. 20

2.2.2 Using Control Flow Graph……. 21

2.2.3 Using Program Dependence Graph……..... 24

2.2.4 Using System Dependence Graph…….. 25

2.2.5 Using ddgraph……..... 26

2.2.6 Using TestGraph…… 27

2.2.7 Using Class Graph……. 27

2.2.8 Using Application Call Graph……... 29

2.2.9 Using COSDG……... 29

2.3 Tools…….…….. 31

2.4 Comparison of Graphical Methods…… 31

Chapter 3. Problem Statement……..… 33
Chapter 4. Methodology……..……..… 34
Chapter 5. Experimental Results…… 36
5.1 Tool for generating Control Flow Graph…….. 36
5.2 System Dependence Graph API……..……..... 37
5.3 Implementation…….…… 37
5.3.1 Using CFG Generator……. 37
5.3.2 Using SDG API…….. 41
Chapter 6. Conclusion and Future Scope…….. 47
6.1 Conclusion…… 47
6.2 Future Work……..... 50
References……. 51
List of Publications.……... 57

List of Figures

Fig 1.1: Path for Testing Activities …….. 03
Fig 1.2: Levels of Testing Object-Oriented Software ……. 05
Fig 1.3: Integration Testing Strategies ……..... 06
Fig 1.4: System Testing Strategies …..…… 06
Fig 1.5: Dependence Graph [61]……... 07
Fig 1.6: Block/Node representing set of statements……. 08
Fig 1.7: CFG for Sequence, Selection and Iteration type of constructs [2]…….. 08
Fig 1.8(a): An Example Program…….. 09
Fig 1.8(b): Control Flow Graph of example program……... 09
Fig 1.9(a): An Example Program [8]…… 10
Fig 1.9(b): Basic Blocks of the Program [8]……. 10
Fig 1.10(a): Block representing set of statements [8]……... 11
Fig 1.10(b): Nodes representing set of statements [8]…….. 11
Fig 1.11: Control Dependence Graph.……... 11
Fig 1.12(a): An Example Program[63]……. 12
Fig 1.12(b): its Control Dependence Graph [63]…….. 12
Fig 1.13: Data Dependence Graph…… 12
Fig 1.14: Data Dependence Graph of example program in Fig 1.12(a) [63]…… 13
Fig 1.15: A Sample Java Program to calculate factorial of a number [62]……... 14
Fig 1.16: The Program Dependence Graph [62]……... 14
Fig 1.17: An Example Program [70]……. 15
Fig 1.18: The System Dependence Graph [70]……. 16
Fig 1.19: An Example Program [62]……. 16
Fig 1.20: The System Dependence Graph [62]……. 17
Fig 2.1: A family of path selection criteria [16]…… 19
Fig 2.2(a): A sample program in Ada language [25]…… 22
Fig 2.2(b): its EIAG [25]…….. 22
Fig 2.3(a): Code for Bag class declaration [36]…… 27
Fig 2.3(b): Testgraph for Bag [36]…… 27
Fig 2.4(a): An algebraic specification for the class of integer stack [37]……. 28
Fig 2.4(b): its Class Graph [37]……. 28

Fig 2.5: Graphical Symbols used in COSDG [41]…… 29
Fig 2.6(a): An Example Java Program [41]…….. 30
Fig 2.6(b): Various types of method calls in COSDG [41]……... 30
Fig 5.1: Framework/Schema for CFG Generator plug-in [72]…….. 36
Fig 5.2: UML Class Diagram for CFG Generator [72]……. 37
Fig 5.3: Code for Demo Class in Java……... 38
Fig 5.4: Demo Class program in Eclipse Environment……. 38
Fig 5.5: Steps to use CFG Generator plug-in…… 39
Fig 5.6: Control Flow Graph for add( ) method…… 40
Fig 5.7: Control Flow Graph for main( ) method……. 40
Fig 5.8: Demo Class Program in NetBeans Environment……. 41
Fig 5.9: Control Dependence Graph for the Demo Class……. 42
Fig 5.10: Control Dependence Graph for add( ) method…….. 42
Fig 5.11: Control Dependence Graph for main( ) method…… 43
Fig 5.12: Data Dependence Graph for the Demo Class…… 43
Fig 5.13: Data Dependence Graph for add( ) method……... 44
Fig 5.14: Data Dependence Graph for main( ) method…… 44
Fig 5.15: Program Dependence Graph for the Demo Class…….. 45
Fig 5.16: System Dependence Graph for the Demo Class…… 45
Fig 6.1: CFG, PDG, SDG and their sub-graphs…… 48

List of Tables

Table 2.1 Tools Comparison…….. 31
Table 2.2 Graphical methods comparison…….. 32
Table 6.1 Comparison of CFG, PDG and SDG……. 47
Table 6.2 Description of Graphs shown in Fig 6.1…… 49

Chapter 1
Introduction

Software testing is the process of executing a software system to determine whether it matches its requirement specification when executed in its intended environment. It is a procedure which encompasses a wide spectrum of different activities like testing of a small piece of code by the developer, to the validation of a large information system by the customer. The testing activity should be carried out according to a formal procedure in which rigorous planning and documentation is alsorequired. Testing is one of the oldest forms of verification. The ultimate goal of software testing is to construct systems with high quality.

1.1 Software Testing Terminologies

1.1.1 Test Case Generation: A test case is a well-documented procedure designed to test the functionality of a system feature. The primary purpose of designing a test case is to find errors in the program behavior. Test Case Generation is a method to generate test cases which provides description of a test, independent of the way a given system is designed. Every form of test generation uses a source document and in the most informal of test methods, the source document resides in the mind of the tester who generates tests based on the knowledge of the requirements. In more advanced test processes, requirements serve as a source for the development of formal models. In the various stages, the test cases could be devised aiming at different objectives, such as exposing deviations from user's requirements or assessing the conformance to a standard specification [1].

To demonstrate the test case generation procedure, consider an example in which three sides of a triangle are given as integers x, y, and z and it is desired to have a program to determine the type of the triangle that whether it is an equilateral or an isosceles or a scalene. The behavior (output) of the program depends on three integer values given as sides. The constraints set for above scenario can be divided into the four categories.

C1: The values of x, y and z are integers.

C2: Each input contains exactly three values namely x, y and z.

C3: The values of x, yand z are greater than zero.

C4: The length of the longest side is less than the sum of the lengths of the other two sides.

Test Set for C1 and C2: In first category, a decimal value or a string value will be entered when an integer value is needed, it will normally give rise to an exception resulting in the termination of the program. In category two, wrong number of inputs will be provided.

Test Set for C3: Although this third category refers to all three input variables with independent situations. To guarantee that each invalid situation is checked independently, invalid test sets should be recognized for each of the variables having a non-positive value:

1. $\{(x, y, z) \mid x \leq 0, y, z > 0\}$
2. $\{(x, y, z) \mid y \leq 0, x, z > 0\}$
3. $\{(x, y, z) \mid z \leq 0, x, y > 0\}$

Test Set for C4: For the fourth category, the relative sizes of the values are important. However, each one of the three variables can be the one that has the largest value (corresponds to the longest side). Thus, three more invalid test sets can be generated as:

4. $\{(x, y, z) \mid x \geq y, x \geq z, x \geq y + z\}$
5. $\{(x, y, z) \mid y \geq x, y \geq z, y \geq x + z\}$
6. $\{(x, y, z) \mid z \geq x, z \geq y, z \geq x + y\}$

In above mentioned example, the program produces different output for each of the three different types of triangles. A valid triangle is called equilateral if all three sides are equal, isosceles if exactly one pair of sides is equal, and scalene if none of the sides are equal. This yields five valid test sets for valid data one for equilateral, three for isosceles, depending on which pairs of sides are equal and one for scalene:

7. $\{(x, y, z) \mid x = y = z\}$
8. $\{(x, y, z) \mid x = y, z \neq x\}$

9. {(x, y, z) | y = z, x ≠y}

10. {(x, y, z) | x = z, y ≠x}

11. {(x, y, z) | x ≠y, y ≠z, x ≠z}

Thus, we have 11 test sets that represent 11 different scenarios where integer values have been provided to all three sides of a triangle. For each of these test sets, we need to generate at least one test case to ensure that the program can handle the data values from the test set correctly.

1.1.2 Test Execution Automation: Test Automation is the use of the software to control the execution of tests where the comparison of actual outcomes to predicted outcomes has been performed**.** It automates the setup and running of tests that have been either automatically or manually generated. Test execution automation is a mandatory prerequisite to test design automation. It is also a fundamental precondition to any kind of regression testing.

1.1.3 Testing Activities: Testing incorporates four activities as shown in Fig 1.1[2].

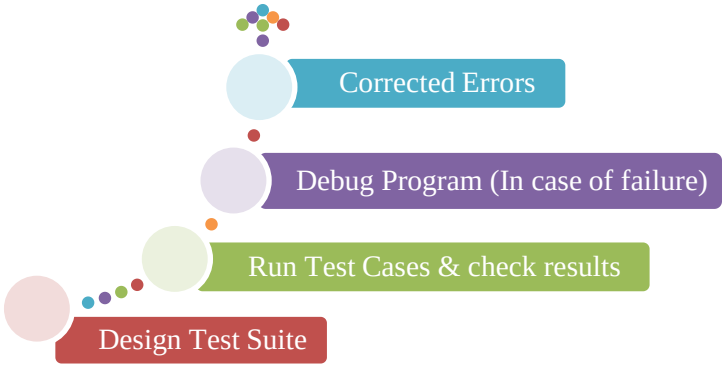


Fig 1.1 Path for Testing Activities

1) Test Suite Design: The set of test cases using which a program is to be tested.
2) Running test cases and checking results: Each test case is executed and the results are compared with the expected results. A mismatch between the actual result and expected result indicates a failure.

3) Debugging: For each observed deviation in the previous activity, debugging is carried out to identify the statements that results an error.
4) Error Correction: After locating the error, the code is appropriately amended to rectify the incorrectness.

1.2 Object-Oriented Concepts

The advent of Object-Oriented programming has brought numerous blessings to all the aspects of software design and development, but it has not reduced the need for testing. Reusable components should be tested more than software fragments and used only within a single system. Object-Oriented programming has many useful features such as information hiding, encapsulation, inheritance, polymorphism and dynamic binding. These object-oriented features facilitate software reuse and component-based development.

1.2.1 Testing Object–Oriented System

One important issue in Object-Oriented programming is how to test Object-Oriented software. Postponing testing towards the end of Software Development Life Cycle (SDLC) should be avoided. It should start at reasonable points in analysis and design phases. This will help to uncover problems early in the development process.

As shown in Fig 1.2, the three levels of testing Object-Oriented software were suggested namely Unit testing (or method level), Integration testing and System testing [3, 4].

- Unit testing: This focuses on the testing of individual method in a class. It ensures statement coverage, decision coverage and path coverage. Statement coverage implies that all statements have been traversed atleast once. Decision coverage implies that all conditional executions have been traversed and path coverage ensures the execution of true and false parts of the loop. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed.

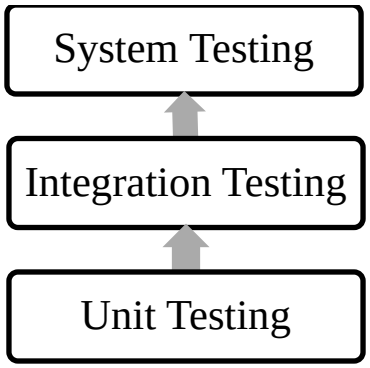


Fig 1.2 Levels of Testing Object-Oriented Software

- Integration Testing: This testing is carried out after all the modules have been unit tested. The objective of the integration testing is to detect the errors at the module interfaces. For example, it has to be checked that no parameter mismatch will occur when one module invokes the functionality of another module. During integration testing, different modules of a system are integrated in a planned manner using an integration plan. After each integration step, the partially integrated system is tested [4].

  As shown in Fig 1.3, there are three different strategies for integration testing of Object-Oriented systems [5]:

  - ✓ Thread-based testing: It integrates the set of classes required to respond to one input or event for the system. Each thread is integrated and tested individually.
  - ✓ Use-based testing: It begins the construction of the system by testing those independent classes that are very few of server classes. After the independent classes are tested, the next layer of classes, called dependent classes, that use the independent classes are tested.
  - ✓ Cluster testing: It includes a cluster of collaborating classes which is exercised by designing test cases that attempt to uncover errors in the collaborations.

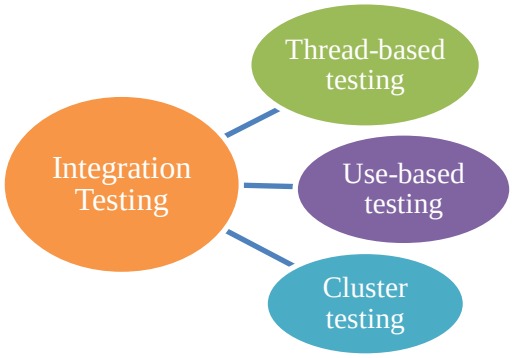


Fig 1.3 Integration Testing Strategies

- System Testing: After all the units of a program have been integrated together and tested, system testing is taken up. System tests are designed to validate a fully developed system to assure that it meets its requirements. As shown in Fig 1.4, there are four different strategies for system testing.
    - ✓ Recovery testing indicates how well does the system recover from faults.
    - ✓ Security testing verifies that whether protection mechanisms built into the system will protect from unauthorized access or not.
    - ✓ Stress testing checks for any abnormal load on the system.
    - ✓ Performance testing investigates the run-time performance of an integrated system [4].

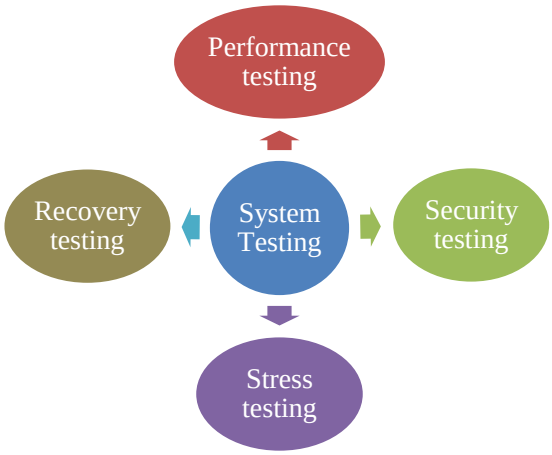


Fig 1.4 System Testing Strategies

1.3 Graphical Methods

A graph G = (N, E) is defined as a finite set of nodes N and a finite set of edges E. The graphical methods provide the structural representation of the system that facilitates testing the logical development of the program [6].

A dependence graph represents program features and dependencies between many objects. There are many dependence graphs such as Data Flow Graph (DFG), Program Dependence Graph (PDG), System Dependence Graph (SDG), Extended System Dependence Graph (ESDG), Call- based Object-Oriented System Dependence Graph (COSDG), *etc*. For example, Figure 1.5 represents a dependence graph where edges between nodes A, B, C, D and E are representing dependence relationships. For each edge in the graph, the node (statement) at the tail of the edge must complete its execution before the node at the head may begin. For example, before node C should execute, nodes A and/or B must complete their execution.

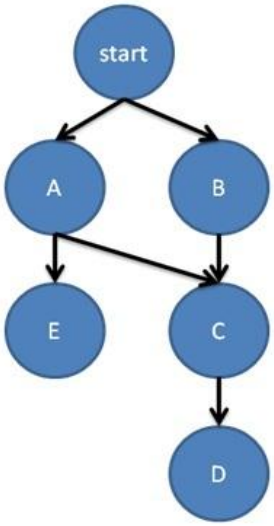


Fig 1.5 Dependence Graph [61]

1.3.1 Control Flow Graph (CFG)

A control flow graph describes the sequence in which the instructions of a program will get executed. A CFG (or flow graph) G is defined as a finite set of nodes N and a finite set of edges E where an edge (i, j) in E connects two nodes n_i and n_j in N. In a flow graph

of a program, each basic block becomes a node and edges are used to indicate the flow of control between blocks.

For example, in Fig 1.6, blocks (nodes) are labeled such that block b_i corresponds to node n_i. An edge (i, j) connecting basic blocks b_i and b_j implies that control can go from block b_i to block b_j [7].

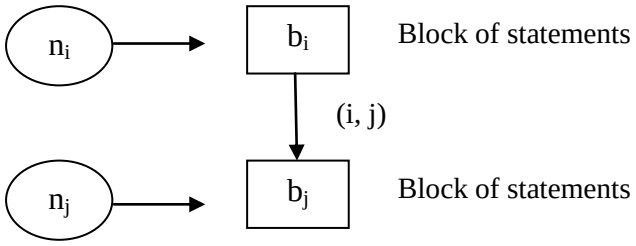


Fig 1.6 Block/Node representing set of statements

Most of the programs are constructed with the three types of constructs namely sequence, selection and iteration. Fig 1.7 summarizes how the CFG for these three types of constructs can be drawn. The CFG representation of the sequence and decision types of statements is straightforward. For the Iteration type constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore control flows from the last statement of the loop to the top of the loop [2].

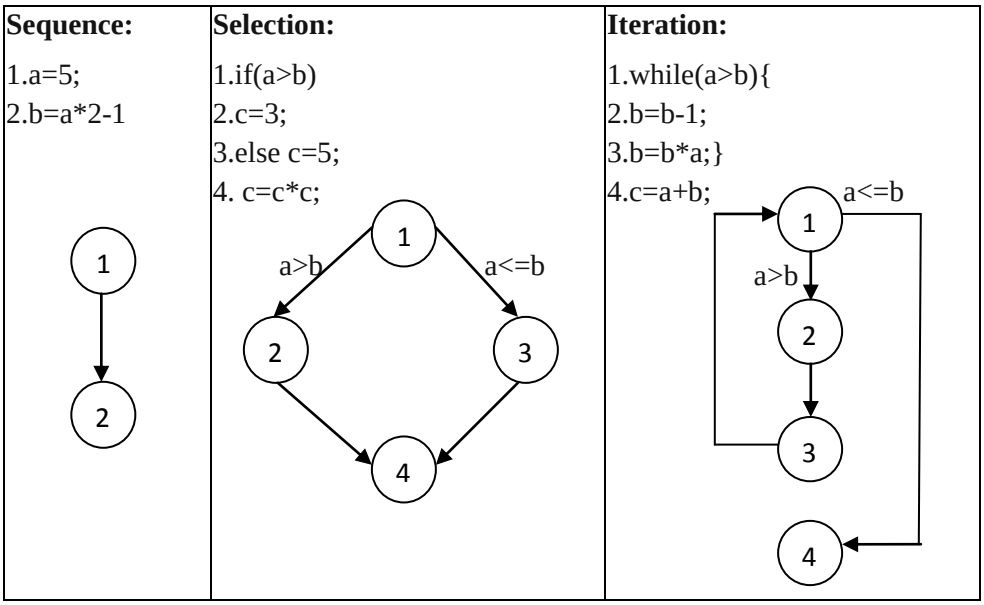
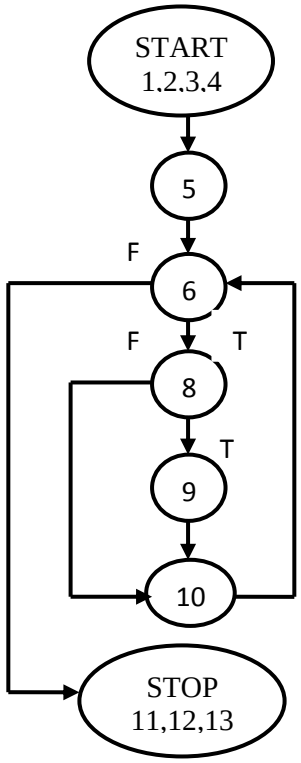


Fig 1.7 CFG for Sequence, Selection and Iteration type of constructs [2]

Consider an example program in Fig 1.8(a), where *x* and *count* are integer variables. The while loop will run until the value of *count* is greater than zero and if the value of *x* is smaller than 20, then the value of *x* will be incremented by 1. If the value of *x* is not smaller than 20, then the value of *count* will be decremented by 1. The Control Flow Graph of this program appears in Fig 1.8(b). The node *start* is the only entry node and the node *stop* is the only exit node of the given Control Flow Graph.

```
1.  class Example
2.  {
3.     public static void main(String args[ ])
4.    {
5.        int x=10, count=5;
6.        while(count>0)
7.       {
8.          if(x<20)
9.            x=x+1;
10.         count=count-1;
11.       }
12.    }
13.  }
```

(a)



(b)

Fig 1.8 (a) An Example Program (b) Control Flow Graph of example program

Considering another example for CFG in Fig 1.9(a), the code snippet present will take two integers x and y as input parameters. There are total of 16 lines in this program including start and stop. The execution of the program begins at line 2 and moves through lines 3 and 4 to line 5 containing an *if* statement [8]. Considering that there is decision at line 5, control could go to one of the two possible destinations at lines 6 or 8. Thus, the sequence of statements starting at line 2 and ending at line 5 constitutes a basic block. Its

only entry point is at line 2 and the only exit point is at line 5. As shown in Fig 1.9(b), the program contains a total of nine blocks numbered sequentially from 1 to 9.

```
1.  Start
2.  int x, y, power;
3.    float z;
4.    input (x, y);
5.    if(y<0)
6.      power=-y;
7.    else
8.      power=y;
9.    z=1;
10.   while (power!=0)
11. {
12.     z=z*x;
13.     power=power-1;
14. }
15.   if(y<0)
16.   z=1/z;
17.   output(z);
18. Stop
```

(a)

Block	Lines	Entry Point	Exit Point
1	2, 3, 4, 5	2	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	12, 13	11	14
7	15	15	15
8	16	16	16
9	17	17	17

(b)

Fig 1.9 (a) An Example Program (b) Basic Blocks of the Program [8]

Fig 1.10(a) depicts the control flow graph with statements in each block and in Fig 1.10(b), a circular node will represent the set of statements as represented by the rectangular block in Fig 1.10(a).

1.3.2 Control Dependence Graph

Control Dependence Graph shows dependencies between statements with the help of control conditions. As shown in Fig 1.11, for statements X, Y, Y' and Z in a program, if Y is control dependent on X, then X must have two exit paths where one of the exit paths always results in the execution of Y and the other exit path may result in Y' (where Y will not be executed) [9].

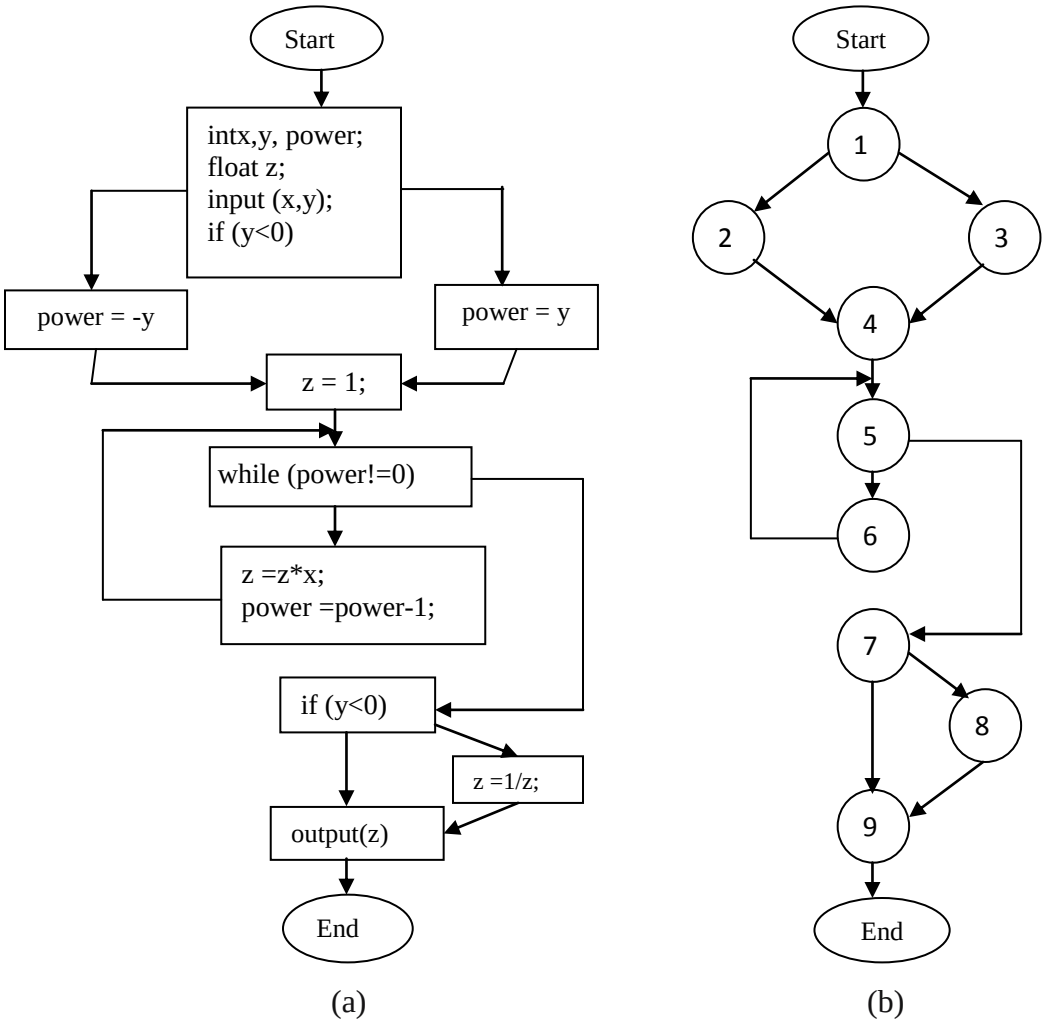


Fig 1.10 CFG of a program (a) Block representing set of statements (b) Nodes representing set of statements [8]

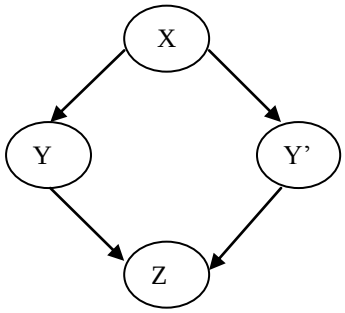


Fig 1.11 Control Dependence Graph

Consider an example program in Fig 1.12(a), which consists of statement 1 to statement 9. u( ) and v( ) are two methods. The while loop will execute till the value of n will be

greater than zero. If the value of variable x will be greater than zero, then variable b will be assigned the value of variable a else variable c will be assigned the value of variable b.

Fig 1.12(b) represents Control Dependence Graph of the example program. Statement 1, 2 and 9 are control dependent on Start of the program. Statement 3 and 4 are control dependent on statement 2 and similarly, statement 5 and 7 are control dependent on statement 4. Control dependency is represented with regular arrow lines in the graph.

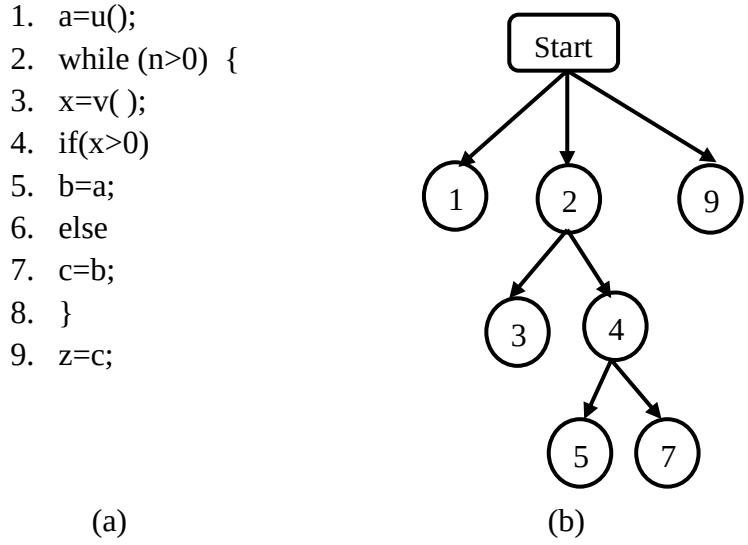


Fig 1.12(a) An Example Program (b) its Control Dependence Graph [63]

1.3.3 Data Dependence Graph

Data Dependence Graph represents flow of data from one statement to another statement. As shown in Fig 1.13, for two statements X and Y in a program, Y is data dependent on X, if X defines a variable v, Y uses v, and there exists a directed path from X to Y along which there is no intervening definition of v [9].

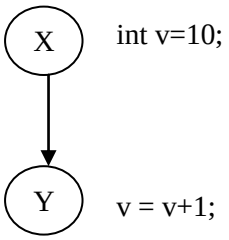


Fig 1.13 Data Dependence Graph

Fig 1.14 represents Data Dependence Graph of the example program given in Fig 1.12(a). Statement 5 is data dependent on statement 1, statement 4 is data dependent on statement 3, statement 7 is data dependent on statement 5 and statement 9 is data dependent on statement 7. Data dependency is represented with dotted arrow lines.

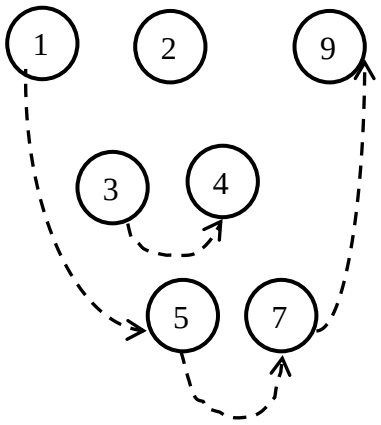


Fig 1.14 Data Dependence Graph of example program in Fig 1.12(a) [63]

1.3.4 Program Dependence Graph (PDG)

In Program Dependence Graph, the nodes will represent statements and predicate expressions. The edges incident to a node represents data values and the control conditions [10]. A PDG represents both control dependence and data dependence in a single graph. It consists of vertices and edges where vertices represent the program statements and edges represent the data dependency and control dependency [11].

Control dependency edges are labeled either true or false and the source of a control dependency edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex *v1* to vertex *v2* means that during execution, whenever the predicate represented by *v1* is evaluated and its value matches the label on the edge to *v2*, then the program component represented by *v2* will be executed. A data dependency edge from vertex *v1* to vertex *v2* means that the program's computation might be changed if the relative order of the components represented by *v1* and *v2* are reversed.

A program dependence graph contains a flow dependency edge from vertex *v1* to vertex *v2* iff all of the following conditions hold [10, 11]:

- *v1* is a vertex that defines variable *x* and *v2* is a vertex that uses *x*.
- Control that reach *v2* after *v1* through an execution path in which there is no intervening definition of *x*.

Consider the program given in Fig 1.15 where the code fragment is used to calculate the factorial of a number [62]. The execution of statements 11 and 12 is dependent on the control predicate at statement 9. The statement 11 is data dependent on statements 7,8, 12 and itself. Fig 1.16 represents the corresponding Program Dependence Graph of the program given in Fig 1.15, where control dependence edges are represented as bold lines and data dependence edges are represented by light colored regular lines.

```
1.  class Factorial
2.  {
3.    public static void main(String args[ ])
4.   {
5.     int fact;
6.     int n;
7.     n =4;
8.     fact = 1;
9.     while (n != 0)
10.   {
11.     fact = fact * n;
12.     n = n-1;
13.   }
14. }
15. }
```

Fig 1.15 A Sample Java Program to calculate factorial of a number [62]

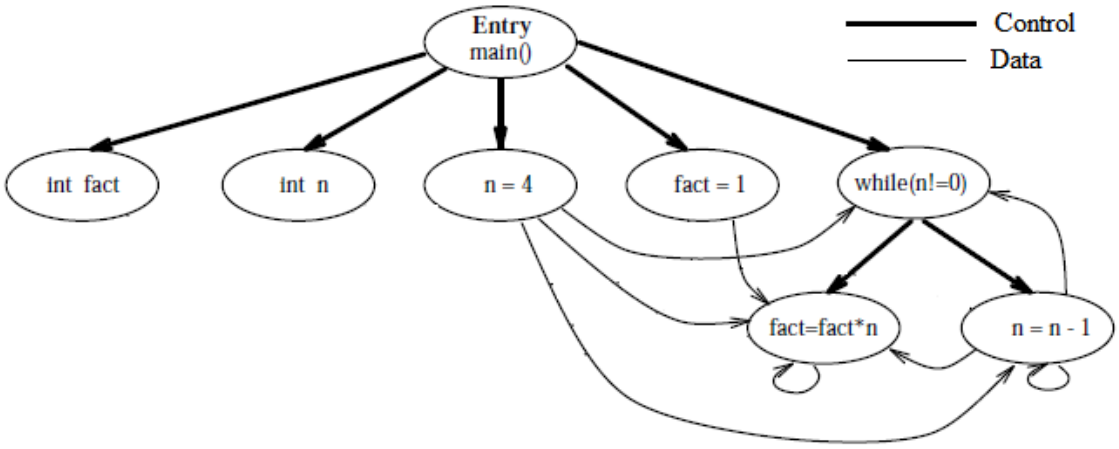


Fig 1.16 The Program Dependence Graph corresponding to the program in Fig 1.15 [62]

1.3.5 System Dependence Graph (SDG)

The System Dependence Graph (SDG) is an extension of the Program Dependence Graph (PDG) and represents a program that consists of multiple procedures and involves procedural calls. SDG includes a Program Dependence Graph to represent a system's main program, Procedure Dependence Graphs to represent a system's auxiliary procedures and some additional edges to interconnect these graphs [9].

Each Procedure Dependence Graph contains an entry vertex that represents entry into the procedure. To model parameter passing, SDG associates each procedure entry vertex with formal-parameter vertices namely a formal-in vertex for each formal parameter of the procedure and a formal-out vertex for each formal parameter that may be modified by the procedure [13].

SDG associates each call-site in a procedure with a call vertex and a set of actual-parameter vertices with an actual-in vertex for each actual parameter at the call-site and an actual-out vertex for each actual parameter that may be modified by the called procedure. A call edge connects a call vertex to the entry vertex of the called procedure's dependence graph. Parameter-in and parameter-out edges represent parameter passing. Parameter-in edges connect actual-in and formal-in vertices and parameter-out edges connect formal-out and actual-out vertices [12].

```
public static void main(String args[])
{
int i = 1;
int sum = 0;
while (i<11) {
        sum = add(sum, i);
        i = add(i, 1); }
System.out.println("sum = \n" + sum);
System.out.println("i =\n" +  i);
}
static int add(int a, int b)
{
return (a+b);
}
```

Fig 1.17 An Example Program [70]

Fig 1.17 depicts a program to find sum of numbers from 1 to 10. This program uses two methods namely main( ) and add( ) [70]. Fig 1.18 shows the System Dependence Graph of this program representing the flow within two methods. As the Program Dependence Graph can only represent the flow in a single procedure but System Dependence Graph is able to represent multiple procedures.

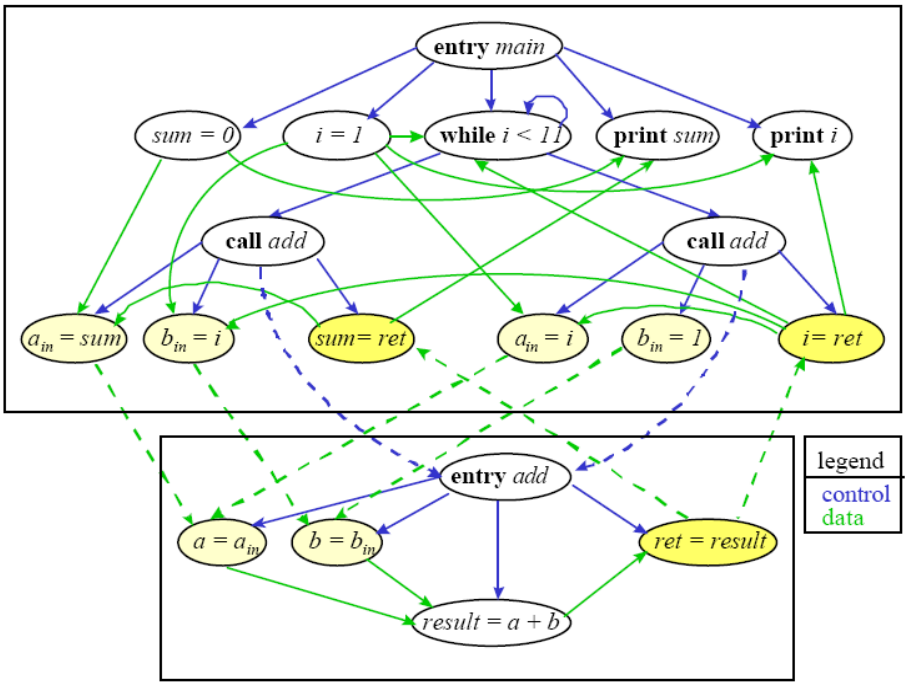


Fig 1.18 The System Dependence Graph of the example program in Fig 1.17 [70]

```
1.int main()                    11. int func(int a, int b, int c)
2. {                            12. {
3.   int sum;                   13.   c = c + 1;
4.   int x;                     14.    if (c == 4) {
5.   int y;                     15.      c = 4 * c;
6.    x=4;                      16.      return c;
7.    y=6;                      17.    }
8.   sum=func(x,y,4);           18.   a = a + c;
9.   sum=sum+1;                 19.   return a;
10. }                           20. }
```

Fig 1.19 An Example Program [62]

Consider the program presented in Fig 1.19, that consists of two functions namely main( ) and func( ) [62]. In method named main( ), statement 3, 4, 5, 6, 7, 8 and 9 are control

dependent on entry node (Statement 1). There are three actual-in parameters namely x, y and 4 which are control dependent on statement 8. Actual-in parameters x and y are data dependent on statement 6 and 7 respectively. The call from statement 8 is going to statement 11. In method named func( ), statement 13, 14, 18 and 19 are control dependent on statement 11. There are three formal-in parameters namely a, b and c. There are parameter-in edges going from actual-in parameters to formal-in parameters. Also, statement 15 and 16 are control dependent on statement 14. Statement 14, 15 and 18 are data dependent on statement 13. Statement 18 in program will be executed, if and only if, the return statement at line 16 is not executed. Therefore, statement 18 is control dependent on statement 14 (the return statement's control predicate). Figure 1.20 presents the System Dependence Graph of the program that is shown in Fig 1.19.

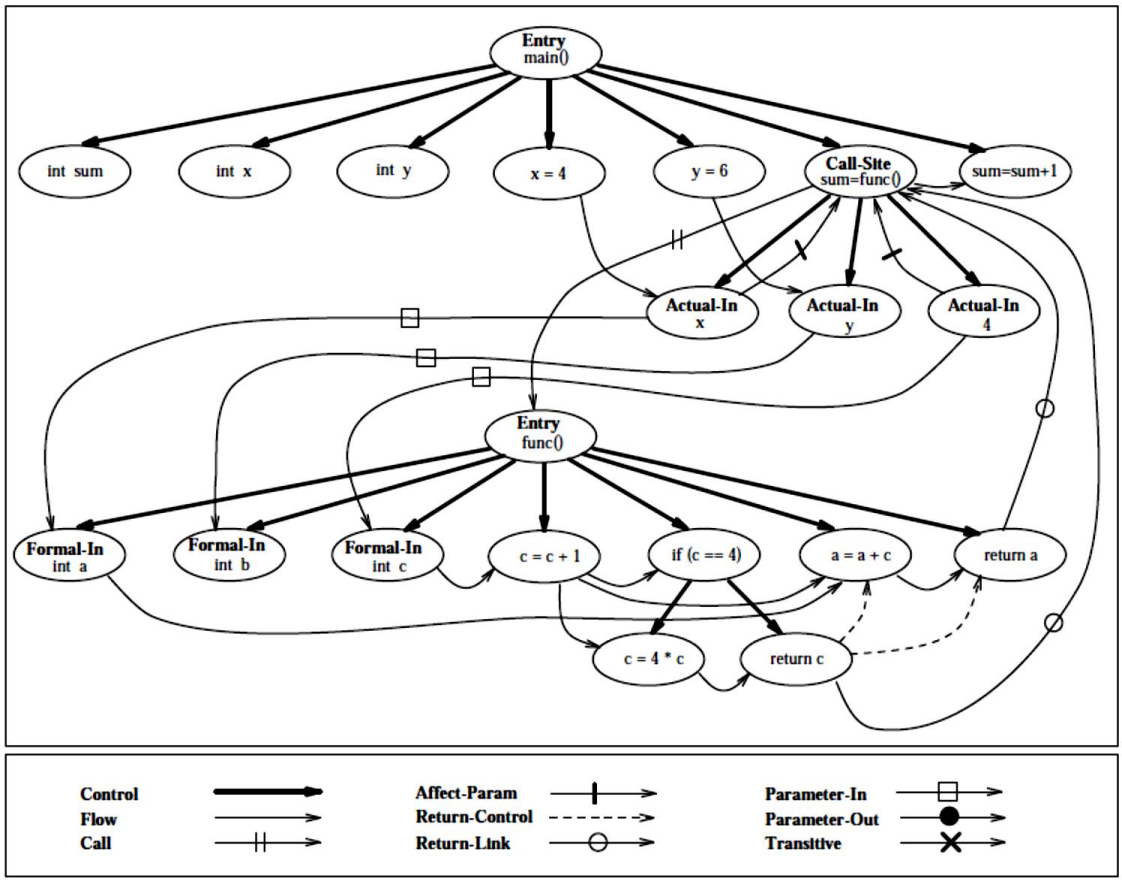


Fig 1.20 The System Dependence Graph corresponding to the program in Fig 1.19 [62]

1.4 Thesis Layout

The rest of the thesis is organized in the following manner:

- Chapter 2 reviews the related work done in the area of various graphical methods used in test case generation along with their advantages and disadvantages.
- Chapter 3 specifies the concise problem statement.
- Chapter 4 describes the methodology of work conducted.
- Chapter 5 provides experimental results of the work done.
- Chapter 6 gives the conclusion of the thesis with the future scope of the topic.

Chapter 2

Literature Survey

2.1 Graphical Methods

The graphical methods give the structural representation of the system that facilitates testing the logical development of the program. It also partitions the system into a number of smaller subsystems from the structural description [3]. Test case generation is a method to generate test cases where a test case is a description of a test, independent of the way a given system is designed. Several researchers have proposed different techniques to generate test cases and in most of the approaches proposed so far, either the source code or the object/byte code is used to build the data structures for test case generation. Some of the approaches use some form of flow graph or call graph to represent the different features of a program and to generate test cases. Ferguson and Korel [14] divided these methods in three classes: random, path-oriented, and goal-oriented test data generation.

The selection of a path can largely affect the whole process of test data generation. Till date various graphical methods are proposed in the field of software testing. Work in this area of test case generation started in the year 1982. As the years passed by, the industry started realizing the benefits of using graphs in test case generation new models and approaches came up. Taking into consideration various ideas put forth by the industry and academicians, major focus of this literature is to provide review of all the graphical approaches used for the test case generation by providing a chronological sequence of all the research activities done in this area starting from year 1982 till date.

2.2 Test Case Generation

2.2.1 Using Def-Use Graph

The def-use graph is a program graph for representing the status of a variable. It consists of a set of variable definitions, computations and predicative uses assigned to each node and edge. Fosdick and Osterwall proposed data flow graph for automatic error detection

and data flow analysis in software reliability [15]. Rapps andWeyuker presented a control flow criteria i.e. All-Paths (path coverage), All-Edges(branch coverage), and All-Nodes (statement coverage). The All-Paths criterion requires that a path set contain every path through a def-use graph. The All-Edges and All-Nodes criteria require that a path set cover every edge and every node respectively [16, 17]. The authors have also created the def-use graph by associating a set with each edge and two sets with each node where *def(i)* is the set of variables with global definition in node *i, c-use(i)* is the set of variables with global *c-use* in node *i* and *p-use(i,j*) is the set of variables for edge *(i,j*) containing *p-use*. A family of path selection criteria was also introduced which is shown in Fig 2.1

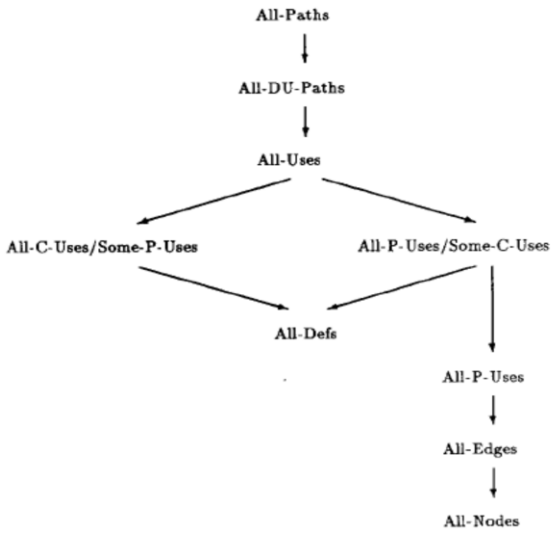


Fig 2.1 A family of path selection criteria [16]

Clarke et al. proposed a new family of path selection criteria with some modifications in the family proposed by Rapps and Weyuker. The authors compared the strengths and weaknesses of both the families and suggested minor changes to improve their performance [18]. But the authors did not implement fault detection capabilities. Souter and Pollock proposed Points-To-Escape graph for the construction of contextual def-use associations for object oriented programs [19]. The researchers used a prototype testing framework for structural testing and object flow analysis. Vincenzi et al. introduced def-use graph (DUG) for control and data flow analysis at intra-method level [20]. The authors also developed a tool JaBUTi (Java Bytecode Understanding and Testing) for coverage analysis and structural testing of Java programs using def-use graph.

2.2.2 Using Control Flow Graph (CFG)

A control flow graph is a directed graph where nodes represent the basic blocks and the edges represent control flow paths [21]. Frankl and Weyuker introduced control flow and data flow testing criteria for programs written in Pascal and also defined a new family of adequacy criteria called feasible data flow testing criteria which is derived from the data flow testing criteria [22].

Agrawal proposed branch coverage and testing of programs to find subsets of nodes and edges in a flowgraph [23]. The author introduced dominator relationships among superblocks which are used to identify a subset of the super blocks and these techniques reduce object code size, runtime overhead and cost of coverage testing of programs. Dominator relationships were represented using Block Dominator Graph and Edge Dominator Graphs. The author proved the effectiveness of his technique by incorporated them into a prototype tool SPYDER which provides dynamic program slicing facilities for C programs. The author also presented a technique to find a small subset of a program statements and decisions with the property that covering the subset implies covering the rest [24]. The author also presented algorithm to construct Global Dominator Graph which shows dominator relationships among mega blocks at inter-procedural level. Global dominator graph is the combination of block dominator graphs. Inter-procedural jump statements are also handled using this graph.

An EventGraph is a control flow graph of a program unit and interactions are represented between the program units such as procedures and functions. The Event InterActions Graph (EIAG) is used as a model for concurrent programs and it consists of Event Graphs and Interactions. Katayama et al. proposed a method for structural testing of concurrent programs written in Ada programming language for test case generation and executionof the programs [25]. Fig 2.2(a) shows a sample program in Ada language and its corresponding EIAG is shown in Fig 2.2(b). After generating test-cases on the EIAG, a method for selecting test-data has described. The measures to cope with the infeasible test-cases are generated and clarified. The authors have solved the non-deterministic execution problem of concurrent programs by inserting synchronization points in the

source code. They have also implemented the tool TCgen (Test-Case generator written in Ada language) that automatically generates copaths, which are test-cases for a concurrent program.

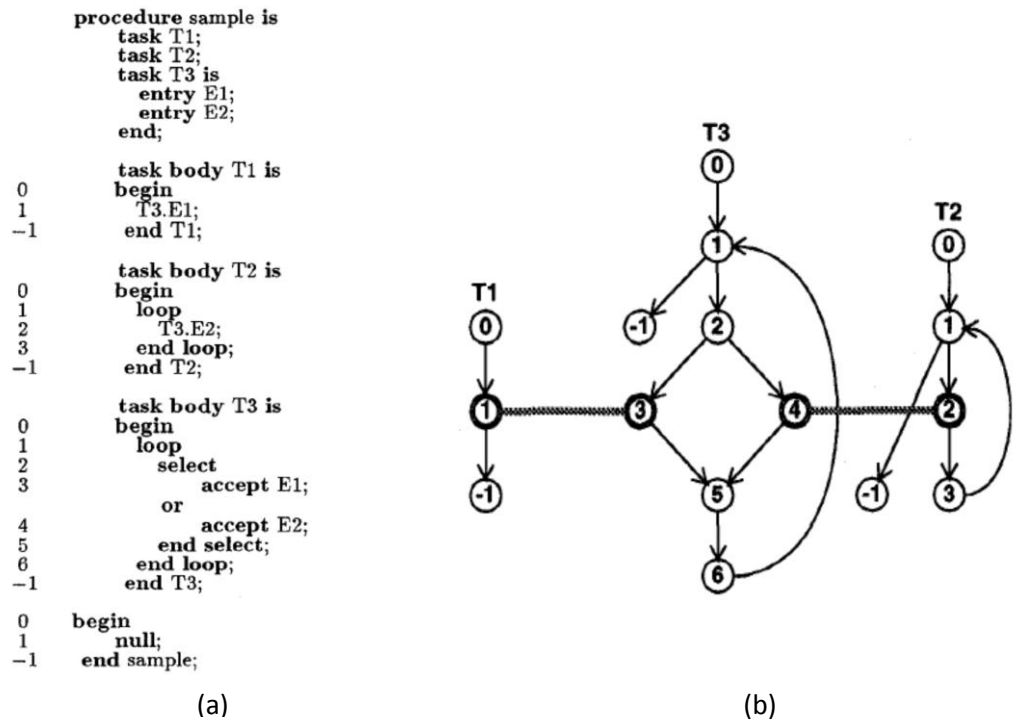


Fig 2.2(a) A sample program in Ada language (b) its EIAG [25]

Edvardsson presented a survey on automatic test data generation using control flow graph [26]. The author described basic notions and concepts of test data generation and how a test data generator system works. The author also identified and explained some problems of automatic generation. Martena et al. implemented a system to generate test cases using CFG for inter-class testing [27]. The shortcoming of this technique is that it does not accept code with inheritance, polymorphism and exception handling.

A Class Specification Implementation Graph (CSIG) is a graphical representation which shows a class from two distinct perspectives, namely the class as specified and the class as implemented. In this graph, each class method is represented by two control flow graphs where one graph visualizes at control flow as specified and other graph at control flow as implemented. Beydada and Gruhn presented the application of CSIGs in

regression testing. The control flow graphs are the main constituents of a CSIG and therefore testing techniques implemented on control flow graphs can also be combined with CSIGs with some modifications. A test suite reduction strategy has been incorporated to the CSIG construction algorithm to decrease the total number of test cases required. The authors have shown how CSIGs can be combined with an existing test case selection algorithm for regression testing [28]. The authors also used CSIGs for integrated class-level black and white-box testing. Both the specification of a method and its implementation were represented as control flow graphs which allows black and white box testing by structural techniques [29].

A Java Interclass Graph (JIG) is a new control flow representation for AspectJ software which captures the semantic inter-relations of aspect-related interactions. A JIG contains a CFG for each method that is internal to the set of classes [30]. Each call site is expanded into a call node and a return node and the call node is linked with the entry node of the called method. There is a path edge between the call node and the return node to represent the path through the called method. Xu and Rountev proposed a new regression test selection technique for AspectJ programs. Aspect-oriented software development is a popular approach for modularizing cross-cutting concerns, which simplifies software maintenance and evolution. The executable code of an AspectJ program is Java bytecode produced by an AspectJ compiler [30]. The authors have developed AspectJ Inter-Module Graph (AJIG) which extends some features of JIG. The authors also presented a two-phase graph traversal algorithm in which the first phase does the inter-procedural traversal and second phase does the intra-procedural comparison.

Jenny Li et al. proposed a new Super Nested Block (SNB) method to reduce instrumentation run-time overhead in coverage testing using control flow graph [31]. The researchers implemented the SNB method with an automatic generated on-line instrumenter of a code coverage testing tool and found that instrumentation overhead was less than 1% of overall execution time.

Mouy et al. proposed a technique that deals with the difficulties of structural unit testing to test functions which call other functions [32]. The authors incorporated the functional

information on the called functions within the structural information. The authors used Control Flow Graph (CFG) which may be viewed as an extension of the classic CFG. This CFG allows to characterize test selection criteria ensuring the coverage of the source code of the function under test. The authors also described how to automate test data generation with grey-box (combinations of black-box and white-box) test selection strategies and applied the results using PathCrawler tool examples in the C language.

2.2.3Using Program Dependence Graph (PDG)

The PDG represents a program as a graph. It is a graph in which the nodes are statements and predicate expressions. The edges incident to a node representing data values on which the node's operations depend and the control conditions on which the execution of the operations depends [10, 11].

A PDG represents both control dependence and data dependence in a single graph. For statements X and Y in a program, if X is *control dependent* on Y then there must be at least two paths out of Y. In this, one path always causes X to be executed and the other path may result in X not being executed. A *data dependence* exists between statements X and Y in a program if X defines a variable *v*, Y uses *v* and there is a path from X to Y in the program on which *v* is not defined [33, 34].

Rothermel and Harrold implemented PDG for regression testing in object-oriented software. The researchers presented an algorithm that constructs class dependence graphs (ClDG) for classes and application programs. The researchers used these graphs to determine which tests can cause a modified class to produce different output than the original [12, 33]. But the researchers did not considered polymorphism and dynamic binding in their approach.

There were also few other graphs that extended the features of PDG for program slicing such as Object Program Dependence Graph (OPDG) [64] , Dynamic Object Program Dependence Graph(DOPDG) [64], Efficient Dynamic Object Program Dependence Graph (EDOPDG) [65] and so on.

2.2.4 Using System Dependence Graph (SDG)

The System Dependence Graph (SDG) is an extension of the Program Dependence Graph (PDG) and represents a program that consists of multiple procedures and involves procedural calls. SDG models a language in which parameters are passed by value and where a complete system consists of a single (main) program and a collection of auxiliary procedures [11].

Horwitz et al. presented SDG for inter-procedural slicing and make use of context-free grammer in constructing SDG [11]. They presented all the dependency relationships in SDG and PDG. An algorithm to implement inter-procedural slicing using SDG was also discussed. The cost of constructing SDG for slicing in various contexts was also determined.

Larson and Harrold extended the SDG of Horwitz et al. to represent Object-Oriented programs [12]. They have constructed Class Dependence Graphs (ClDG) for each class in an object-oriented program. A ClDG captures the control and data dependence relationships that can be determined about a class without knowledge of calling environments. Each method in a ClDG is represented by a procedure dependence graph. The ClDG construction expands each method entry by adding formal-in and formal-out vertices similarly as procedure dependence graphs.

Liang and Harrold present an SDG for object-oriented software that is more precise than previous representations and is more efficient to construct than previous approaches [13]. Based on this new SDG, they introduced the concept of object slicing and an algorithm to implement this concept. Object slicing enables the user to inspect the statements in the slice, object-by-object,and is helpful for debugging and impact analysis.

Mohapatra et al. presented a technique for dynamic slicing of Object-Oriented programs, which extends the System Dependence Graph (SDG) [66]. The graph is known as Extended System Dependence Graph (ESDG) which handles the features of object-oriented programs such as polymorphism, inheritance etc. Their algorithm is named as Edge Marking Dynamic Slicing (EMDS) because it is based on marking and unmarking the edges of the ESDG.

Zhao presented a Java-based graph that encapsulates the benefits offered by the previous approaches of SDG. The Graph was named as Java System Dependence Graph (JSDG) and it enables the representation of Java-specific features such as interfaces, packages and single inheritance [67]. Walkinshaw et al. extended this Java-based graph that is known as Java System Dependence Graph (JSysDG) [68]. A JSysDG is a multigraph which maps out control and data dependencies between the statements of a Java program.

Xi et al. presented an approach of Coarse-grained Dynamic Slice for Java Program [69]. This technique uses AspectJ code tracing tactic to gather method execution traces, which comprises information of method calls. Dynamic Java System Dependence Graph (DJSDG) is used for the intermediate representation and the slice computation is also done on this graph.

2.2.5 Using ddgraph

The ddgraph is a reduced flow graph whose arcs are associated with program segments. Bertolino and Marre presented a generalized algorithm that finds a path cover for a given program flowgraph [35]. The authors have presented two important relations between ddgraph arcs namely dominance and implication. The authors have created dominator tree and implied tree for dominance and implication paths respectively. An algorithm is proposed that covers all the unconstrained arcs of a given ddgraph and the paths are derived one at a time, each path covering at least one uncovered unconstrained arc. The authors also proposed procedure for sub-ddgraphs. This algorithm was simple and flexible because it can satisfy different policies in the construction of path coverage.

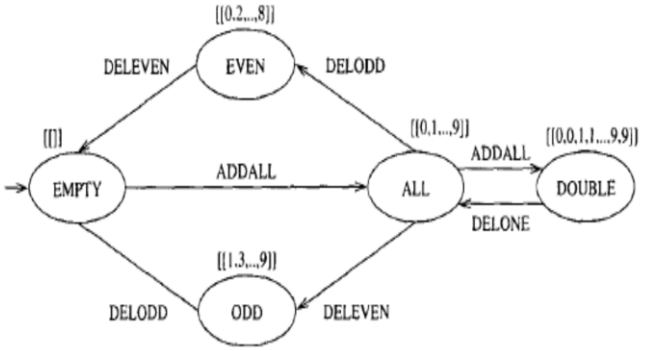
2.2.6 Using TestGraph

A testgraph is a directed graph in which the nodes have unique identifiers and there is a designated start node. McDonald and Strooper presented ClassBench methodology for the testing of inheritance hierarchies in C++ using the testgraph. ClassBench framework provides a graph editor for testgraphs, graph traversal algorithms that automatically traverses a stored testgraph and it also supports debugging and regression testing [36].

Fig 2.3(a) shows a *Bag* class declaration code written in C++. *Bag* provides access to a bag of integers. The constructor *Bag(n)*creates an empty bag, the parameter n represents the maximum size of the bag. Assuming that *cut* is an object of type *Bag, cut.add(x)* adds *.r* to *cut; cut.add(x)* does not change *cut* if *cut* is *full***.** If *x* is in *cut, cut.detach(x)* removes one copy of *x* from *cut:* otherwise, there is no change. *cut.getItemsInContainer* returns the number of elements in *cut***,** and *cut.hasMember(x)* returns true or false according to whether *x* is in *cut***.**

```
class Bag: public
Collection {
public:
Bag(int);
void add(int);
void detach(int);
intgetItemsInContainer( )
const;
inthasMember(int) const;
private:
…….}
```

(a)



(b)

Fig 2.3(a) Code for Bag class declaration (b) Testgraph for Bag [36]

The authors generated the test cases by repeatedly traversing the testgraph beginning at the start node. The three types of coverage have been considered: node coverage, arc coverage and path coverage. The authors have shown that a high level of reuse of testing information can be possible with ClassBench methodology. This methodology ensures that classes which reuse code via inheritance can be adequately tested without the need to develop a new test suite.

2.2.7 Using Class Graph

A class graph captures relations between classes where nodes are represented as classes and edges represent the relations between classes. Chen and Lu presented a class graph with algebraic expressions for a class which contains a syntax declaration and a semantic

specification, which specify the structure and the behavioral properties of the operations involved in the class. Fig 2.4(a) shows an algebraic specification for the class of integer stack and its class graph appears in Fig 2.4(b). According to the definitions in [37], in Fig 2.4(a), *new* is a creator, *push* is a constructor, *pop* is a transformer, *top* and *empty* are observers. By the definition, a normal form must be a sequence starting from a creator, and followed by several constructors. Hence, it corresponds to a path in the class graph starting from the node *new* and ending in the node *push.*

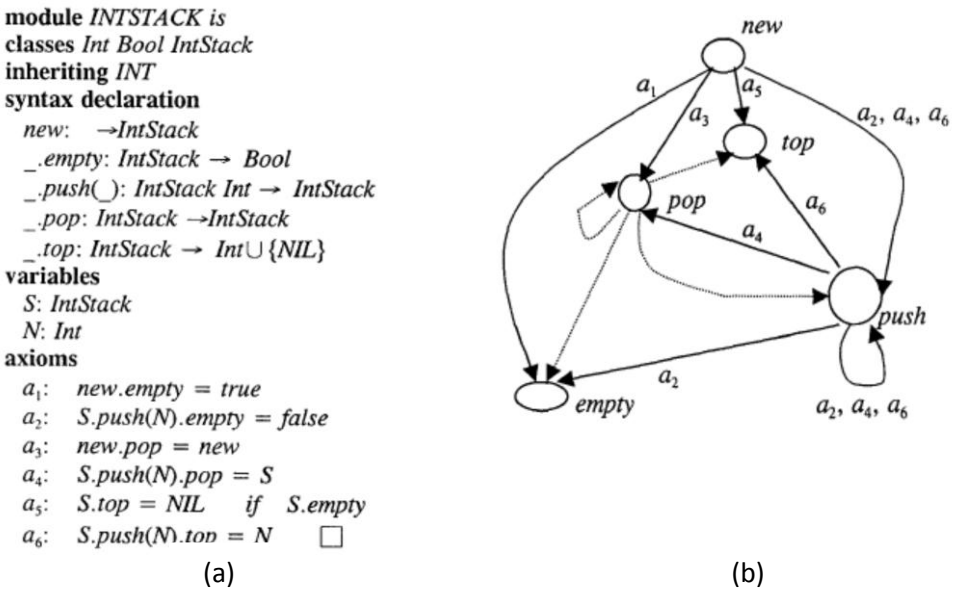


Fig 2.4(a) An algebraic specification for the class of integer stack (b) its Class Graph [37]

Leung and Wong proposed an enhancement of the classification tree by replacing the two kinds of relation by a Class-Class relation. The researchers described the Class Graph approach and compared it with the classification tree (CT) and presented how test cases can be derived from class graph in a more efficient way [38]. The CT method is unable to fully express information obtained from specifications and it can only represent two kinds of relations namely classification-classes and class-classification. It cannot express the relation between classes directly. In order to express the class-class relation, the researchers proposed class graphs which cover the information from specifications better than CT method. The class graphs can also handle more complicated relations between classifications and their syntax is simpler than CT. These can capture more precise

information because it eliminates the need of human decisions in selecting valid test cases.

2.2.8 Using Application Call Graphs

A call graph is a representation of calling relationships among methods [39]. Zhang and Ryder explored various approaches to generate more accurate Application Call Graphs for Java. A new data reachability algorithm is proposed and implemented to resolve library callbacks accurately [40]. This fine-tuned data reachability algorithm results in fewer unauthentic callback edges as compared to a simple algorithm that generates an application call graph by traversing the whole program call graph. Application Call Graphs can capture the calling relationships more accurately among application methods. The authors have shown this graph with Call-chain based testing and def-use pair based testing.

2.2.9 Using Call based Object-Oriented System Dependence Graph (COSDG)

Call-based Object-Oriented System Dependence Graph (COSDG) is a directed, connected multigraph G = (V, E), consisting of a set of vertices V and a set of edges E. A vertex represents one of the three categories of vertices, namely, statement vertices, entry vertices, and parameter vertices. An edge represents one of the seven categories of edges, namely, control dependence edges, data dependence edges, parameter dependence edges, method call edges, summary edges, class member edges, and inheritance edges [9].

Fig 2.5 shows various graphical symbols used to represent different types of edges and vertices in COSDG [41].

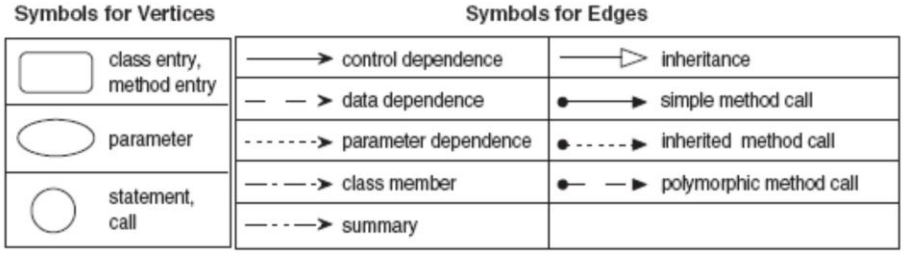
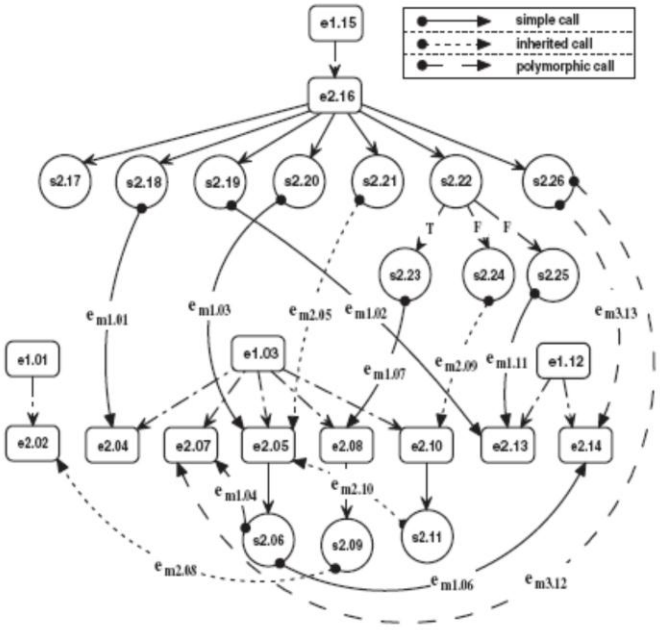


Fig 2.5 Graphical Symbols used in COSDG [41]

Najumudheen et al. proposed a dependence-based representation for object-oriented programs as COSDG. COSDG captures important object-oriented features such as class, inheritance, and polymorphism. It also includes details of method visibility in aderived class, and different types of method call edges namely simple, inherited and polymorphic [9, 41]. The authors also proposed an algorithm for the construction of COSDG and explained its working with an example.

```
01: class A{                          15: class X  {
02: void m0( ) { }       //base       16:   void main (String[ ] args)   {
    {                                 17:   int x=args.length;
03:   class B extends A  {            18:   B Obj_B=new B ( );        //simple
04:     B ( ) { }            //constr. 19:   C Obj_C=new C ( );        //simple
05:     void m1 ( )                   20:   Obj_B.m1( );              //simple
06:       {  m2 ( );  }               21:   Obj_C.m1( );            //inherited
07:     void m2 ( )  { }              22:    if (x==1)
08:     void m3 ( )  { }              23:      Obj_B.m3( );           //simple
09:       {  m0 ( );  }                     else
10:     void m4 ( )                   24:      Obj_C.m4( );         //inherited
11:       {  m1 (  );  }              25:  Obj_B=new C( );           //simple
    }                                      }
12: class C extends B  {              26: Obj_B.m2( );          //polymorphic
13:   C ( )   {  }          //constr.       }
14:   void m2 (  )  {  }    //ovrdn.
```



(b)

Fig 2.6(a) An Example Java Program (b) Various types of method calls in COSDG [41]

Fig 2.6(a) shows a sample Java program and Fig 2.6(b) represents its COSDG. The three types of method calls are represented using COSDG in Fig 2.6(b) namely simple, inherited and polymorphic method calls, created at different call sites in the program.

2.3 Tools

There are different tools available, which are listed and compared in Table 1.

Table 2.1 Tools Comparison

Year	Tool Name	Language Support	GUI Support	Type of Coverage
1987	ASSET [22, 42]	Pascal	N	Data Flow
1988	SiemensTSL[43,44,45]	Ada	N	Data Flow and Control Flow
1993	SPYDER [23, 46]	C	Y	Block and Edge coverage
1995	ANTLR [9, 47]	C, C++, Java	Y	Control and data flow
1996	TCgen [25, 48]	Ada	N	Test Coverage
1996	PiSCES [20, 49, 50]	C, C++, Java	Y	Statement, Decision and Code Coverage
1997	JCover 2.1 [20, 51]	Java	Y	Statement, branch, method and class coverage
1998	xATAC [23, 52]	C, C++	Y	Statement, Decision and control flow Coverage
1999	TCAT [20, 53]	C, C++, java	Y	Statement, Decision and Test coverage
2001	TATOO [19, 54]	Java	Y	Code and Object coverage
2002	JTest [20,55,56]	Java	Y	Statement, branch and code
2007	JUnit [20, 57, 58]	Java	Y	Black box testing
2003	JaBUTi [20, 59]	C, C++, Java	Y	Data flow and Control Flow
2005	PathCrawler [32, 60]	C, C++	Y	Path Coverage

2.4 Comparison of Graphical Methods

From the given literature, the advantages and disadvantages of various graphical methods used for test case generation have been observed. These features are listed in Table 2.2.

Table 2.2 Graphical methods comparison

S No.	Approach	Advantages	Disadvantages
1.	Def-Use Graph [15, 16, 17, 18, 19, 20]	• Provides data flow and control flow coverage • Represents the status of a variable	Not provides fault detection capabilities.
2.	Control Flow Graph (CFG)[21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32]	• Provides control flow and branch coverage. • Can be used to represents dominator relationships. • Can be used in automatic test case generation.	Cannot represent some features of object-oriented programs like polymorphic calls, method calls, inheritance and exception handling.
3.	Program Dependence Graph (PDG) [10, 11, 12, 33, 34]	• Represents both control and data dependence in a single graph.	Represents only a single procedure.
4.	ddgraph [35]	• Represents dominance and implication relations between the arcs. • Simple and flexible in construction of path coverage.	Used for path coverage in branch testing only.
5.	TestGraph [36]	• Considers node, arc and path coverage. • Provides a high level of reuse of testing information.	Cannot eliminate maintenance problem of inheritance.
6.	Class Graph [37, 38]	• Captures relations between classes. • Handles complicated relations between classifications and their syntax.	--
7.	Application Call Graph [39, 40]	• Resolves library callbacks more accurately. • Can be implemented with call-chain based and def-use pair based testing	Not used in white-box testing yet.
8.	Call-based Object-Oriented System Dependence Graph (COSDG) [9, 41]	• Captures object-oriented features like inheritance, polymorphism, etc. • Covers dependence, flow and call graph details. • Covers details of method visibility in a derived class.	Cannot represent exception handling code.

Chapter 3

Problem Statement

After reviewing the literature of Software Engineering Models, Testing techniques and various Graphical methods for representing the procedural calls, it has been analyzed that Flow Graph departs a very little information as compare to Dependence Graphs (like Program Dependence Graph, System Dependence Graph, *etc*).

Flow Graph represents the basic flow of control present in the program logic where as dependence graphs represent the data as well as control dependence relationship that can be present among various statements/methods of a program snippet.

Dependence Graphs can be compared against Flow Graph for various features like single/multiple procedures, statement/branch coverage, intra/inter procedure calls, flow/context sensitive and dependencies relationships, *etc*.

Chapter 4

Methodology

The proposed work addresses the comparison of flow graphs with dependence graphs on the basis of features like single/multiple procedures, statement/branch coverage, intra/inter procedure calls, flow/context sensitive, inheritance, polymorphism, dynamic binding features and dependencies relationships, *etc.*

For implementing the Comparative Analysis between Control Flow Graph (CFG), Program Dependence Graph (PDG) and System Dependence Graph (SDG), following steps have been followed:

1. Plug-in named "CFG Generator" has been used for generating Control Flow Graph (CFG) on Eclipse (version 3.4.2)GUI environment.

    1.1. Install plug-in named "CFG Generator" into Eclipse 3.4.2 environment.
    1.2. Import the .java file of the program under consideration.
    1.3. Click on respective method name in Outline Window. Choose option "CFG Generator" and then choose sub-option named "Build".
    1.4. In window named Flow Chart Viewer, the desired CFG will be generated.

2. Generation of Program Dependence Graph (PDG): This includes the generation of Control Dependence Graph as well as Data Dependence Graph and combining the above generated graph.

    2.1 Function named ProceduralDependenceGraphMatrix(), that resides in SystemDependenceGraph.jar,has been used that will give an Adjacency Matrix for the respective method under concern.
    2.2 Adjacency Matrix will provide a unique address for all the statements present in the program and assign a numeric value for Control and Data dependence relationships. (Numeric one will represent data dependence and numeric two will represent control dependence between the two nodes).

2.3 All the nodes that are related with data and control dependence relationship with a particular node will be enumerated in a distinct list.

2.4 For traversing through all the nodes listed in above step 2.3, a method named ControlDependenceBFSIterator() has been used. Methods named getname(), getancestornode() will be used for fetching the self-address of the node/statement as well as parent address of the particular node/statement.

2.5 After getting all the respective addresses of all parent nodes and its children, a consolidated graph can be constructed that will provide the merged view of Control Dependence and Data Dependence Graph, here control as well as data dependence will be shown.

3. Generating System Dependence Graph (SDG)

3.1 For getting list of methods present in a particular class, ConstructGraphClassAdapterhas been used (that take Class name as input).

3.2 For all the methods listed in above step 3.1, the control dependency can be computed at Class as well as System level.

Chapter 5

Experimental Results

This chapter focuses on a tool for generating Control Flow Graph (CFG) and implementation of System Dependence Graph API for generating System Dependence Graph (SDG).

5.1 Tool for generating Control Flow Graph

A Control Flow Graph (CFG) describes the sequence in which the instructions of a program will get executed. A CFG (or flow graph) G is defined as a finite set of nodes N and a finite set of edges E where an edge (i, j) in E connects two nodes n_i and n_j in N.

The Eclipse "CFG Generator" is a plug-in for the Eclipse IDE that generates Control Flow Graphs for Java code. It generates the graphs based on the evaluation of the source code. CFG Generator will generate the Control Flow Graph of the selected code snippet in Flow Graph Viewer window and also computes the MacCabe's complexity of the generated graph [71, 72]. Figure 5.1 is a simplified representation of the framework involved for Control Flow Generator. From the UML class diagram given in Fig 5.2, we can get a first idea about the most important classes involved in the process of CFG generation.

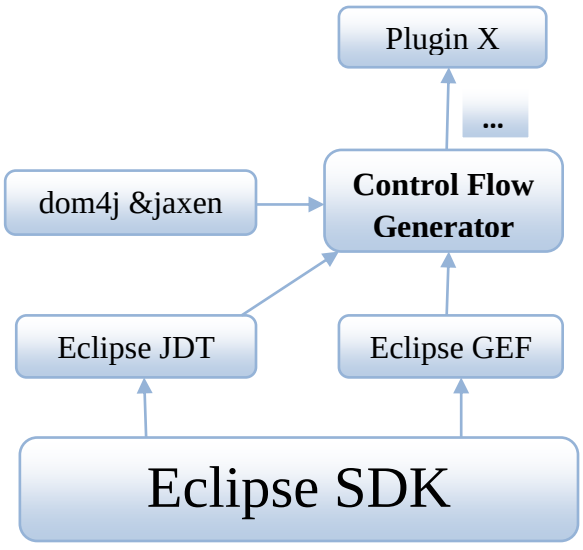


Fig 5.1 Framework/Schema for CFG Generator plug-in [72]

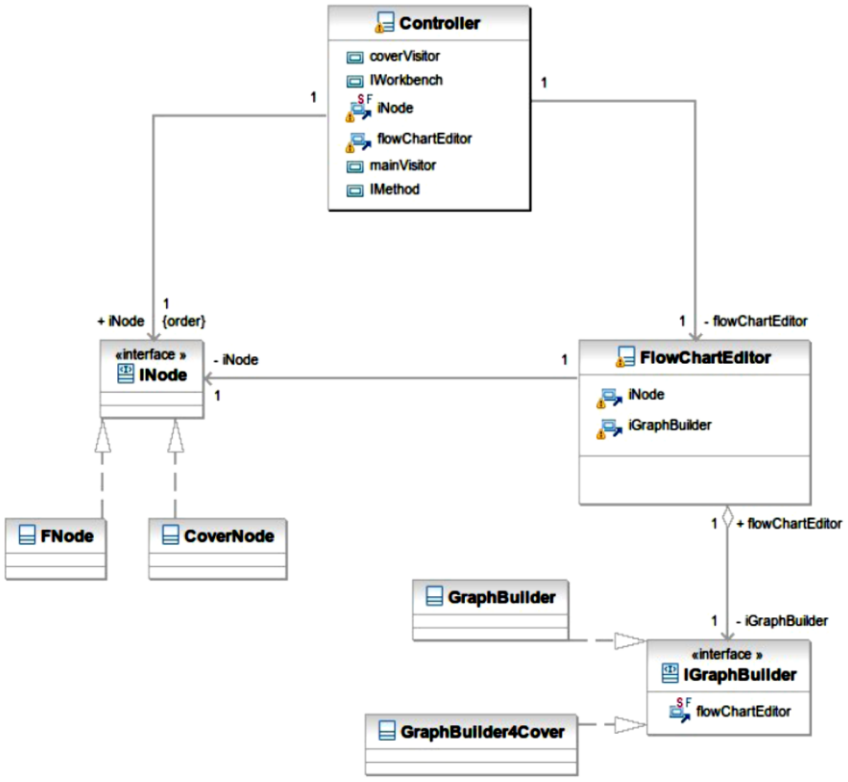


Fig 5.2 UML Class Diagram for CFG Generator [72]

5.2 System Dependence Graph API

"SystemDependenceGraph.jar" from Java System Dependence Graph API [70] has been used to generate System Dependence Graph. It has been developed by TONG Chun Yin under the supervision of Dr. LO Eric Chi Lik and Mr. LUK Ming Hay in Department of Computing in The Hong Kong Polytechnic University in 2009-2010. It takes java byte code as input and generates the System Dependence Graph.

5.3Implementation

5.3.1 Using CFG Generator

CFG Generator has been used to generate Control Flow Graph by using Eclipse 3.4.2. Consider the example program shown in Fig 5.3 that represents code for Demo class and stored under a package that also named as demo. Here, Demo class contains two methods namely add( ) and main( ). The method main( ) will compute the sum of integers from 1 to 10. The while loop will execute until the value of variable *i* is less than 11.

```
1.  package demo;
2.  public class Demo
3.  {
4.    public static int add(int i, int j)
5.    {
6.      int r = i + j;
7.      return r;
8.    }
9.    public static void main(String[] args)
10.  {
11.     int sum, i;
12.     sum = 0;
13.     sum = sum + 1;
14.     i = 1;
15.     while (i < 11) {
16.       sum = add(sum, i);
17.       i = add(i, 1);
18.       }
19.    System.out.println("sum = " + sum);
20.    System.out.println("i = " + i);
21.   }
22. }
```

Fig 5.3 Code for Demo Class in Java

After importing the above Demo class in Eclipse environment, the program will appear as shown in Fig 5.4 screen below. Here the screen has been divided into different frames.

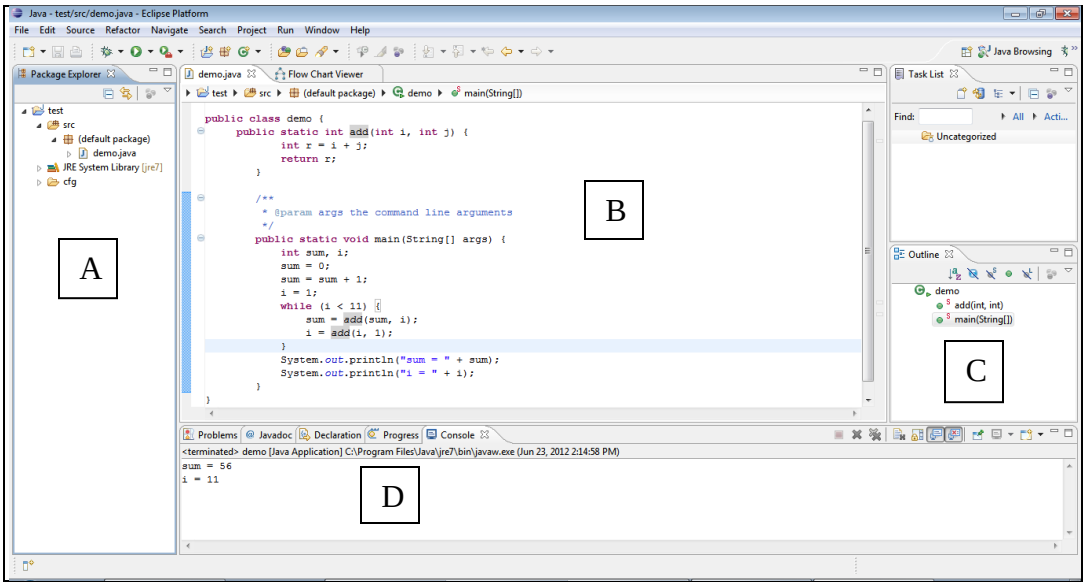


Fig 5.4 Demo Class program in Eclipse Environment

As shown in Fig 5.4, Frame A represents Project Explorer Window where the program will be shown in tree structure. The name of the package will be at the top and methods will be represented as its subparts. Frame B represents the working area where code is present. Frame C represents the Outline Window which will show the demo class methods in tree structure. Frame D represents the output of the Demo class program after compiling and execution of source code.

After successful compilation and execution, the next step is to generate Control Flow Graph (CFG) using CFG generator plug-in present in Eclipse. For using this plug-in, right click on the method name (whose CFG has to be generated) in the Outline Window (as shown in Frame C in Fig 5.5). After right click, a pop-up menu will appear (as shown in Frame E). This pop-up menu contains the option to use CFG Generator. On selection, the CFG Generator option displays a small menu that contain three options for CFG genearation. From these three options, the Build option will be selected to generate the CFG.

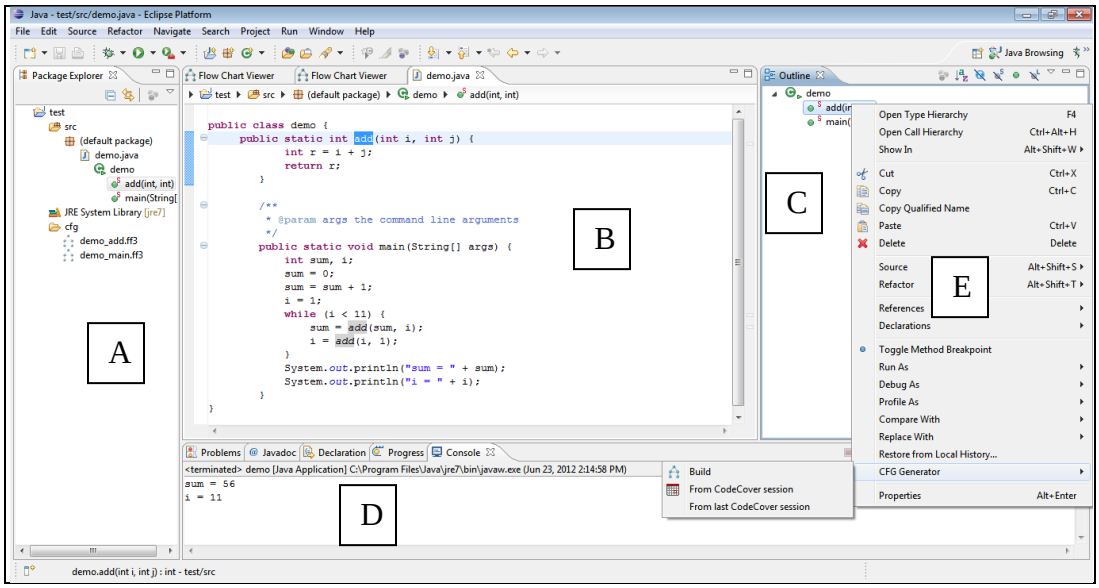


Fig 5.5 Steps to use CFG Generator plug-in

After Build option has been selected, it will generate the Control Flow Graph (CFG) of the selected method in Flow Chart Viewer Window as shown in Fig 5.6. Frame F represents the Flow Chart Viewer window that contains the Control Flow Graph(CFG) of method named add( ) in Demo class program.

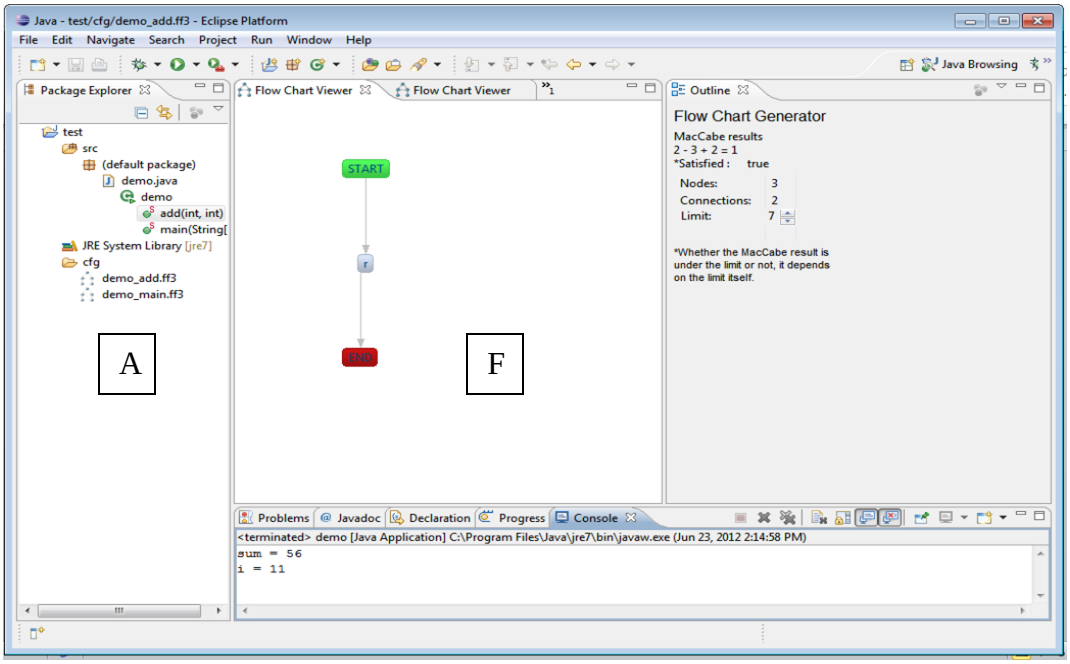


Fig 5.6 Control Flow Graph for add( ) method

Fig 5.7 represents the Control Flow Graph (CFG) for the method named main( ) in Demo class program. Frame F represents the CFG for the main( ) method in the Flow Chart Viewer window.

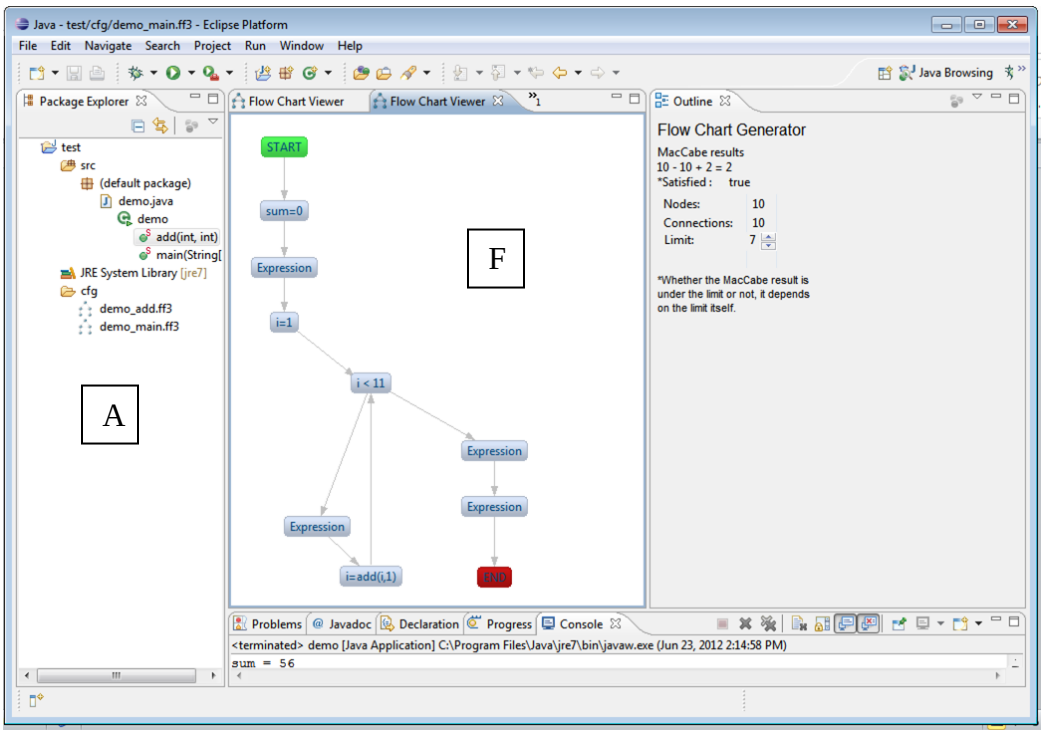


Fig 5.7 Control Flow Graph for main( ) method

5.3.2 Using SDG API

A Prototype using System Dependence Graph API has been customized in the Java language to generate System Dependence Graph.

Program Dependence Graph (PDG) represents both control dependence and data dependence in a single graph. In other words, PDG is a combination of Control Dependence Graph and Data Dependence Graph. It consists of vertices and edges where vertices represent the program statements and edges represent the data dependency and control dependency.

Fig 5.8 represents the code for Demo Class program in NetBeans environment. Frame A represents the working area where Demo class program has been written and Frame B represents the Properties Window that displays the properties of the class.

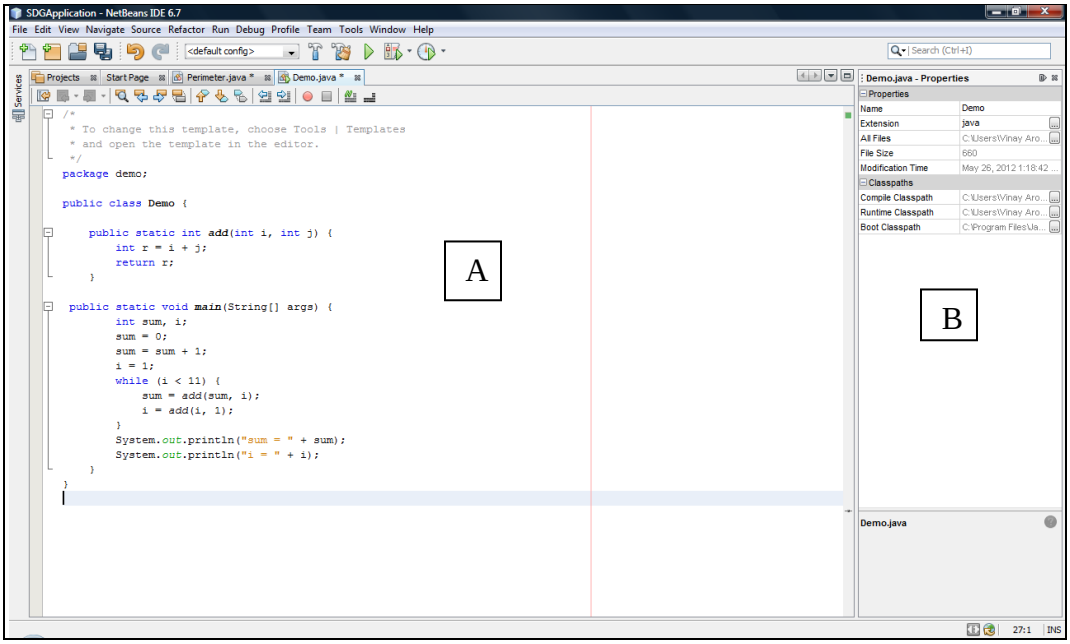


Fig 5.8 Demo Class Program in NetBeans Environment

Fig 5.9 represents the Control Dependence Graph of Demo Class program. Frame A represents the statements of the Demo Class program along with line numbers. Frame B represents the Control Dependence Graph of the program. The nodes are represented in circular form and edges are represented by blue colored regular lines.

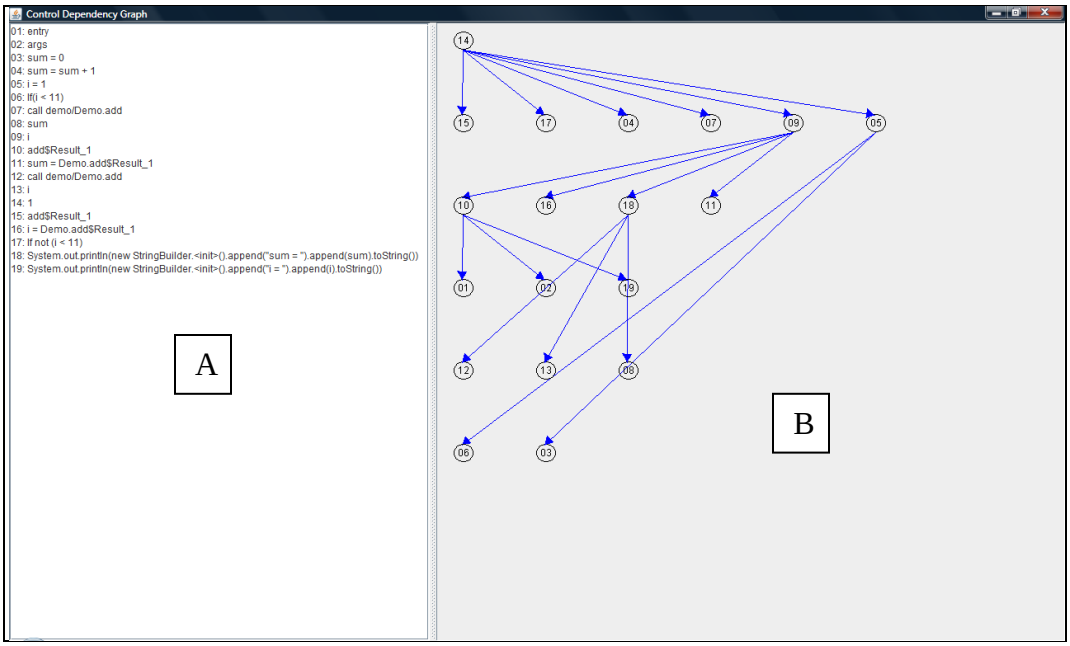


Fig 5.9 Control Dependence Graph for the Demo Class

Fig 5.10 represents the Control Dependence Graph for the method named add( ) in Demo Class program. Frame A represents the program in tree structure. The name of the package will be at the top and methods are represented as its sub parts. Frame B represents the code snippet for method add( ) along with its line numbers. Frame C represents its corresponding Control Dependence Graph.

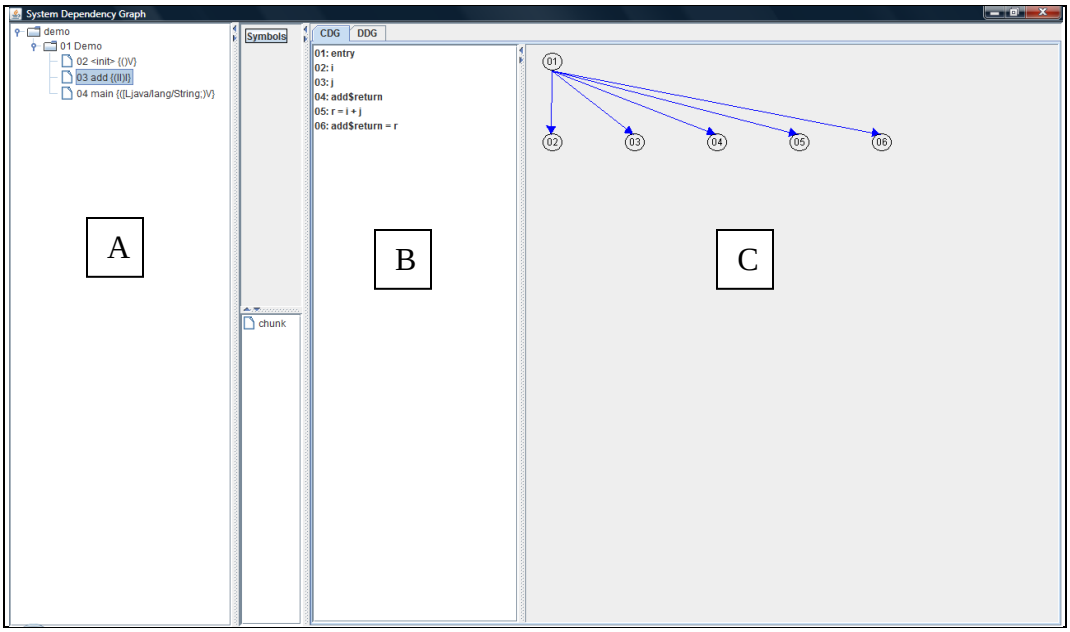


Fig 5.10 Control Dependence Graph for add( ) method

Fig 5.11 represents Control Dependence Graph for the main( ) method in Demo class Program.

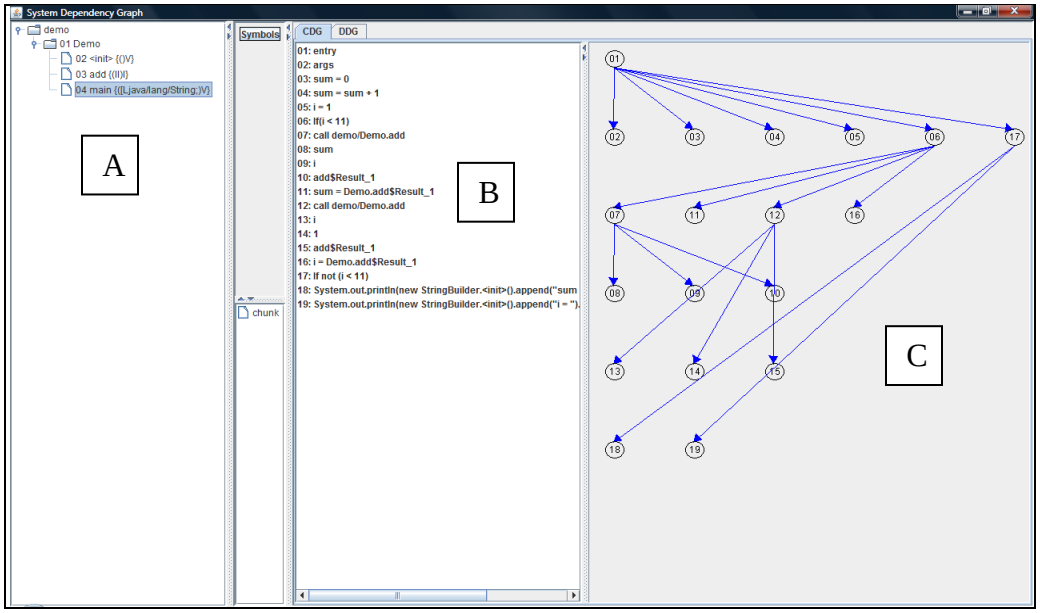


Fig 5.11 Control Dependence Graph for main( ) method

Fig 5.12 represents the Data Dependence Graph of Demo Class program. Frame A represents the statements of the Demo Class program along with line numbers. Frame B represents the Data Dependence Graph of the program. The nodes are represented in circular form and have also the edges are represented by blue colored regular lines.

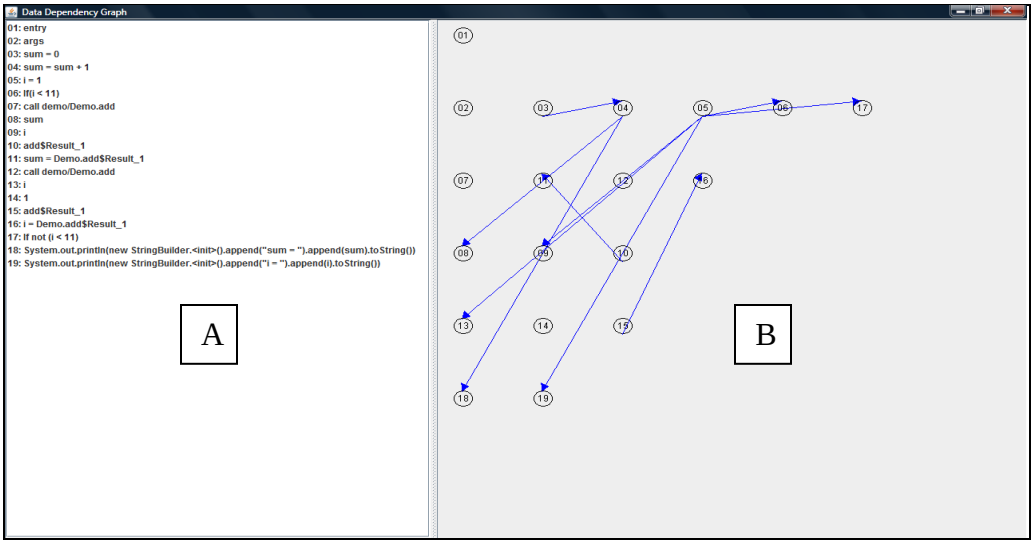


Fig 5.12 Data Dependence Graph for the Demo Class

Fig 5.13 represents the Data Dependence Graph for the method named add( ) in Demo Class program. Frame B represents the code snippet for method add( ) along with its line numbers. Frame C represents its corresponding Data Dependence Graph.

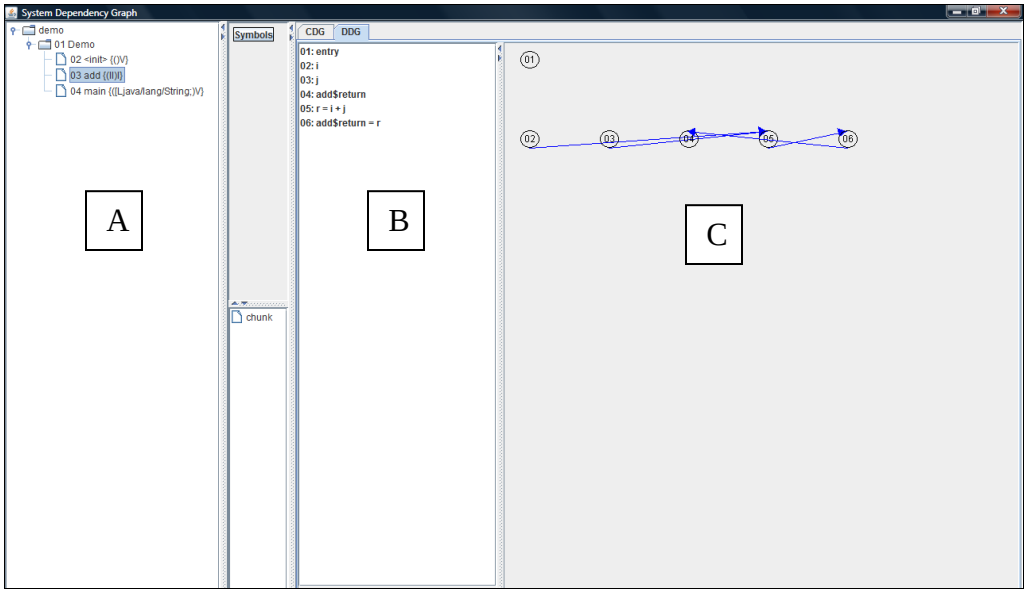


Fig 5.13 Data Dependence Graph for add( ) method

Fig 5.14 represents Data Dependence Graph for the main( ) method.

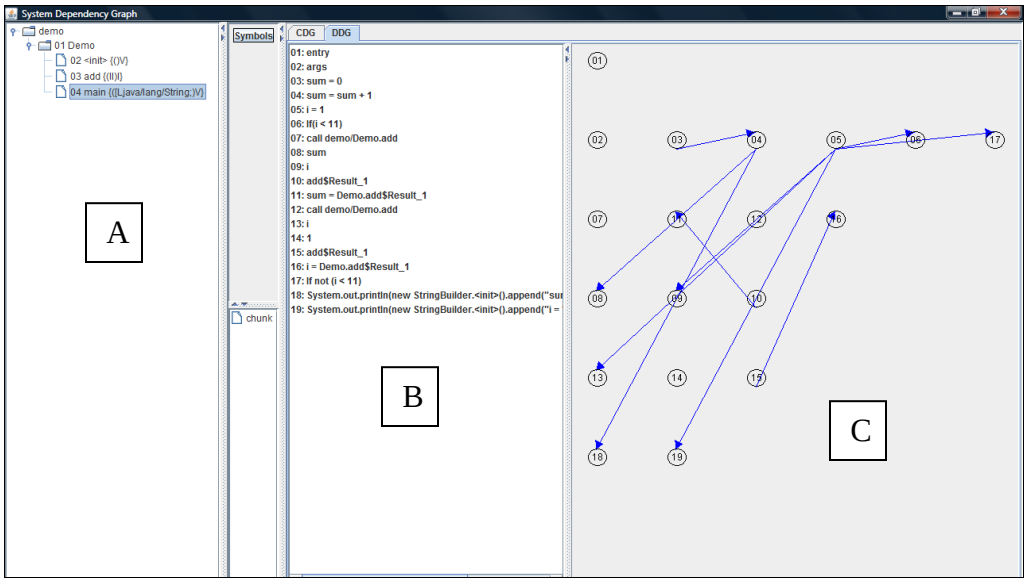


Fig 5.14 Data Dependence Graph for main( ) method

Fig 5.15 represents the Program Dependence Graph (PDG) for the Demo Class program. Frame A represents the statements of the program along with its line numbers. Frame B represents the PDG of the program. The nodes are represented as rectangular blocks and blue colored edges are data dependence edges and red colored edges represent control dependency.

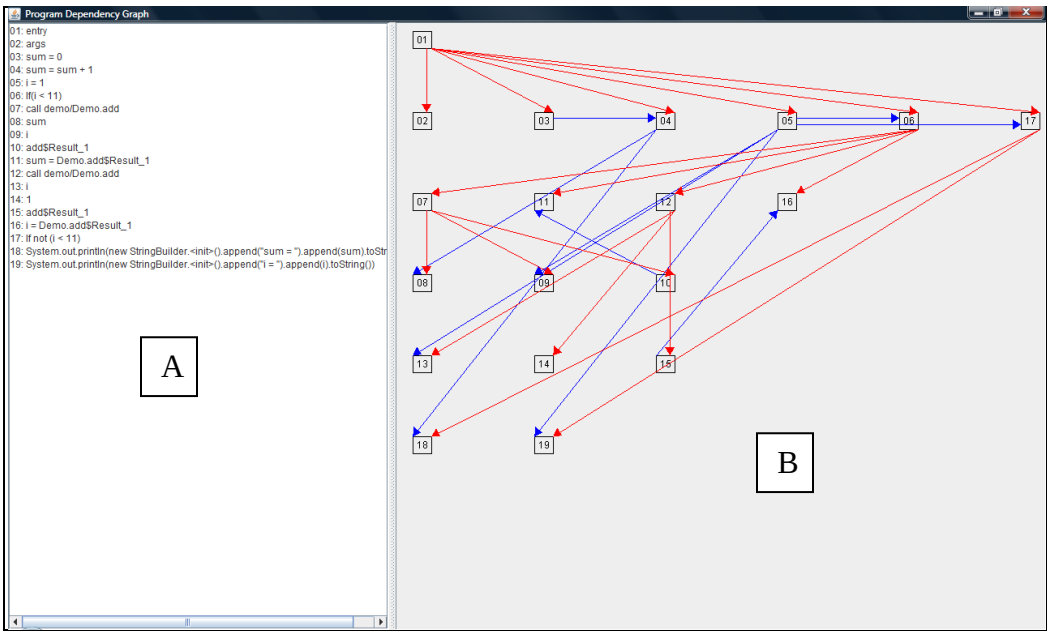


Fig 5.15 Program Dependence Graph for the Demo Class

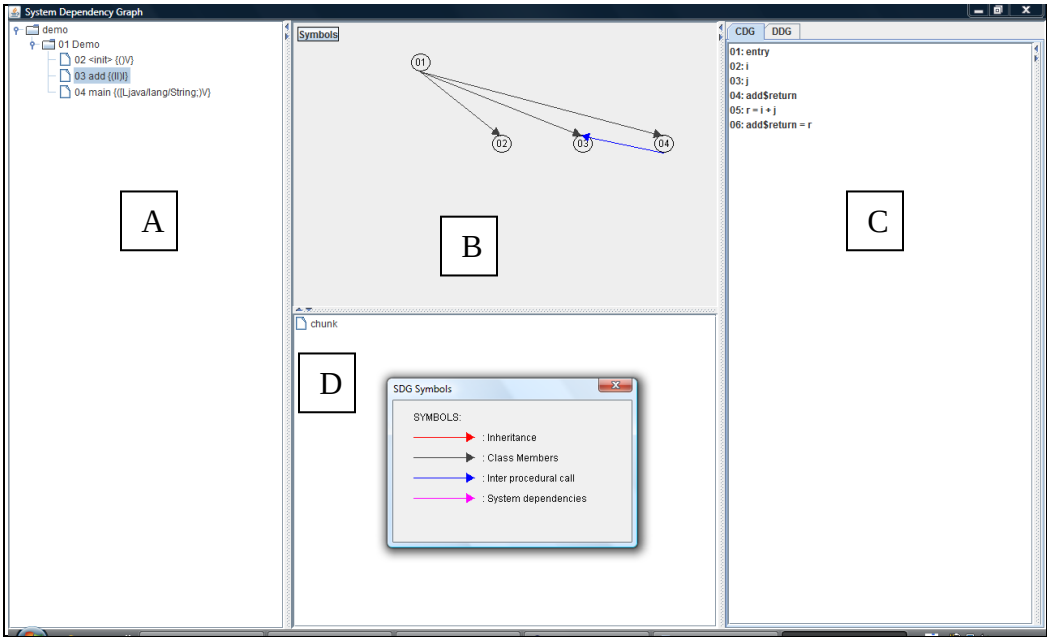


Fig 5.16 System Dependence Graph for the Demo Class

Fig 5.16 represents the System Dependence Graph (SDG) for the Demo class program. Frame A represents the program in tree structure. Frame B represents its corresponding SDG and Frame C represents the code for the selected method. Frame D represents the various symbols used in SDG to represent method calls. Red colored edges represent inheritance, gray color edges represent class members, blue color represents inter-procedural calls and pink color shows system dependencies.

Chapter 6

Conclusion and Future Scope

6.1 Conclusion

From the given literature survey and experimental setup, it has been concluded that each and every graphical method given above supports some unique features/parameters which helps in depicting various relationships/dependencies. On the basis on some parameters, the comparative analysis of Control Flow Graph (CFG), Program Dependence Graph (PDG) and System Dependence Graph (SDG) has been done to better understand the usage of these graphs.

Table 6.1 Comparison of CFG, PDG and SDG

S No.	Parameters	CFG	PDG	SDG
1.	Control Dependency [2,7, 9,10,21,22,34]	✓	✓	✓
2.	Data Dependency [9,10,13,34]	×	✓	✓
3.	Transitive Dependency [11, 12, 13, 62]	×	×	✓
4.	Single Procedure [7,10,11,12,21,33]	✓	✓	✓
5.	Multiple Procedures [10,11,21,33,63]	×	×	✓
6.	Intra-procedural Calls [10,11,12,34,62]	×	✓	✓
7.	Inter-procedural Calls [10,11,12,34,62,63,64]	×	×	✓
8.	Multiple types of Edges [7,10,23, 34,63]	×	✓	✓
9.	Slicing [10, 11, 33, 34, 62, 63]	×	✓	✓
10.	Flow-Sensitive [7,63]	✓	✓	×
11.	Context-Sensitive [7,63]	×	×	✓
12.	Inheritance & Polymorphism [9,10,12,27,64]	×	×	✓
13.	Dynamic Binding [12,13,33,34,62,64]	×	×	×
14.	Test Case Generation [10,11,21,22,34]	✓	✓	✓
15.	Exception Handling [27, 63]	×	✓	×
16.	Parameter Passing [9,12,41, 62, 67]	×	×	✓

Control Flow Graph (CFG) can be taken as base graph for various other representations like BDG, EDG, EG, CCFG, *etc*. Similarly, Program Dependence Graph (PDG) can be taken as base graph for various other graphs such as ClDG, OPDG, DOPDG, *etc*. System Dependence Graph extends the features of PDG and can be taken as base graph for other graphs like ESDG, JSDG, COSDG, *etc*. Fig 6.1 represents a hierarchical tree structure of CFG, PDG, SDG and their sub-graphs.

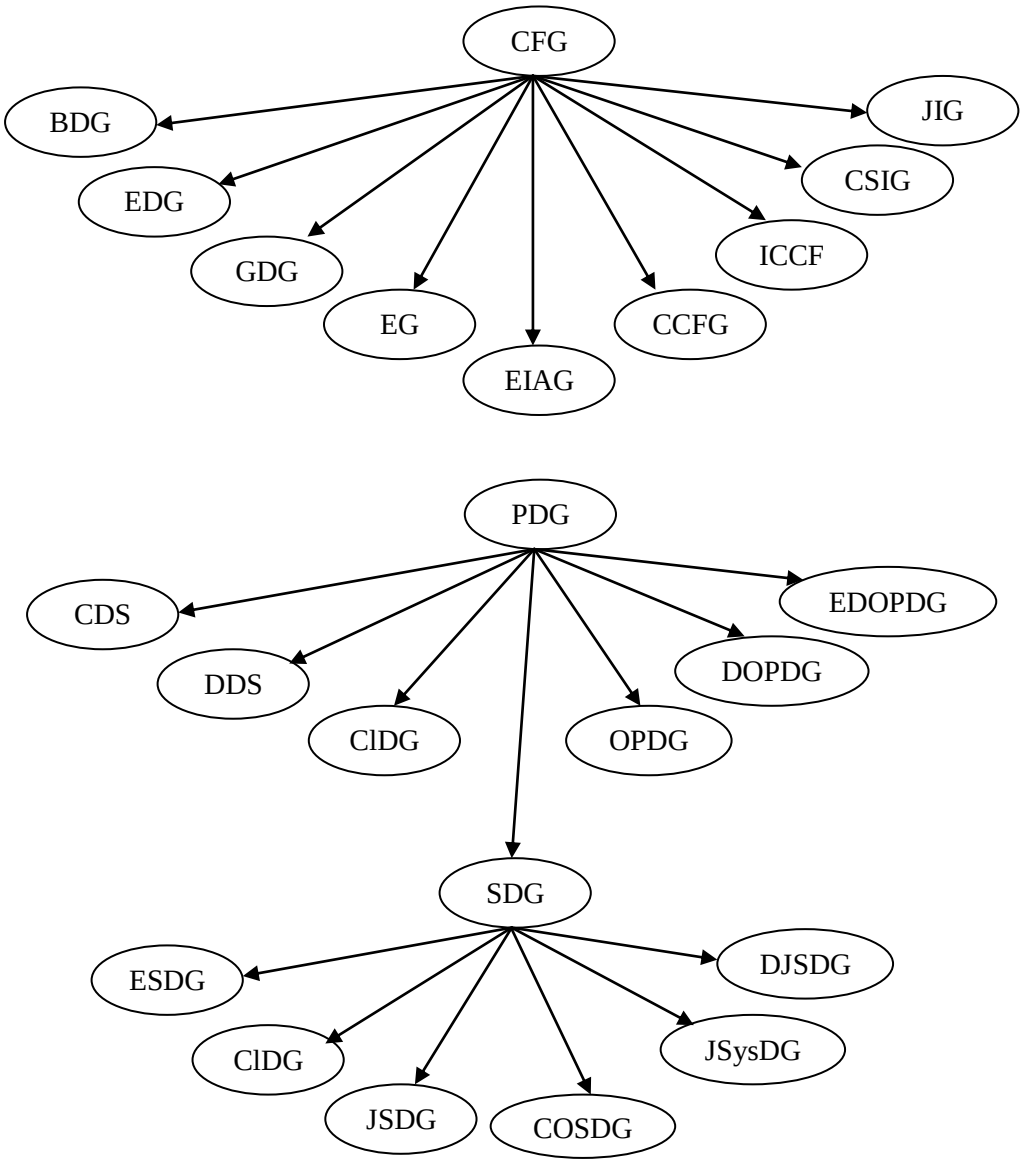


Fig 6.1 CFG, PDG, SDG and their sub-graphs

Table 6.2 Description of Graphs shown in Fig 6.1

S No.	Acronym	Abbreviation	Description
1.	CFG	Control Flow Graph	Flow between nodes & edges
2.	BDG	Block Dominator Graph	Dominator relationship among blocks
3.	EDG	Edge Dominator Graph	Dominator relationship among edges
4.	GDG	Global Dominator Graph	BDG + EDG + Inter-procedural level
5.	EG	Event Graph	CFG + Interaction between procedures
6.	EIAG	Event InterActions Graph	EG+Interaction for concurrent programs
7.	CCFG	Class Control Flow Graph	CFG + Call Graph between classes
8.	ICCFG	Inter-Class Control Flow Graph	CFG + Inter-class relationship
9.	CSIG	Class Specification Implementation Graph	CFG for each class method
10.	JIG	Java Inter-class Graph	Inter relationship of AspectJ programs
11.	PDG	Program Dependence Graph	Control +data dependencies for single procedure
12.	CDS	Control Dependence Sub-graph	Control dependencies for single procedure
13,	DDS	Control Dependence Sub-graph	Data dependencies for single procedure
14.	OPDG	Object Program Dependence Graph	PDG+ Object-Oriented Features
15.	DOPDG	Dynamic Object Program Dependence Graph	OPDG + Dynamic Slicing
16.	EDOPDG	Efficient Dynamic Object Program Dependence Graph	DOPDG+ few modifications
17.	ClDG	Class Dependence Graph	Set of PDGs + Inter-procedural calls within class
18.	SDG	System Dependence Graph	Set of PDGs + Interprocedural calls for whole system
19.	ESDG	Extended System Dependence Graph	Extends SDG by representing a class with ClDG
20.	JSDG	Java System Dependence Graph	SDG + Interfaces & Packages in Java
21.	JSysDG	Java System Dependence Graph	JSDG_ few modifications
22.	DJSDG	Dynamic Java System Dependence Graph	JSDG + Dynamic Slicing
23.	COSDG	Call-based Object-oriented System Dependence Graph	Dependencies + Flow + Call Graph+ Inherited Call +Polymorphic calls

The major contribution of this work is given below:

- The key contribution of work done is the comparative analysis of various graphical techniques for representing the system that is having Method Calls.
- The foundation of the proposed technique is System Dependence Graph (SDG). With this representation, developer can better understand the method calls within the program.

6.2 Future Work

This proposed work has focused on Control Flow Graph (CFG), Program Dependence Graph (PDG) and System Dependence Graph (SDG) for comparative analysis.

There are the following points that can be explored further:

- Area using System Dependence Graph for dealing exception handling in Java can be explored further.
- Flow sensitivity in procedural as well as Object-Oriented program can be explored using System Dependence Graph.

References

[1] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams*", In Future of Software Engineering, pp. 85-103,* 2007.

[2] R. Mall, Fundamentals of Software Engineering, Prentice Hall of India.

[3] M. D. Smith and D. J. Robson, "A Framework for Testing Object-Oriented Programs*", Journal of Object-Oriented Programming, pp. 45-53,* June 1992.

[4] K. Tai, F. J. Daniels, "Test Order for Inter-Class Integration Testing of Object-Oriented Software", IEEE, 1997.

[5] G. Suganya, S. Neduncheliyan, "A Study of Object Oriented Testing Techniques: Survey and Challenges", IEEE.

[6] M. E. Paige, "On partitioning Program Graphs", *IEEE Transactions on Software Engineering, vol. se-3, no. 6, pp. 386-393,* November 1977.

[7] Robert Gold, "Control Flow Graph and Code Coverage", *Int. J. Appl. Math. Computer Sci., vol. 20, no. 4, pp. 739–749,*2010.

[8] A. P. Mathur, Foundations of Software Testing, Pearson, 2011.

[9] E S F Najumudheen, R. Mall, D. Samanta, "A Dependence representation for coverage testing of object-oriented programs", *Journal of Object Oriented Technology (JOT), Vol. 9, No. 4, pp. 1-23,* 2010.

[10] J. Ferrante, J. Worren and K. Ottenstein, "The Program Dependence Graph and its use in optimization", *In ACM Transactions on Programming Languages and Systems, vol. 9, no. 3, pp. 319-349,* July 1987.

[11] S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs," *In ACM Transactions on Programming Languages and Systems, vol. 12, pp. 26-60,* January 1990.

[12] L. Larsen and M. J. Harrold, "Slicing object-oriented software", In *18th International Conference on Software Engineering,* pp. 495–505, Mar. 1996.

[13] D. Liang and M. J. Harrold, "Slicing objects using System Dependence Graphs", *International Conference on Software Maintenance, pp. 358-367,* November 1998.

[14] R. Ferguson and B. Korel, "The chaining approach for software test data generation", *ACM Transactions on Software Engineering and Methodology, vol. 5, no. 1, pp. 63-86,* January 1996.

[15] L. D. Fosdick and L. J. Osterwall, "Data Flow Analysis in Software Reliability", *InACM Computing Surveys, vol. 8,* September 1976.

[16] *S* .Rapps and E. J. Weyuker, "Data flow analysis techniques for test data selection," *In Proc. 6th Int. Con$ Software Engineering, IEEE Computer Society, Tokyo, Japan, pp. 272-277,* September 1982.

[17] *S.* Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Eng., vol. SE-I 1, no. 4, pp. 367-375,* April 1985.

[18] L. A. Clarke, A. Podgurski, D. J. Richardson and S. J. Zeil, "A formal evaluation of data flow path selection criteria",*IEEE Transactions on Software Engineering, vol. 15, no. 11,* November 1989.

[19] A. L. Souter and L. L. Pollock, "The construction of contextual def-use associations for object oriented systems", *IEEE Transactions on Software Engineering, vol. 29, no. 11, pp. 1005-1018,* 2003.

[20] A. M. R. Vincenzi, J. C. Maldonaldo, W. E. Wong and M. E. Delamaro, "Coverage testing of Java programs and components", *In Science of Computer Programming,* 2005.

[21] Frances E. Allen, "Control Flow Analysis", *In Proceedings of a Symposium on Compiler optimization, ACM SIGPLAN Notices, vol. 5*, July 1970.

[22] P. G. Frankl and E. J. Weyuker, "An Applicable Family of Data Flow Testing Criteria", *IEEE Transactions on Software Engineering, vol. 14, no. 10*, October 1988.

[23] HiralalAgrawal, "Dominators, Super Blocks and Program Coverage", *In Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming language,* 1994.

[24] H. Agrawal, "Efficient Coverage testing using Global Dominator Graphs", *In Proceedings of the ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, Toulouse, France, vol. 24, no. 5, pp. 11-20,* September 1999.

[25] T. Katayama, Z. Furukawa, K. Ushijima, "A Method for Structural Testing of Ada Concurrent Programs Using the Event Interactions Graph", *In Proceedings of Asia-Pacific Software Engineering Conference, pp. 355-364,* 1996.

[26] J. Edvardsson, "A survey on automatic test data generation", *In Proceedings of the 2nd Conference on Computer Science and Engineering, pp. 21-28,* October 1999.

[27] V. Martena, A. Orso and M. Pezze, "Interclass Testing of object oriented software", *In Proceedings of 8th IEEE International Conference on Engineering of Complex Computer Systems, pp. 135-144,* 2002.

[28] S. Beydeda, V. Gruhn, "Class specification implementation graphs and their application in regression testing", *In Proceedings of the 26th Annual International Computer Software and Applications Conference, pp. 835-840,* 2002.

[29] S. Beydeda, V. Gruhn, "Class specification implementation graphs for integrated black and white box testing", *Software Engineering and Applications,* 2002.

[30] G. Xu and A. Rountev, "Regression Test Selection for AspectJ Software", *In 29th International Conference on Software Engineering, pp. 65-74,* 2007.

[31] J. J. Li, D. M. Wesis and H. Yee, "An automatically generated runtime instrumenter to reduce coverage testing overhead", *In Proceedings of the 3rd international workshop on Automation of software test, ACM Transactions*, May 2008.

[32] P. Mouy, B. Marre, N. Williams, P. L. Gall, "Generation of all-paths unit test with function calls", *In 1st International Conference on Software Testing, Verification, and Validation, pp. 32-41, IEEE,* 2008.

[33] G. Rothermel and M. J. Harrold, "Selecting regression tests for Object-Oriented Software", In Proceedings of International Conference on Software Maintenance, *pp. 14-25, IEEE transactions,* 1994.

[34] K. Tewary and M. J. Harrold, "Fault Modeling using the Program Dependence Graph", *In Proceedings of 5th International Symposium on Software Reliability Engineering, pp. 126-135, IEEE Transactions,* 1994.

[35] A. Bertolino and M. Marre, "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs", *IEEE Transactions on Software Engineering, vol. 20, no. 12, pp. 885 – 899,* December 1994.

[36] J. McDonald and P. Strooper, "Testing Hierarchies in the ClassBench Framework", *In Proceedings of Technology of Object-Oriented Languages, pp. 229-240,* 1998.

[37] *Huo Yan Chen and Jian-Zhu Lu, "*The Use of Class Graph to Analyze the Effectiveness ofan Approach for Object-Oriented Class-Level Testing", IEEE Transactions, 2000.

[38] K. R. P. H. Leung, W. Wong, "Towards a more efficient way of generating test cases: Class Graphs", *In Proceedings of 1st Asia-Pacific Conference on Quality Software, pp. 285-293,* 2000.

[39] B. Ryder, "Constructing the Call graph of a program", *IEEE Transactions on Software Engineering, vol. se-5, no. 3,* May 1979.

[40] W. Zhang and B. Ryder, "Constructing accurate application call graphs for java to model library callbacks", *In Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, pp. 63-74,* 2006.

[41] E S F Najumudheen, R. Mall, D. Samanta, "A Dependence Graph-based Test Coverage Analysis technique for Object-Oriented programs", In *6th International Conference on Information Technology: New Generations,* IEEE, 2009.

[42] P. G. Frankl, "ASSET User Manual"**,** *Dep. Computer Sci. Courant Inst. Math. Sci., New York Univ., New York, Tech. Rep. #318,* September 1987.

[43] M. J. Balcer, W. Hasling, and T.J. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications", *In Proc. of the 3rd Symposium on Testing, Analysis, and Verification, ACM Press, New York, pp. 210-218,* 1989.

[44] T. J. Ostrand and M. J. Balcer, "The Category-Partition Method for Specifying and Generating Functional Tests", *In Communications of the ACM, pp. 676-686,* June 1988.

[45] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, "Experiments on the Effectiveness of Dataflow and Control flow-Based Test Adequacy Criteria", *IEEE Transactions,* 1994.

[46] H. Agrawal, R. A. DeMillo, and E. H. Spafford, "Debugging with dynamic slicing and backtracking", *Software Practice and Experience, vol. 23, no. 6, pp. 589-616,* June 1993.

[47] "ANTLR: ANother Tool for Language Recognition", http://www.antlr.org/ (accessed 16/02/2012).

[48] T. Katayama, T. Komoda, Z. Furukawa and K. Ushijima, "Definition of the Test-case for Concurrent Programs and Prototype of Test-case Generation System," *In Tvans. IPS Japan, vo1. 34, no. 11, pp. 2223-2232,* 1993.

[49] A. Binns, G. McGraw, "Building a Java software engineering tool for testing applets", *In IntraNet Conference, New York, USA,* 1996.

[50] Reliable Software Technologies Corp., "White paper: PiSCES Coverage Tracker Technical Brief", A web document at URL http://www.rstcorp.com/techbrf.html, 1995 (accessed 16/02/2012).

[51] MAN MACHINE SYSTEMS, "JCover 2.1—user's guide", WEB page, Available at: http://www.mmsindia.com/JCover.html, 2002 (accessed 16/02/2012).

[52] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. L. London, W. E. Wong, S. Ghosh and N. Wilde, "Mining System tests to aid software maintenance", *IEEE Computer, vol. 31, no. 7, pp. 64-73,* July 1998.

[53] S. Research, "User's guide—TCAT for Java/Windows—version 1.2", Página WEB, Available at: http://www.soft.com/, 1999 (accessed 16/02/2012).

[54] A. L. Souter, T. M. Wong, S. A. Shindo and L. L. Pollock, " TATOO: Testing and Analysis Tool for Object-oriented Software", *In Proc. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, April 2001.

[55] P. Corporation, "Automatic Java software and component testing: using JTest to automate unit testing and coding standard enforcement", WEB page, Available at: http://www.parasoft.com/, 2002 (accessed 16/02/2012).

[56] P. Corporation, "Using design by contract to automate Java software and component testing", WEB page, Available at: http://www.parasoft.com/, 2002 (accessed 16/02/2012).

[57] K. Beck, E. Gamma, "JUnit cookbook", WEB page, Available at: http://www.junit.org/, 2002 (accessed 16/02/2012).

[58] Ashley J.S Mills, "JUnit Testing Utility Tutorial", The University Of Birmingham, 2005.

[59] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, J. C. Maldonaldo, "JaBUTi: A coverage analysis tool for Java programs", *In 17th Brazilian Symposium on Software Engineering, Manaus, AM, Brazil, pp. 79-84*, 2003.

[60] N. Williams, B. Marre, P. Mouy, and M. Roger, "PathCrawler: Automatic generation of path tests by combining static and dynamic analysis," *In Proc. EDCC, pp. 281–292*, 2005.

[61] "Dependence Graph", Available at: http://threadingbuildingblocks.org/docs/help/hh_goto.htm#reference/flow_graph/dependency_flow_graph_example.htm (accessed 10/5/2012).

[62] P. E. Livadas and S. Croll, "System Dependence Graphs Based on Parse Treesand their Use in Software Maintenance", *IEEE Transactions*, 2005.

[63] C. Hammer, J. Krinkle and G. Snelting, "Information Flow Control for Java based on Path Conditions in Dependence Graphs", *In Proceedings of IEEE International Symposium on Secure Software Engineering, Virgina, USA*, 2007.

[64] B. Xu, Z. Chen and H. Yang, "Dynamic slicing object-oriented programs for debugging", In the Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation, pp. 115 – 122, 2002.

[65] S. Park, "Efficient Dynamic Slicing of Object Oriented Programs", Dept. of R&D, Korea Micro System, pp.143-721, January2003.

[66] D. P. Mohapatra, R. Mall, and R. Kumar, "A node marking dynamic slicing technique for object-oriented programs", In Proceedings of Workshop on Software Development and Architecture, Bangalore, pp.1 – 15, January 2004.

[67] J. Zhao, "Applying program dependence analysis toJava software," in *Proc. Workshop on Software Engineering and Database Systems*, (Taiwan), pp. 162–169, December 1998.

[68] Neil Walkinshaw, Marc Roper, Murray Wood, "The Java System Dependence Graph", Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003.

[69] Liu Xi, Miao Li, Zhao Dan, Li Wei, "An approach of coarse-grained dynamic slices for Java programs", IEEE 3rd International Conference on Communication Software and Networks, pp.670 – 674, May 2011.

[70] TONG Chun Yin under the supervision of Dr. LO Eric Chi Lik and Mr. LUK Ming Hay , " Java System Dependence graph API ", *Department of computing, The Hong Kong polytechnic University*, 2010, Available at: http://www.comp.polyu.edu.hk/~csllo/teaching/SDGAPI (accessed 25/4/2012).

[71] "Eclipse," Available at: http://www.eclipse.org/documentation/(accessed 5/5/2012).

[72] A. Alimucaj, "Control Flow Graph Generator Documentation", University of Applied Sciences, Version 1.0, June 2009.

List of Publications

Published:

[1] S. Verma, V. Arora, "Survey on Graphical Methods for Test Case Generation", IJMAN of International Forum of Researchers Students and Academician, Issue No. 2, Vol. No. 2, May 2012.