Technique for Debugging Incomplete Constructor Anomaly in Object-Oriented System

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

Master of Engineering
in
Software Engineering

Submitted By
Mehak Jindal
(Roll No. 801031029)

Under the supervision of:
Mr. Vinay Arora
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2012

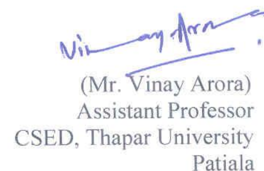
Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Technique for Debugging Incomplete Constructor Anomaly in Object- Oriented System*", 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.

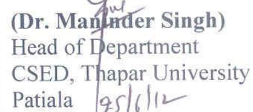
(Mehak Jindal)

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

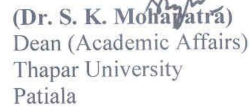


(Mr. Vinay Arora)
Assistant Professor
CSED, Thapar University
Patiala

Countersigned by



(Dr. Maninder Singh)
Head of Department
CSED, Thapar University
Patiala



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

i

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. Vinay Arora, 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 express my heartfelt thanks to my parents, my brother and my well-wishers for their co-operation, which they were always ready to extend.

At the end, I would like to thank my friends especially Shveta Verma for her support. I want to express my appreciation to every person who contributed with either inspirational or actual work to this thesis.

Mehak Jindal

Abstract

The Object-oriented (OO) programming is an evolutionary approach to software engineering which encompasses the entire software life cycle. With the use of object-oriented approach, programming becomes modularize but at the same time testing becomes complex and program will be difficult to debug, which in turn results in a huge loss of resources. To overcome the difficulty of debugging, a technique named program slicing has been introduced. It is a technique to extract the desired parts of program using some slicing criterion. It has been introduced to aid the developers in the area of debugging and program comprehension by reducing the complexity of the program.

There are many dependence graphs like Control Dependence Graph, Data Dependence Graph, Program Dependence Graph, System Dependence Graph, *etc.* that can be used for program representation in the process of program slicing. There are many faults like State definition anomaly, Anomalous behavior construction, Incomplete constructor, *etc.* that are related to Inheritance and Polymorphism features of object-oriented system. The process of debugging the faults present in the Object-Oriented code can be made approachable by the use of Program Slicing with dependence graphs.

Table of Contents

Certificate………………………………………………………………………. i

Acknowledgement………………………………………………………………... ii

Abstract………………………………………………………………………… iii

Tables of Contents…………………………………………………………….. iv

List of Figure…………………………………………………………………... vi

List of Tables…………………………………………………………………... viii

Chapter 1 Introduction………………………………………………………... 1

1.1 Object-oriented features…………………………………………………... 1

1.1.1 Object and classes………………………………………………… 2

1.1.2 Inheritance……………………………….................................. 2

1.1.3 Polymorphism……………………………………………………….. 4

1.2 Graphical Representation of Program………………………………… 4

1.2.1 System Dependence Graph……………………………………...... 6

1.3 Inconsistencies in programming language…………………………….. 8

1.3.1 Problem in inheritance and polymorphism……………………….. 9

1.3.2 Anomaly related to constructor Methods……………………….... 9

1.4 Program Slicing……………………………………………………….... 11

1.4.1 Slicing Criterion……………………………………………………... 12

1.4.2 Classification of Program Slicing………………………………… 12

1.4.3 Type of slicing……………………………………………………. 12

1.4.4 Levels of slicing……………………………………………………... 14

1.4.5 Direction of slicing…………………………………………………... 15

1.5 Organization of Thesis…………………………………………………… 15

Chapter 2 Literature survey………………………………………………….. 16

2.1 Dynamic Slicing………………………………………………………..... 16

2.2 Dynamic slicing of Object-Oriented Programs……………………….... 16

2.2.1 Using Dynamic Dependence Graph…………………………….. 16

2.2.2 Using Forward Approach……………………………………….. 17

2.2.3 Using Object Programs Dependence Graph (OPDG)……….…... 18
2.2.4 Using Byte code traces……………………………………….. 21
2.2.5 Using System Dependence Graph (SDG)………………………. 22
2.2.6 Using Dynamic Catch Approach……………………………….. 25
2.2.7 Using Dynamic Impact Analysis Approach…………………….. 25
2.2.8 Using Control Dependence Graph (CDG)……………………… 25
2.3 Tools of Program Slicing……………………………………………….. 25
2.4 Comparison of Dynamic Slicing of Object-Oriented Approaches……... 26
2.5 Anomalies due to Inheritance and Polymorphism……………………… 27
Chapter 3 Problem Statement………………………………………………… 28
3.1 Gap analysis in Existing Work………………………………………….. 28
3.2 Problem Statement……………………………………………………..... 28
Chapter 4 Methodology…………………………………………………….. 30
Chapter 5 Experimental Results…………………………………………… 32
5.1 Implementation………………………………………………………… 32
Chapter 6 Conclusion and Future Work…………………………………….. 38
6.1 Conclusion……………………………………….......................... 38
6.2 Future Work…………………………………………………………... 38
References………………………………………………………………… 39
List of Publications……………………………………………………….. 43

List of Figures

Fig. No.	Figure Description	Page No.
1.1	Features of Object-Oriented Language	2
1.2	A sample Java code for single inheritance and its graphical representation	3
1.3	A sample Java code for multilevel inheritance and its graphical representation	3
1.4	Hierarchical structure of polymorphism	4
1.5	Dependence Graph Representation	5
1.6	Relationship between Dependence Graphs	6
1.7	(a) A sample dependence graph (b) System dependence graph of sample program [7]	7
1.8	(ACB1) Anomalous Construction behavior [8]	9
1.9	Incomplete Construction of state variable fd [8]	11
1.10	Classification of Program Slicing	12
1.11	A sample program and static slicing with criterion (10, product)	13
1.12	A sample program and dynamic slicing with criterion (1, 10^1, z)	14
1.13	Example of forward and backward slicing	15
2.1	A sample program [29]	20
2.2	DODG of the sample program [29]	20
2.3	EDOPDG of the sample program [29]	20
2.4	Architecture of JSLICE Tool [31]	21
2.5	A sample C++ program [34]	23
2.6	The SDG of a program [34]	23
2.7	The updated SDG of the program shown in Figure 2.5 with colored nodes represented the slice [34]	24
5.1(a)	Code for Area class	32
5.1(b)	Code for Perimeter class	33
5.1(c)	Code for Volume class	33

5.2	Affected statements when IC anomaly occurs	33
5.3	System Dependence Graph of Input Program	34
5.4	System Dependence Graph (with different types of Edges)	34
5.5	Control Dependence Graph	35
5.6	Data Dependence Graph	35
5.7	Slicing criterion to perform slicing	36
5.8	Slice/Chunk of the Methods	36
5.9	Slice/Chunk of the statements of interest	37

List of Tables

Table No.	Table Description	Page No.
1.1	Dependence Graphs.	5
1.2	Faults and Anomalies due to Inheritance and Polymorphism [8].	9
2.1	Comparison of DOPDG and EDOPDG [29].	21
2.2	Comparison of Program Slicing Tools	26
2.3	Comparison of Dynamic Slicing of Object-Oriented Approaches.	26
2.4	Analysis of Anomalies present in OO System.	27

Chapter 1
Introduction

The Object-Oriented (OO) process is an evolutionary approach to software engineering which encompasses the entire software life cycle. Object-oriented technology is both immense and far-reaching process. The end users of computer systems notice the effects of object-oriented technology in the form of increasingly easy-to-use software applications. Since its inception in the early eighties, the object technology has made rapid progress and is now near to its maturity. The Object-Oriented software development style has become extremely popular, and is being widely used in industry as well as academic circles. For the software engineer, object-oriented technology encompasses programming languages, development methodologies, management of object-oriented projects and computer aided software engineering [1].

Object-Oriented approach has some properties like class, message, variable etc. in every application of software. These features are needed to be understood by maintainers and developers. Most of the software applications perform different types of software solutions. These software solutions are becoming complex and their superiority has been primarily bounded by the many factors like cost and time. Object-oriented language gives more flexibility to program but at the same time, it is very complex and difficult to debug and test for errors. This problem can result in a huge loss of various resources. It has been found that almost 50% of the software built today goes waste or unused because of complex structure of the programs. An object-oriented program incorporates various problems like complex dependencies, complex structure, inheritance, polymorphism, dynamic binding, *etc*.

1.1 Object-Oriented Features

Object-Oriented Programming (OOP) 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. Figure 1.1 represents the three main features of OO programming.

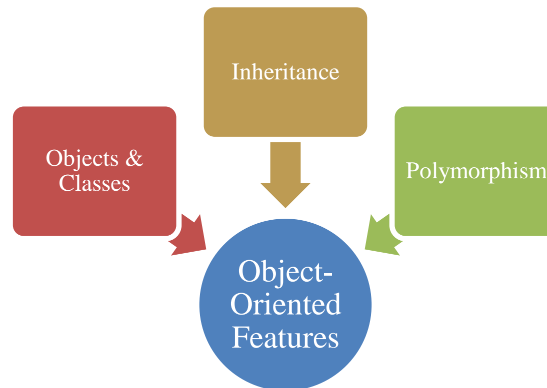


Figure1.1 Features of Object-Oriented language

1.1.1 Objects and Classes

In the Object-Oriented approach, a system is designed as a set of interacting objects. It is convenient to think of each object as representing a tangible real world entity such as a library member, an employee, a fan, a boy, *etc*. When the system is analyzed and developed in terms of natural objects, it becomes easy to understand the design and the implementation of the system. Similar objects constitute a class, means that objects possessing similar attributes and having similar methods would constitute a class.

1.1.2 Inheritance

The mechanism of deriving a new class from an old one is called inheritance. The base class or original class is known as parent class and the new class obtained through inheritance is called derived class. Each derived class can be considered as a specialization of its base class because it modifies or extends the basic properties of the base class in certain ways.

The advantage of using inheritance is:

- Reusability of Code.
- Helps to enhance the properties of a class.

There are two types of inheritance:

- Single inheritance (Only one base class)

Figure 1.2 represents a sample Java code having two classes: vehicle and car. Here vehicle class is our base class and car is derived class which is inheriting the properties of base class.

```
class vehicle
{
  vehicle ()
    {……}
int a;
…..
}
class car extends vehicle
{
    car ()
      {…… }
int b;
…….
}
```

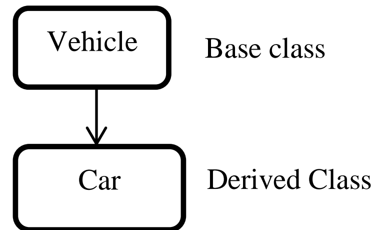


Figure 1.2 A sample Java code for single inheritance and its graphical representation

- Multilevel inheritance (Derived from a derived class)

Figure 1.3 represents a sample Java code which contains three classes: vehicle, car and racing_car. Here car class is derived from vehicle class and racing_car is derived from car class.

```
class vehicle
{
  vehicle ()
    {……}
  int a;
  …..
}
class car extends vehicle
{
    car ()
     {…… }
   int b;
  …….
}
class racing_car extends car
{
   racing_car ()
     {….}
    int c;
}
```

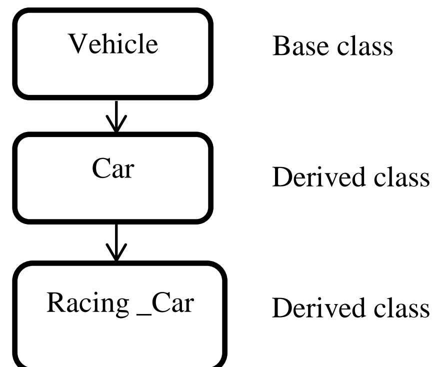


Figure 1.3 A sample Java code for multilevel inheritance and its graphical representation

1.1.3 Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means one name multiple forms. It is the ability of a reference variable to change behavior according to the object instance it is holding. Figure 1.4 represents hierarchical structure of polymorphism.

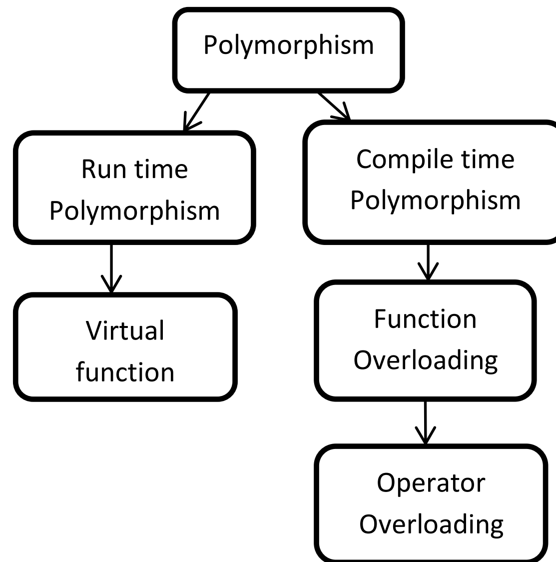


Figure 1.4 Hierarchical structure of polymorphism

1.2 Graphical Representation of Program

A graph G = (N, E) is defined as a finite set of nodes N and a finite set of edges E. It partitions the system into a number of smaller subsystems from the structural description. For example, a Control Flow Graph (CFG) describes the sequence in which the different instructions of a program get executed.

There are many other graphs such as Data Flow Graph (DFG), Program Dependence Graph (PDG), Control dependence graph (CDG), Data dependence graph (DDG), System Dependence Graph (SDG), Extended System Dependence Graph (ESDG), Class Dependence Graph (ClDG), Call- based Object-Oriented System Dependence Graph (COSDG), *etc* that represents dependence relationships among various components of a program. Table 1.1 represents various dependence graphs.

Table 1.1 Dependence Graphs

Dependence Graph	**Description of the graph**
Control Dependence Graph	Control dependencies for single procedure.
Data Dependence Graph	Data flow dependencies for single procedure.
Program Dependence Graph (PDG)	(Control+ Data) dependencies for single procedure.
System Dependence Graph (SDG)	(Set of PDGs + Interprocedural Calls) for whole system.

A dependence graph represents program features and dependencies between many objects. For example, in Figure 1.5, the execution of B & C depends on A. Similarly, the execution of D depends on C.

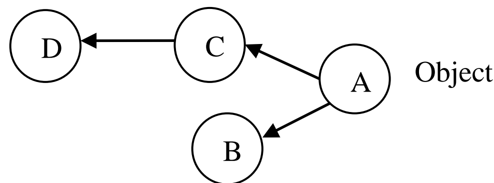


Figure 1.5 Dependence Graph Representations

The various types of dependence graphs are explained below:

- Control Dependence Graph: Control dependence graph shows dependencies between statements with the help of control conditions. Let C be a Control dependence graph with nodes x and y where x is a predicate node. Node y is control dependent on node x if there is at least one path from x to program exit that includes y and at least one path from x to program exit that that excludes y [2, 3].
- Data Dependence Graph: Data Dependence Graph represents flow of data from one statement to another statement. Let D be a data dependence graph with nodes x and y. Node y is data dependent on x if [4] :
  - ✓ Variable v is defined at x and used at y.

- ✓ There exists a path of non-zero length from *x* to *y* not containing any node that redefines variable *v*.

- Program Dependence Graph (PDG): The Program Dependence graph represents a program as a graph. It represents both control dependence and data dependence in a single graph [4]. Figure 1.6 represents the relationship between these graphs.

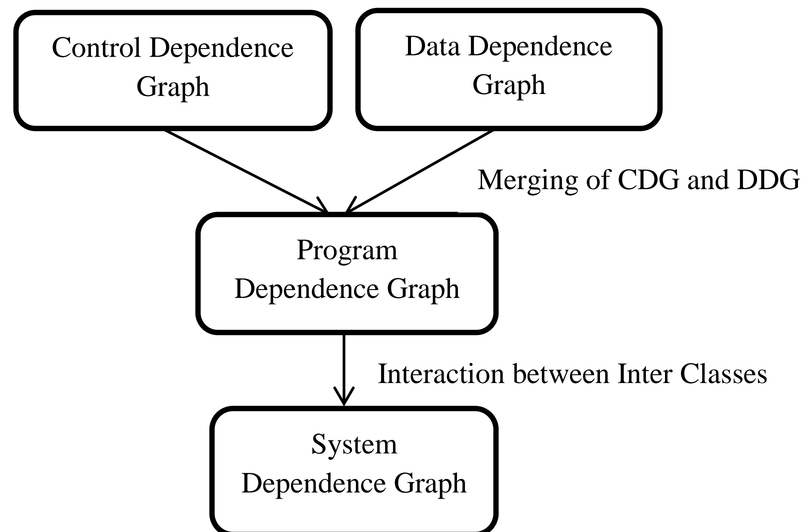


Figure 1.6 Relationship between Dependence Graphs

1.2.1 System Dependence Graph

A System Dependence Graph (SDG) is an extension of the program dependence graph [4]. It contains one procedure graph for each procedure. Each procedure dependence graph contains an entry vertex that represents entry into procedure. To represent parameter passing, an SDG associates each procedure entry vertex with formal parameter vertices: 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. An SDG associates each call site in a procedure with a call vertex and a set of actual-parameter vertices: 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 [5].

Larsen and Harrold [6] have developed their own SDG to accommodate the need of object-oriented programs. Their proposed representation helps in modeling both complete

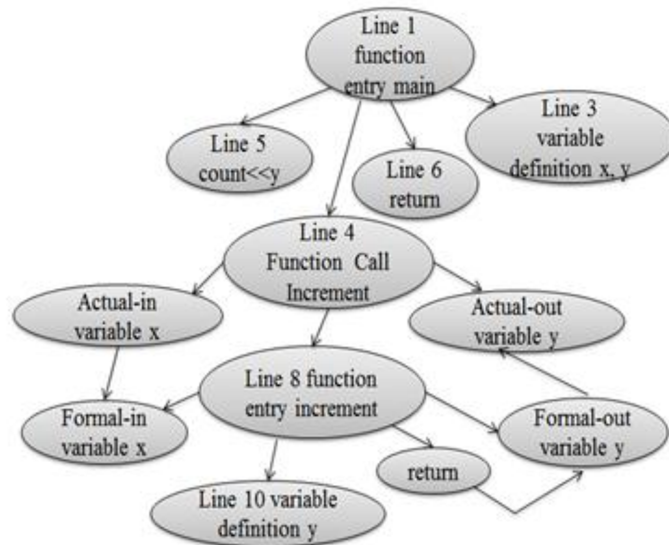
as well as incomplete systems. A brief discussion about how SDG is used to represent the above two systems are given below:

Incomplete systems: In order to facilitate analysis, it is needed to represent individual class of object-oriented software. It is done with the help of class dependence graph (ClDG) [6]. A ClDG captures the control and data dependence relationships that can be determined about a class without knowledge of the calling environment. The methods in a ClDG are represented using the same procedure dependence graph. Each method has a method entry vertex that represents the entry into the method [6].

Complete systems: Construct the SDG for a complete program by connecting calls in the partial system dependence graph to methods in the ClDG for each class. It involves connecting call vertices to method entry vertices, actual-in vertices to formal-in vertices, and formal-out vertices to actual-out vertices [6].

Figure 1.7(a) represents a sample program of C++ to increment the values of integer variables. There are two procedures in the program: main() and increment(). Figure 1.7(b) represents the system dependence graph of this program.

```
1)  int main ()
2)  {
3)      int x=3; y;
4)      y=increment (x);
5)      cout<<y;
6)      return 0;
7)  }
8)  int increment (int x)
9)  {
10)   int y=y+1;
11)   return y;
12) }
```



(a) (b)

Figure 1.7 (a) A Sample Program (b) System Dependence Graph of Sample Program [7]

1.3 Inconsistencies in Programming language

- Error: It is deviation from logic, syntax or execution. An error may produce an incorrect output or may terminate the execution of the program. There are two types of errors:

  Compile time error: The syntactical errors that are detected and displayed by the compiler are known as compile time errors. The most common compile time errors are:

  - ✓ Missing semicolons
  - ✓ Missing brackets
  - ✓ Misspelling of keywords
  - ✓ Incompatible data types and many more

  Run time error: Sometimes a program may compile successfully but may not run properly. Such program may produce wrong results due to wrong logic. Most common run time errors are:

  - ✓ Dividing an integer by zero.
  - ✓ Converting invalid string to a number.
  - ✓ Accessing an element that is out of bounds of an error.
  - ✓ Using the null object reference.
  - ✓ And many more.
- Bug: It is inconsistency between functional requirement and actual implementation. If something is already documented to be implemented but implementation is not consistent as per requirement then it is called a bug.
- Failure: The inability of a system or component to perform its required functions within specified performance requirements.
- Fault: An incorrect step, process, or data definition in a computer program which causes the program to perform in an unintended or unanticipated manner.
- Anomaly: Any condition which departs from the expected result. This exception can come from documentation (e.g requirement specification, design documents, user documents). An anomaly is not necessarily a problem in the software, therefore error, fault, failure can be considered as an anomaly.

1.3.1 Problems in Inheritance and Polymorphism

As shown in Table 1.2, there are nine types of anomalies that can occur due to inheritance and polymorphism.

Table 1.2 Faults and Anomalies due to Inheritance and Polymorphism [8]

Acronym	**Fault/Anomaly**
ITU	Inconsistent Type Use (Context swapping)
SDA	State Definition Anomaly
SDIH	State Definition Inconsistency (due to state variable hiding)
SDI	State Definition Incorrectly (possible post-condition violation)
IISD	Indirect Inconsistent State Definition
ACB1	Anomalous Construction Behavior (1)
ACB2	Anomalous Construction Behavior (2)
IC	Incomplete Construction
SVA	State Visibility Anomaly

1.3.2 Anomaly related to Constructor Methods

A constructor of the subtype always calls the corresponding construction of the supertype before execution. There are three types of anomalies related to this method:

- Anomalous Construction Behavior(1) (ACB1)

Consider the Figure 1.8, where the constructor of class C calls function f(). As actual type of object is D, so when C's constructor will call f(), then the constructor f() of D will be called instead of C's constructor f(). Also, D's constructor f() uses D.x variable which has not been initialized yet because of order of construction, that is, first C's constructor will be called then D's constructor will be called. Hence, it results in anomaly [8].

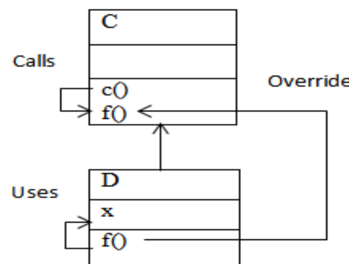


Figure 1.8 (ACB1) Anomalous construction behavior [8]

- Anomalous Construction Behavior(2) (ACB2)

This anomaly is similar to ACB(1), except that overridden method called by super class constructor makes use of the variable which is defined in the constructor of super class. But the definition is used after calling method and no definition of the variable exists in overridden method [8].

- Incomplete (failed) Construction (IC)

IC faults occur when all state variables are not defined in the constructor or a state variable is assigned the wrong value. In the context, a value is always right or wrong with respect to the specification.

There are two possibilities for faults here. First, the construction process may have assigned an initial value to a particular state variable, but it is the wrong value. That is, the computation used to determine the initial value that is having error. Second, the initialization of a particular state variable may have been overlooked. In case, there is a data flow anomaly between the constructor and each of the methods that will first use the variable after construction [8].

- Assigning a wrong value to a variable.
- No value is assigned to a variable.

Figure 1.9 represents an example to show Incomplete Constructor anomaly. Class ‘AbstractFile’ contains the state variable ‘fd’ that is not initialized by a constructor. The intent of the designing of AbstractFile is that a descendant class provides the definition of fd prior to its use, which is done by method open() in the descendant class SocketFile. If any descendant that can be instantiated defines fd and no method is called that uses fd prior to the definition, then no problem will occur. However, a fault will occur if either of these conditions is not satisfied.

Observe that while the designer’s intent is for a descendant to provide the necessary definition, a data flow anomaly exists within AbstractFile with respect to fd for methods read() and write(). Both of these methods use fd, and if either is called immediately after construction, a fault will occur. This design introduces an element of non-determinism

into AbstractFile since it is not known at design time what type of instance fd will be bound to, or if it will be bound (i.e. defined) at all.

Suppose that the designer of Abstract File also designed and implemented SocketFile, as also shown in Figure 1.9. By doing so, the designer has ensured that the data flow anomaly that exists in AbstractFile is declined by the design of SocketFile. However, this still does not eliminate the problem of non-determinism and the introduction of faults since, at some point in the future time, a new descendant can be added that fails to provide the necessary definition [8].

```
1   class abstract AbstractFile
2   {
3      FileHandle fd;
4
5     abstract public open();
6
7     public read() {fd.read (…);}
8
9     public write() {fd.write (…);}
10
11    abstract public close();
12  }

13  class SocketFile extends AbstractFile
14  {
15      public open()
16      {
17        fd = new Socket(…);
18      }
19
20      public close()
21     {
22       fd.flush();
23       fd.close
24     }
25  }
```

Figure 1.9 Incomplete Construction of state variable fd [8]

1.4 Program Slicing

Finding part of the statements in a program that directly or indirectly affect the value of a variable occurrence is referred to as Program Slicing. It is a method to analyze a particular program and used to extract a set of statements in a program which are relevant for particular computation.

The concept of slicing was first presented by Mark Weiser in 1979 and claims that slicing is very beneficial in the decomposition of a program unlike design methodologies. Various slicing criteria for the slicing of a particular program have been given [9]. When developers use program slices for debugging, they debug only defected area and not the whole program, which saves time and effort [10].

1.4.1 Slicing Criterion

Slicing is always done or carried out with some reference to a slicing criterion. Program slicing is a technique that decomposes the program by analyzing dependence relations between the statements. Slicing criteria consist of a pair <s, V> where s is called statement number and V is called as variable of interest [11].

1.4.2 Classification of Program Slicing:

Program slicing can be categorized into three sections:

- Types of Slicing
- Levels of Slicing
- Direction of Slicing

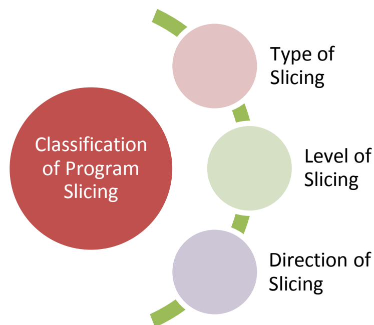


Figure 1.10: Classification of Program Slicing

1.4.3 Types of Slicing

- Static Slicing

Static slicing performs the slicing statically means it is used to detect only those parts of the program that possibly contribute to the computation of the selected function for all possible programs inputs. Static slices are computed by finding consecutive sets of indirectly relevant statements, according to data flow and control flow dependencies.

Figure 1.11 represents static slicing with slicing criterion (10, product). Static slices have large subprograms because of the imprecise computation of slices. In addition, static slices cannot be used in the process of understanding the program execution [12].

All variables that are not relevant to the computation of the sum are sliced away. Statistically available information is used for slicing, hence this type of slicing is called as static slicing.

1. read (n)	**1. read (n)**
2. i :=1	**2. i :=1**
3. sum :=0	3. sum :=0
4. product :=1	**4. product :=1**
5. while i <=n do	**5. while i <=n do**
6. sum := sum+i	6. sum := sum+i
7. Product := product * i	**7. Product := product * i**
8. i :=i+1	**8. i :=i+1**
9. write (sum)	9. write (sum)
10. write (product)	**10. write (product)**

Figure1.11: A sample program and static slicing with criterion (10, product)

- Dynamic Slicing:

Dynamic slicing is used to identify those parts of the program that contribute to the computation of the selected function for a given program execution (program input). Korel and Lasky proposed dynamic slicing which used dynamic analysis to identify only those statements that affect the variables at point of interest on the particular anomalous execution trace. In this way, the size of the slice can be considerably reduced [13, 14].

Dynamic slices are frequently much smaller than static slices. It also helps in the run-time handling of arrays and pointer variables. Static slicing gives less understanding of program execution due to larger size of program. Programmers may still have problems to understand the program and its behavior. The slicing tools usually developed provide limited support during the process of understanding of large programs and their executions. Therefore, it is important to develop some methods that will support the process of understanding of large software systems. One aid to understanding of large software systems is to use an intermediate representation of a program and then compute a slice from the graph. Dynamic slicing technique aims at giving a better understanding of large programs and their executions for a particular input [15].

A dynamic slicing criterion stipulates the input and differentiates between different occurrences of a statement in an execution. Figure 1.12 represents dynamic slicing criteria which consist of input, occurrence of a statement and variable. Dynamic Slice has slicing criterion (n = 1, 10^1, z), where 10^1 denotes the first occurrence of statement in the execution of the program. 'z' is variable on which we have to perform dynamic slicing.

- Input n is 1, c1, c2 both true
- Execution history is **1^1**, 2^1, 3^1, 4^1, 9^1, 2^2, 3^2, **4^2, 5^1, 6^1, 9^2**, 2^3, **10^1**
- Criterion < 1, 10^1, z >

1. read(n)	**1. read(n)**
2. for i:=1 to n do	**2. for i:=1 to n do**
3. a :=2	3. a :=2
4. if c1 then	**4. if c1 then**
5. if c2 then	**5. if c2 then**
6. a :=4	**6. a :=4**
7. Else	7. Else
8. a :=6	8. a :=6
9. Z :=a	**9. Z :=a**
10. Write (z)	**10. Write (z)**

Figure 1.12 A sample program and dynamic slicing with criterion (1, 10^1,z)

The difference between static and dynamic slicing is that dynamic slicing assumes fixed input for a program, whereas static slicing does not make assumptions regarding the input.

1.4.4 Levels of Slicing

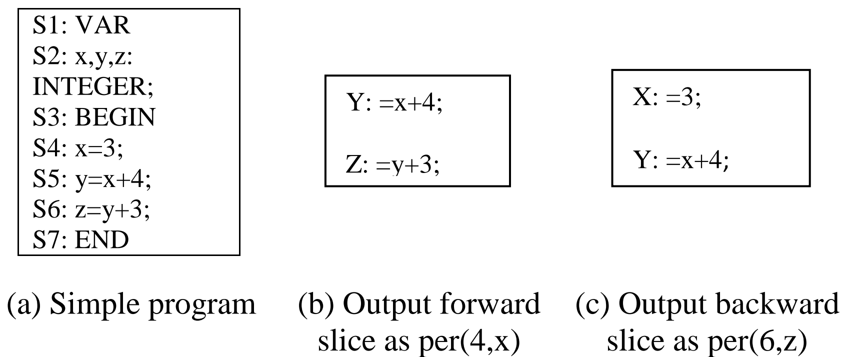
Intra-procedural slicing: It is performed within a single procedure. If any program have many functions calls and procedures then this is not useful. It handles procedures cautiously.

Inter-procedural Slicing: If the program consists of more than one procedure, inter-procedural slicing is used to compute slices which will span over multiple procedures [16].

1.4.5 Direction of Slicing

Backward Slicing: It is computed by tracking back the program code from the slicing criterion. It contains all parts of the program that might directly or indirectly affect the slicing criterion [10].

Forward Slicing: It is computed by tracking forward the program code from the slicing criterion. It contains those program elements or statements which will be affected by the slicing criterion. Figure 1.13 represents an example showing some statements for forward and backward slicing.



(a) Simple program (b) Output forward slice as per(4,x) (c) Output backward slice as per(6,z)

Figure1.13: Example of forward and backward slicing

1.5 Organization of Thesis

The rest of the thesis is organized as follows:

Chapter 2 – This chapter presents the literature survey conducted to study & explore the concept of dynamic slicing techniques for object-oriented programs.

Chapter 3 – This section describes the problem statement and contributes the gap analysis.

Chapter 4 – This part describes the methodology of work conducted.

Chapter 5 – This section focus on the experimental setup of the directed work.

Chapter 6 – This chapter describes the conclusion and future research work.

Chapter 2
Literature survey

2.1 Dynamic slicing

In 1988, Korel and Laski introduced a new type of slicing technique which opposed the Weiser static slicing. Dynamic slice computation is based on the input of program and used criterion [13]. In this new approach, only those statements are included which has relevance in the execution of program and their slice size is smaller than static slice. This dynamic slice is called executable subprogram. Static slicing is efficient in maintenance task but is less capable in debugging. Dynamic slicing helps in debugging, using which we can easily search faulty statements which cause programming errors [15]. Agarwal et al. in 1990 introduced new theories of dynamic slicing to make more precise results with the use of Dynamic Dependence Graph (DDG) [17]. Agarwal et al. also presented new techniques which compute dynamic slicing in the presence of unconstrained pointers. The backtracking in slicing is shown with the debugging tool called Spyder [18, 19]. Their method provided accurate results for supporting pointers, arrays and records, but does not support procedural calls in programs. Mariam introduced the concept of inter-procedural slicing to handle procedural calls and used System Dependence Graph (SDG) for intermediate representation [20].

2.2 Dynamic slicing of Object-Oriented Programs

2.2.1 Using Dynamic Dependence Graph

Algorithm for dynamic slicing using the program dependence graph was first presented by Hiralal and Horgon. In this technique dynamic slicing is used by marking nodes on a static program dependence graph. The computation is not always accurate because various dependencies might not hold for dynamic execution [21]. In their approaches, a new method was proposed for a precise dynamic slice using Dynamic Dependence Graph (DDG) [17, 21]. Zaho extended the DDG, in which the main task was to compute slices of Object-Oriented programs using dynamic Object-Oriented Dependence Graph

(DODG) [22]. The DODG is an arc to classify diagraph (V, A), where V is a finite set of elements called vertices and A is a finite set of elements of the Cartesian product V×V, called arcs. The relationship of control dependence and data dependence is the basis of DODG construction. In this approach, the number of nodes in a DODG is equal to the number of executed statements, which may be boundless for programs having many loops. Since the trace file is used to store the execution history, this technique is expensive [13, 22].

Later, Mohapatra et al. proposed another technique named Compact Dynamic Dependence Graph (CDDG). This technique mainly focuses on handling conditional statements to computing dynamic slicing. The advantage of this approach is that when the conditions are satisfied this type of slicing removes the parts of original program which does not affect the variable at the point of interest. In a program, the CDDG can be constructed after the termination of loop and it considers only those vertices and edges that were marked while omitting the rest. However, during the execution of a Loop Containing Conditionals (LCC), they do not create a new node every time a statement is executed. It is only based on path taken for iteration and the corresponding iteration number. Their tactic is more space efficient than previous approaches and is useful for C++ but it can also be adaptable for other languages like Java [23].

2.2.2 Using Forward Approach

Korel et al. proposed a novel approach of dynamic slicing called as forward computation, which eliminates the shortcoming of expensiveness in terms of space. Dynamic slicing is implemented on backward and forward slices. Both of these slicing techniques are equally good for short execution traces but for long execution traces forward slices are more accurate [24]. They did not consider this approach for the unstructured programs. To overcome this shortcoming, Korel also proposed an approach that computes the notion of a removable block in finding dynamic program slices. These algorithms resulted in more accurate dynamic slices as opposed to the existing algorithms that use control dependencies [25]. Song and Huynh suggested a technique to compute forward dynamic slicing of Object-Oriented programs using Dynamic Object Relationship Diagram (DORD). This approach is expensive in case when there are loops in the

statements. They compute the dynamic slices for each statement immediately after the execution which becomes more complex and expensive in nature [26].

2.2.3 Using Object Program Dependence Graph (OPDG)

Xu and Chen proposed an algorithm for program slicing of Object-Oriented programs which uses the Object Program Dependence Graph (OPDG).Agrawal proposes a dynamic slicing method by marking nodes or edges on a static program dependence graph during execution [17]. The result is not precise, because some dependencies might not hold in dynamic execution. Agrawal also proposes a precise method based on the dynamic dependence graph (DDG) [17], and Zhao applies it to slice object-oriented programs [22]. The shortcoming is that the size of the DDG is unbound. Korel [24], Song [26] and Tibor [27] propose forward dynamic slicing methods and Song also proposes method to slice OOPs using dynamic object relationship diagram (DORD) [26]. In these methods, they compute the dynamic slices for each statement immediately after this statement is executed. After the last statement is executed, the dynamic slices of all statements executed have been obtained. However, only some special statements in loops need to compute dynamic slices.

In this paper they suppose that a program includes three basic structures: sequence, branch and loop. For a sequential structure, if a statement is executed, all statements in the block will be executed. The static dependencies are the same as the dynamic ones. If they are not executed, mark them as "removed" (In their slicing algorithm, the executed statements are marked as "executed", the statements having no such mark are "removed"). The "removed" statements will be deleted first before traversing the static OPDG.

For a branch structure, in a certain execution only one branch is covered. Therefore only the executed statements might affect the dynamic dependencies. By removing the unexecuted statement the redundant dependencies can be removed from the OPDG. For the combination of sequential and branch structures, the result is the same as analyzing the two structures separately. As for a loop structure, the execution histories of some statements must be recorded.

In this approach, forward analysis is joined with backward analysis. In the forward analysis process, researchers mark nodes on the OPDG and compute intermediate dynamic slices. In the backward process, it traverses the OPDG marked to obtain the final dynamic slice. This algorithm is applicable only for C++ language but adaptable for other languages too. It was quite an expensive technique because all methods have to be analyzed before slicing and results are stored in libraries on disk [28].

Park presented an efficient approach on dynamic slicing using Efficient Dynamic Object-Oriented Program Dependence Graph (EDOPDG). This technique is compared with the traditional Dynamic Object-Oriented Program Dependence Graph (DOPDG) which shows that it is more efficient in terms of slice size as shown in table 2.1 [29].

A sample program present in Figure 2.1 and shows DOPDG and EDOPDG of sample program in Figures 2.2, 2.3.

```
E1: class Elevator{
     Public
2:  Elevator(int 1_top_floor)
3.  { current_floor=1;
4:     current_direction=UP;
5:    top_floor = 1_top_floor,}
6:  virtual~Elevator() {}
7: void up()
8:     { current_direction = UP;}
9: void down()
10.  {current_direction = DOWN;}
11: intwhich_floor()
12:   {return current_floor;}
13: Direction direction()
14:   {return current_direction;}
15: virtual void go(int floor)
16: { if(current_direction==UP)
17:    {while ((current_floor!=floor)&&
                    ( current_floor<=top_floor))
18:    add(current_floor, 1);}
       else
19:   { while((current_floor!=floor)&&
                    ( current_floor> 0))
```

```
23: class AlarmElevator:public Elevator{
   Public:
24: AlarmElevator(inttop_floor)
25:   Elevator( top_floor)
26:     {alarm_on = 0;}
27: void set_alarm()
28:      { alarm_on = 1;}
29: void reset_alarm()
30:      { alarm_on = 0;}
31: void go(int floor)
32:  { if(alarm_on)
33:      Elevator::go(floor)
       };
Protected:
Intalarm_on;
   };
34: main(intargc, char**argv){
      Elevator*e_ptr,
35: if(argv[1])
36:   e_ptr = new AlarmElevator(10);
else
37:  e_ptr = new Elevator(10);
38: e_ptr->go(3);
```

20: add(current_floor,-1);} } Private: 21: add(int&a, constint&b) 22: { a = a+b;}; Protected: intcurrent_floor, Direction current_direction; Inttop_floor; }	39: count<< “\n currently on floor:” <<e_ptr->which_floor()<<”\n”: }

Figure 2.1: A sample Program [29]

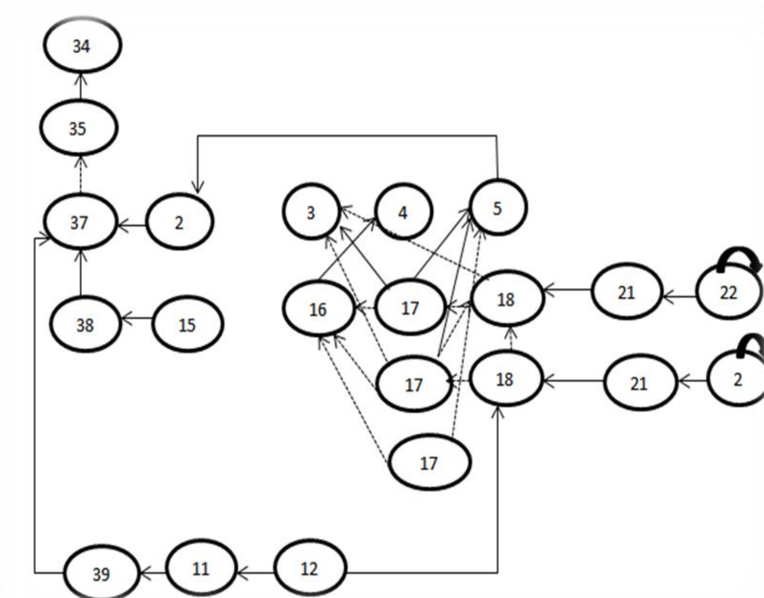


Figure 2.2 DODG of the sample program [29]

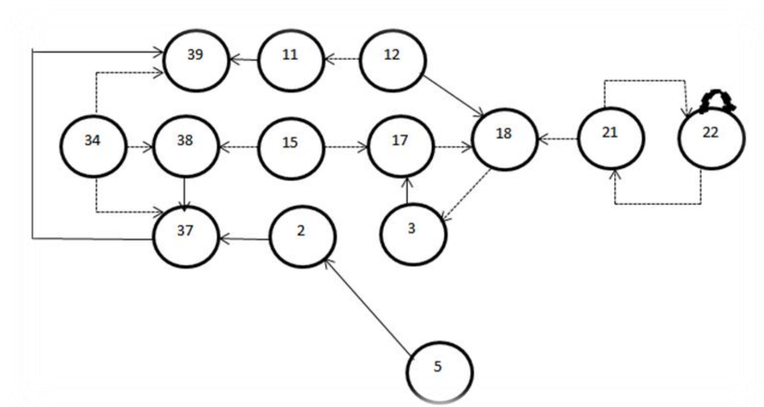


Figure 2.3: EDOPDG of the sample program [29]

The authors also did the comparison of their slices sizes between DODG and EDOPDG shows in Table 2.1 and they conclude that:

Table 2.1: Comparison of DOPDG and EDOPDG [29]

Type	Size of Slices
DOPDG	17
EDOPDG	14

2.2.4 Using Byte Code Traces

Wang et al. introduced a new approach of dynamic slicing algorithm for Java programs which is executed on compact byte code traces, and is useful in optimization. In this, the byte code stream is efficiently represented for the execution of Java program. Then the backward traversal of compressed program trace is executed to compute data/control dependences on-the-fly and slice is updated automatically [30]. This technique is space efficient, and demonstrates how dynamic slicing algorithms can instantly traverse compact traces without resorting to costly decompression [21]. Wang et al. extend their approach, which handles omission errors and the method was implemented on Kaffe virtual machine [31]. In their approach they presented a slicing tool called JSLICE as shown in Figure 2.4.

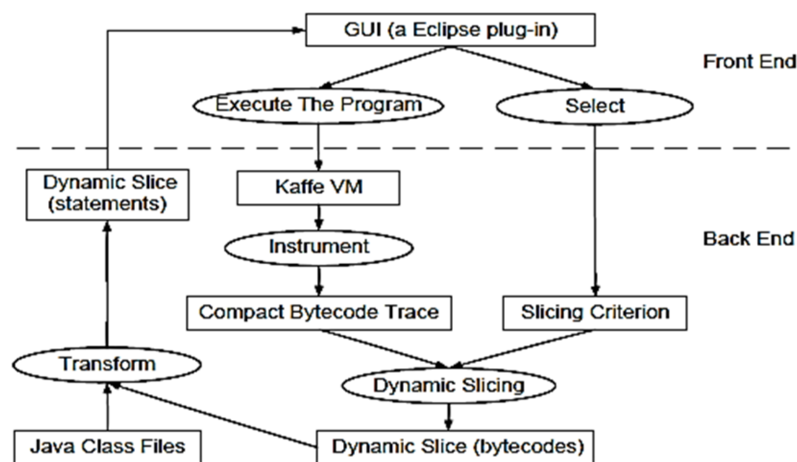


Figure 2.4 Architecture of JSLICE Tool [31]

2.2.5 Using System Dependence Graph (SDG)

Mohapatra et al. presented another technique for dynamic slicing of Object-Oriented programs, which extends the System Dependence Graph (SDG). 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 [32]. Mohapatra et al. proposed an algorithm called Node Making Dynamic Slicing (NMDS) and used Extended System Dependence Graph as an intermediate representation [33]. These algorithms are more efficient as they neither require any trace files for execution recordings, nor have they demanded any new nodes to be created & added to the intermediate representation at run time.

Later, Soubhagya under the guidance of Mohapatra gave a Graph Coloring Approach to Dynamic Slicing of Object-Oriented Programs in his master's thesis [34]. In this approach SDG is used as the intermediate representation of the C++ program and is compared with the Edge Making Dynamic Slicing (EMDS). When a statement invokes a method, they color the corresponding called node to handle method calls. Simultaneously, the corresponding parameter nodes representing the formal parameter vertices and actual parameter vertices are colored. Their algorithm is called as Contradictory Graph Coloring Algorithm (CGCA) which eliminates the restriction that "no two vertices that are sharing the same edge will have the same color" and they have taken the chromatic number of the graph as one i.e. χ (G) =1[34].

The Contradictory Graph coloring Algorithm for dynamic slicing of object-oriented programs takes the following input and output:

Input: Set of nodes of SDG (n1;n2;n3….;n_m), $n_c$being the node representing the slicing criterion, current state "sigma" of slicing criterion, set of variables V (x_1;x_2……x_j).
Output: Colored nodes showing the slice on the SDG.

In this they considered a sample C++ program as shown in Figure 2.5 which doing arithmetic operations and apply CGCA algorithm. Firstly they just represent the program as a System Dependence Graph as shown in Figure 2.6 and then apply algorithm to

shows graph coloring approach. After apply algorithm they show SDG again with colored nodes, these both SDGs shown in Figure 2.7. Theyupdated the graph by coloring those nodes which are able to state restrict the slicing criterion either by control dependence or by data dependence. In Figure 2.5, bold lines show the nodes of system dependence graph in Figure 2.6:

1. **class task{**	**8. i:=1;**
protected:	**9. while (i<11){**
inta,b,c;	**10. sum:=add(sum,i);**
int sum, I;	**11. i:=incr(i);**
public:	}
2. int add(int a, int b){	**12. count<< “SUM”= << sum;**
3. c:= a+b;	}
return (c);	main(){
}	task ob;
4. int incr (int a){	clrscr();
5. a:=a+1;	ob.fun();
return (a); }	}
6. void fun(){	
7. sum:=0;	

Figure 2.5 A Sample C++ Program [34]

Figure 2.6: The SDG of the program [34]

They traverse the nodes of the SDG backward in order to compute the backward dynamic slice of the given object-oriented program. In the SDG, node '10' represents the slicing point. They assume that the node representing the slicing point is the first node to be traversed and the process is continued till every node is traversed. they give an example for the traversal of node '9' ,and find that node '10' is control dependent on the reaching node '9'. Hence, color the node '9'. Similarly, during every iteration of the while loop, a new node is traversed and depending upon the existence of the type of dependence and state restriction, the reaching node is colored. One important point is that if a node is already colored, it needn't be re-colored again, even if it is found to state restrict the slicing criterion.

SDG with colored nodes representing the slice of the example program shown in Fig. 2.7 with respect to the slicing criterion < 10;sum >.

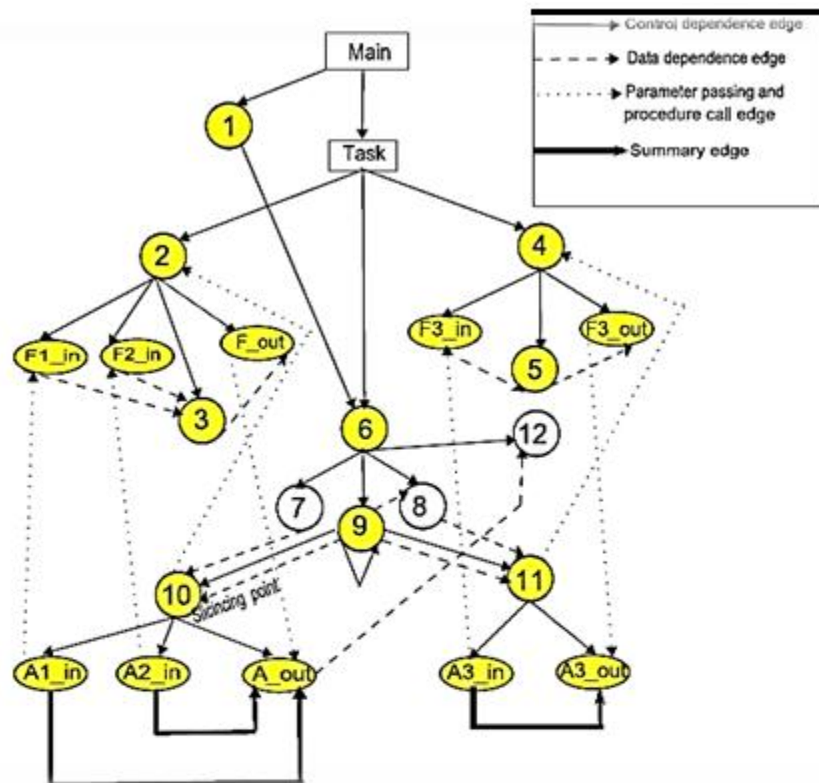


Figure 2.7: The updated SDG of the program shown in Figure 2.5 with colored nodes representing the slice [34]

Xi et al. presented an approach of Coarse-grained Dynamic Slice for Java Program. 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 [35].

2.2.6 Using Dynamic Catch Approach

Ohata et al. presented another technique called Dynamic Catch (DC) slicing, which overcomes the limitations of static and dynamic slicing. Dynamic Catch is a combination of both static and dynamic slices and it computes more accurate slices than static slicing. It also needs less computation time & memory space than dynamic slicing [36]. In this technique Object-Oriented Dependence Catch (OODC) is used for intermediate representation and a unique program slicing method was proposed to evaluate its effectiveness with Java.

2.2.7 Using Dynamic Impact Analysis Approach

Huang et al. presented an approach called dynamic impact analysis for Object-oriented program which calculates the parts of software system affected due to changes to the system. They examined how the identifications of runtime inheritance could be used by dynamic impact analysis for Object Oriented programs. To compute this, Java Dynamic Impact Analyzer (JDIA) tool is used that can be performed on Java Virtual Machine (JVM) and produces more accurate results than other present approaches [37].

2.2.8 Using Control Dependence Graph (CDG)

A technique and its algorithm were presented by Pani and Arundhati to find dynamic slicing for C++ programs. This proficiency computes the slice for objects & function over-loading. The shortcoming of this algorithm was that it did not find the slice for inheritance, polymorphism and dynamic binding [38]. Alexander et al. presented various types of faults in inheritance and polymorphism and explained it in detail how these faults can occur in object-oriented code [8, 39].

2.3 Tools of Program Slicing

Table 2.2 presents a comparison between existing program slicing tools on the basis of languages, direction and type of information [40, 41, 42]:

Table 2.2 Comparison of Program Slicing Tools

Tool Name	Target Language	Type of Information		Slicing Direction	
		Static	Dynamic	Forward	Backward
Kamker's	Pascal subset	N	Y	N	Y
Codesufer [43]	C,C++	Y	N	Y	Y
Spyder [19]	C	N	Y	N	Y
Wiscomsin Program Slicer [44]	C	Y	N	Y	Y
Schatz	FORTEN subset	Y	N	N	Y
FOCUS	C subset	Y	N	N	Y
Unravel [45]	C	Y	N	N	Y
Bandera[46]	Java	Y	N	Y	Y
IndusKaveri Java[47]	Java	Y	N	Y	Y
JSlice [48]	Java	N	Y	-	-

Y Supported, N Not Supported, - Information not available

2.4 Comparison of Dynamic Slicing of Object-Oriented Approaches

We studied above many approaches of Dynamic Slicing and make a tabular comparison of these approaches as shown in table 2.3.

Table 2.3 Comparison of Dynamic Slicing of Object-Oriented Approaches

S.No	Approach	Advantages	Disadvantages
1.	Dynamic Dependence Graph (DDG)[13, 20, 17, 22, 23]	• Provides efficient debugging tool for Object-Oriented programs. • Handles conditional statements.	• This approach is applicable only for C++ Language.
2.	Forward Approach[24, 25]	• This technique is efficient in computation of long execution of slice. • Handles unstructured Program.	• Expensive in terms of use of trace file for execution trace.
3.	Object Program Dependence Graph (OPDG)[28, 29]	• Use of backward and forward analysis in their approach. • Size of slice is small as compared to traditional approaches.	• Expensive in terms of use of trace file for execution trace.
4.	Byte Code Traces[19, 21, 31]	• This is Space Efficient technique because no use of trace file. • Handles Omission Errors (incorrect evaluation of branches)	• This approach does not handle the computation of Object-Oriented features like exception, multithreaded programs.
5.	System Dependence	• Space efficient in terms of	• This technique is

	Graph (SDG)[32, 33, 34, 35]	slice size. • Handles Coarse-grained Dynamic Slice.	applicable only for C++ programs; it does not handle concurrent object-oriented programs.
6.	Dynamic Catch (DC) [36]	• Overcomes the limitation of static slicing like debugging and dynamic slicing like software maintenance.	• This process is slow as compared to previous approaches and DC slice are large in size as compared to dynamic slice.
7.	Dynamic Impact Analysis [37]	• This approach computes more precise results than other dynamic impact analysis techniques.	–

2.5 Anomalies due to Inheritance and Polymorphism

Alexander et. al. represented nine types of faults in OO language features specially in inheritance and polymorphism. They also presented a fault failure model related to polymorphism for object-oriented programs in their thesis. They have discussed each anomaly with a suitable example to show the occurrence of the anomalies [8].

Jeff Offutt et. al described the anomalies related to inheritance and polymorphism feature of Object-Oriented Programming System [49]. Percentage of failure that can occur due to anomalies in OO software has been provided.

Schmitt et. al. presented a thesis on the use of inheritance and polymorphism in reliable software systems, in which these nine types of anomalies have been described. The main focus was on construction types of anomalies like ACB1, ACB2, IC. The use of constructor in OO language can cause new and different problems [50]. Table 2.4 gives the description of research work done by various researchers to present these anomalies:

Table 2.4 Analysis of Anomalies present in OO System

Year	**Author's Name**	**Description**
2001	Alexander	The nine types of faults with examples have been mentioned in the thesis work. It has mentioned that till now no one used any technique to debug these faults.
2001	Jeff Offutt	Faults, as discussed by Alexander have been discussed in a bit elaborated manner. It has been also presented that how our OO software can get affected by these faults.
2007	Peter H. Schmitt	Described the faults at run time and compile time.

Chapter 3
Gap Analysis and Problem Statement

3.1 Gap analysis in Existing Work

Based on the literature review of object oriented testing, testing techniques, slicing techniques and anomalies (related to inheritance & polymorphism) present in object oriented programs, following gaps have been identified [8, 49, 50]:

- No consolidate technique has been developed for debugging the anomaly is present in Object-Oriented programs.
- Till date no graph based technique for finding out a relative slice for IC anomaly has been designed [32, 33, 34, 35].
- Anomaly detection and its removal have not been given a thorough thought for concurrent as well as distributed programming system.
- There is a need to simulate the anomalous behavior related to polymorphism and inheritance for Java language as till date studies has been done for C++ only.

3.2 Problem Statement

After reviewing the literature of Object-Oriented testing technique, slicing technique and anomalies present in Object-Oriented programming, it has been analyzed that testing of Object-Oriented programming is one of the major area in which work can be extended in various directions like program slicing, testing distributed system, detection of anomalies present in an Object-Oriented system.

Program slicing is a technique for extracting points of computer program by tracing the program control and data flow related to particular data item. Existing technique only presents depiction related to the existence of anomalies associated with inheritance and polymorphism features of object-oriented programming.

It has been thoroughly analyzed that the process of detection and removal present in object-oriented programming can be enhanced further by using dependence graph and slicing techniques. System Dependence Graph (SDG) can be used for representing whole

interaction between various classes present in the code snippet. The slicing technique can be used to fetch the program lines related to particular variables so that the variable and the line/lines, which are actually resulting into incomplete Constructor(IC) anomaly can be detected.

Chapter 4
Methodology

The proposed work addresses the IC anomaly that arises due to inheritance and polymorphism feature of object oriented paradigm.

In the proposed methodology, following steps has been followed:

1. Generation of Control Dependence Graph (CDG) for the method/methods.
   - 1.1 Method named ProceduralDependenceGraphMatrix(), that is present in SystemDependenceGraph.jar, has been used that will provide an Adjacency Matrix for the method under consideration.
   - 1.2 Adjacency Matrix will provide the address of various nodes (statement) and a numeric value for Control dependence relationships. (Numeric two will represent control dependence between the two nodes)
   - 1.3 All the nodes that are related with control dependence relationship with a particular node will be listed in a separate array.
   - 1.4 Parent – child relationship has been identified between all the nodes so that a graph can be projected. This graph is known as Control Dependence Graph.
2. Generation of Data Dependence Graph (DDG) for the method/methods.
   - 2.1 Method named ProceduralDependenceGraphMatrix() from SystemDependenceGraph.jar has been used that will provide an Adjacency Matrix for the particular method.
   - 2.2 Adjacency Matrix will provide the address of various nodes and a numeric value for Data dependence relationships. (Numeric one will represent data dependence between the two nodes)
   - 2.3 All the nodes that are related with data dependence relationship with a particular node will be listed in a separate array.
   - 2.4 Parent – child relationship has been identified between all the nodes so that a graph can be projected, this graph is known as Data Dependence graph.

3. Generating Program Dependence Graph (PDG).
    3.1 Method named ControlDependenceBFSIterator() has been used for iteration through all the nodes. Methods named getname(), getancestornode() will be used for fetching the self-address as well as parent address of the particular node.
    3.2 After getting all the respective address of all parent nodes and its children, a consolidated graph can be constructed that will provide the merged view of CDG & DDG, here control as well as data dependence will be shown.

4. Generating System Dependence Graph (SDG)
    4.1 ConstructGraphClassAdapter has been used that take Class name as input and it will provide the method list.
    4.2 For all the methods Control dependency will be computed at Class level and System level (when inheritance is present).

5. Extracting the desired program chunk/slice that may participate in Anomalous Construction that may result either from assigning a wrong value to a variable or no value is assigned to a variable.
    5.1 Select the particular variable and statement of interest.
    5.2 With the help of Adjacency Matrix find out the nodes having data dependence relationship with the selected node and put these nodes in a set (let set N contains all the selected nodes from n_1-----------n_k).
    5.3 Search the set N for the nodes, whose parent node will be having function call to other modules and put that parent node in set M.
    5.4 Find out the child nodes for every node present in set M.
    5.5 Search for the nodes that are Data dependent on the nodes selected in step 5.4.
    5.6 If selected nodes are related to variable of interest, then put these nodes into set N.
    5.7 Set N will provide the desired chunk that may result into anomalous behavior.

Chapter 5
Experimental Results

"SystemDependenceGraph.jar" from Java System Dependence Graph API [51] 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.

5.1 Implementation

A Prototype has been customized in the Java language to implement the steps that were discussed in section 4.1. It takes Java byte-code as input for fetching out the chunk related to Incomplete Construction (IC) anomaly.

Consider the example of Java program shown in Figure 5.1(a), (b), (c). The programs are stored under a package named as inheritance. Figure 5.1(a) represents code for Area class, Figure 5.1(b) represents code for Perimeter class and Figure 5.1(c) represents code for Volume class. As shown in Figure 5.1(a), Area class is using two variables namely length and breadth, where both are initialized to value 1. A function is defined, named calculate(), which is returning an integer value using the formula length multiplied by breadth.

```
1.   package inheritance;
2.      public class Area{
3.        protected int length;
4.        protected int breadth;
5.           public Area()
6.            {
7.               length = breadth = 1;
8.            }
9.          public long calculate()
10.          {
11.            return length*breadth;
12           }
13      }
```

Figure 5.1(a) Code for Area class

```
1.        package inheritance;
2.          public class Perimeter extends Area {
3.          public Perimeter() {
4.           super();
5.           }
6.         public long calculate() {
7.             return 2 * (length + breadth);
8.          }
9.         }
```

Figure 5.1 (b) Code for Perimeter class

```
1.     package inheritance;                        11.    }
2.       public class Volume extends Area {        12.    public static void main(String[] args)
3.       protected int height;                     13.    {
4.       public Volume() {                         14.      Volume obj = new Volume();
5.           super();                              15.     System.out.println("Volume="+obj.calculate());
6.           height = 1;                           16.      Perimeter p = new Perimeter();
7.         }                                       17.      System.out.println("Perimter="+p.calculate());
8.       public long calculator()                  18.      }
9.        {                                        19.    }
10.      return super.calculate() * height;
```

Figure 5.1 (c) Code for Volume class

IC anomaly can occur due to assigning wrong value to a variable. For example, as shown in Figure 5.2, Volume class is extending the Area class, means it is inheriting the properties of area class. In class Volume, we are calculating volume of a rectangle by using variables length and breadth from its base class. If we assign wrong value to a variable (e.g. height), the whole calculation may get affected. The statements which will be affected due to wrong value assignment of height variable in volume class have to be extracted/fetched so that statement having wrong assignment can be tracked.

```
1.     package inheritance;                        11.    }
2.       public class Volume extends Area {        12.    public static void main(String[] args)
3.       protected int height;                     13.    {
4.       public Volume() {                         14.      Volume obj = new Volume();
5.           super();                              15.    System.out.println("Volume="+obj.calculate());
6.           height = 1;                           16.      Perimeter p = new Perimeter();
7.         }                                       17.     System.out.println("Perimter="+p.calculate());
8.       public long calculator()                  18.      }
9.        {                                        19.    }
10.      return super.calculate() * height;
```

Figure 5.2 Affected statements when IC anomaly occurs

On running the tool, the first screen shown in Figure 5.3 will be displayed. Here screen has been divided into various frames, Frame A represents the program in tree structure, where package name will be at the top and methods are represented as the subparts.

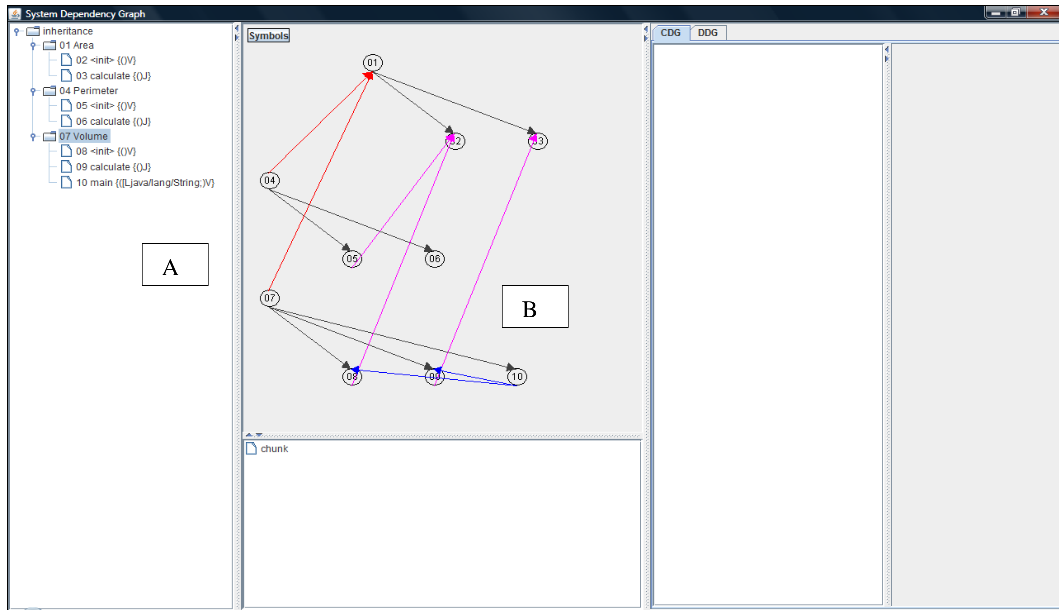


Figure 5.3 System Dependence Graph of Input Program

As shown in Figure 5.4, Frame B represents the SDG of the program selected as input. Block marked with S represents various SDG symbols used 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.

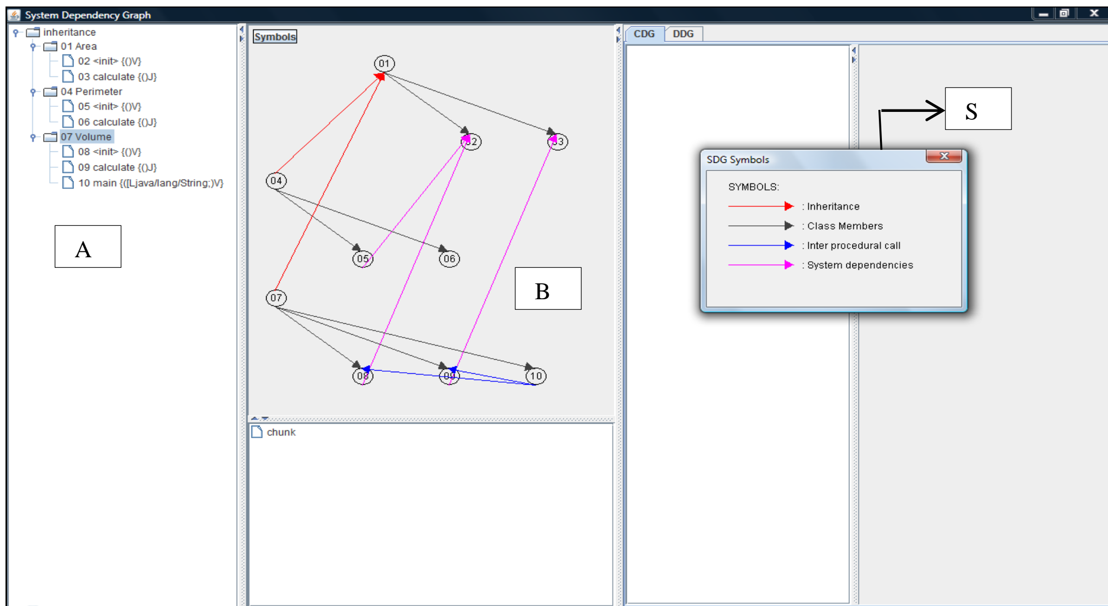


Figure 5.4 System Dependence Graph (with different types of Edges)

In the Figure 5.5 Frame C contains two tabs, named as: CDG and DDG. When any method will be selected from the input program that present in Frame A, its respective code will be shown in Frame C and its corresponding CDG will be generated in Frame D.

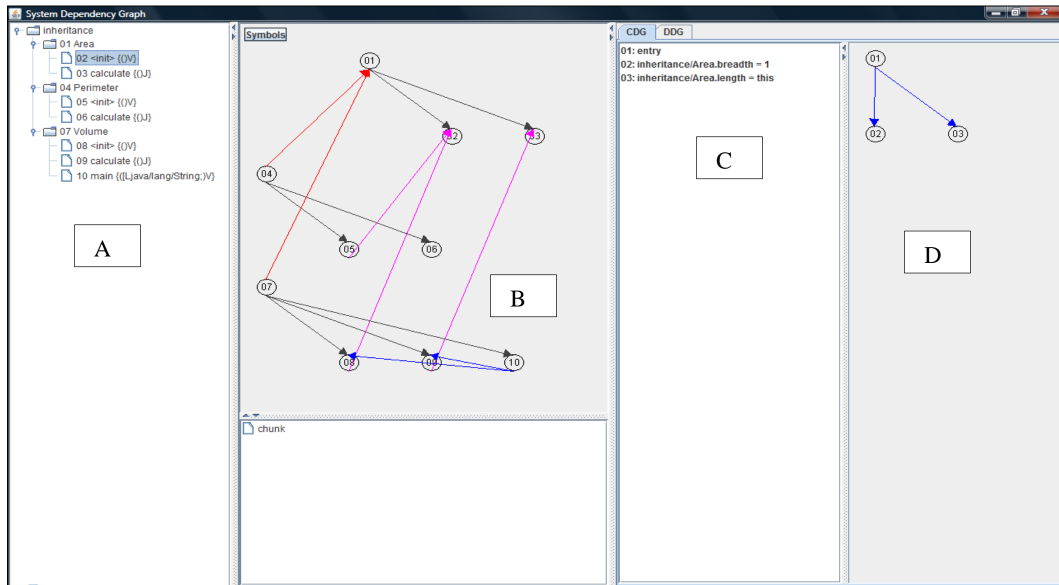


Figure 5.5 Control Dependence Graph

As represented in Figure 5.6, when DDG tab will be clicked in Frame C, its corresponding data dependence graph will be generated in Frame D.

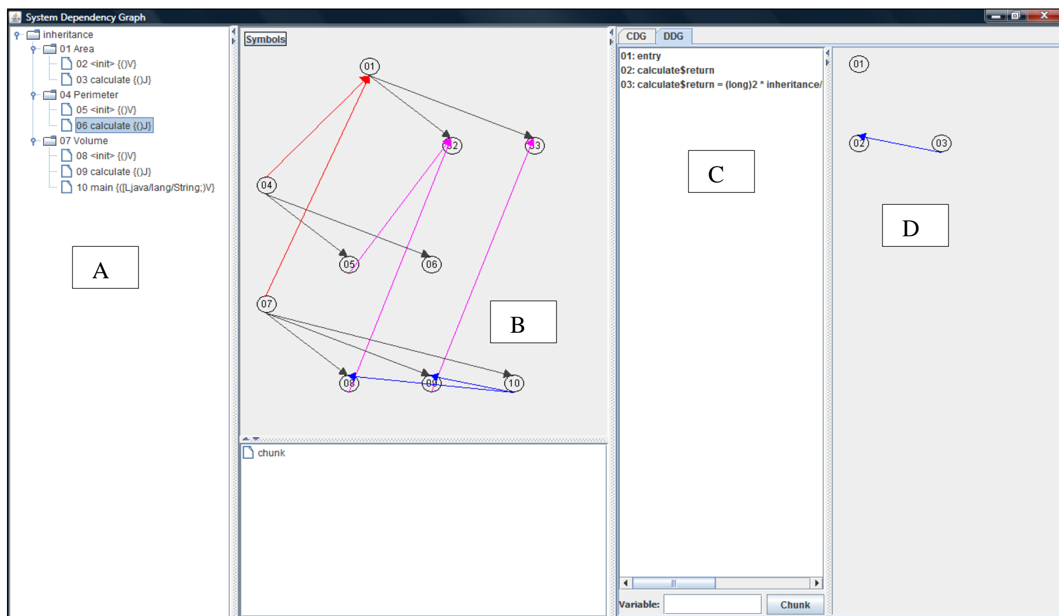


Figure 5.6 Data Dependence Graph

For finding the set of statements that may participate in IC anomaly, a slicing criterion has to be provided, that contain a statement number and a variable name. Consider Figure 5.7 in which slicing criterion has been shown by selecting the statement from Frame C and providing variable name in the text box present in Frame F.

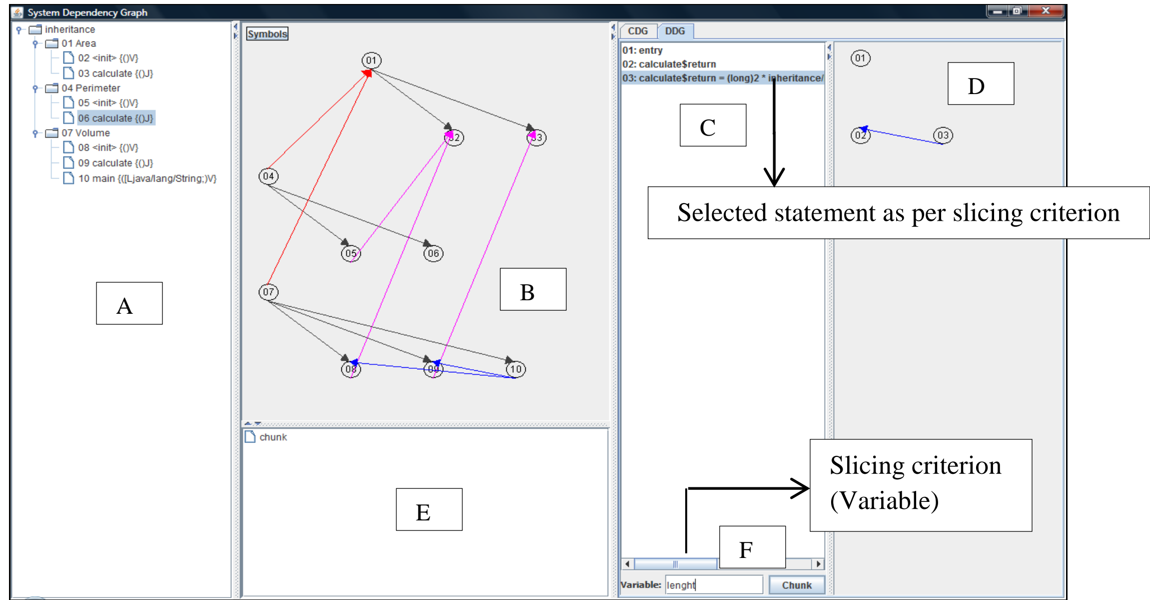


Figure 5.7 Slicing criterion to perform slicing

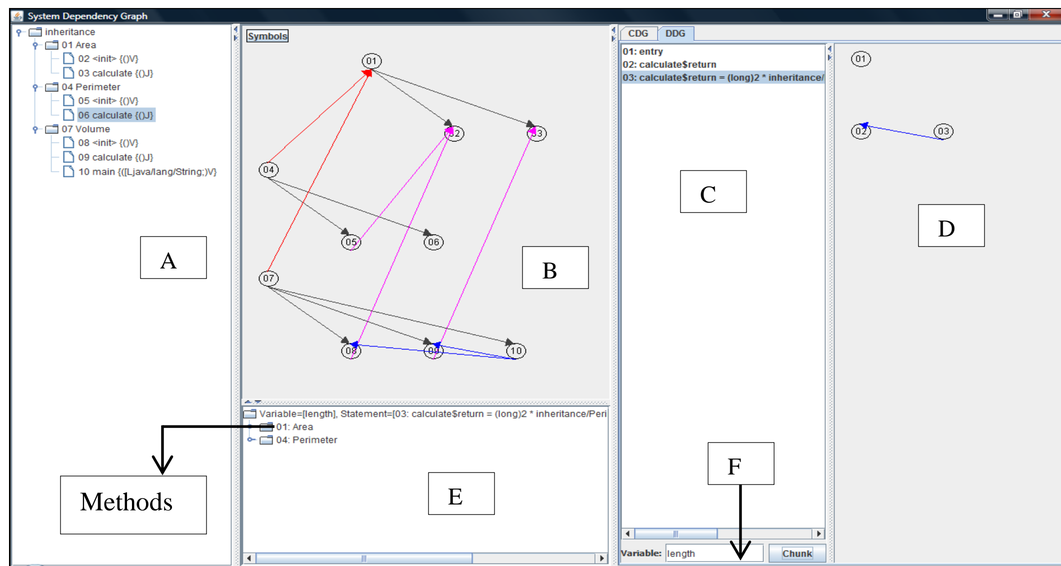


Figure 5.8 Slice/Chunk of the Methods

After clicking on button named Chunk, the respective chunk of statements will be displayed in Frame E. As shown in Figure 5.8, the chunk in Frame E will provide all the

methods in a tree structure, that may participated in the IC anomaly. As shown in Figure 5.9, the chunk represented in Frame E will provide all the nodes or statements that can affect the variable under consideration. The individual statements related to the particular methods can be fetched by further expanding these functions.

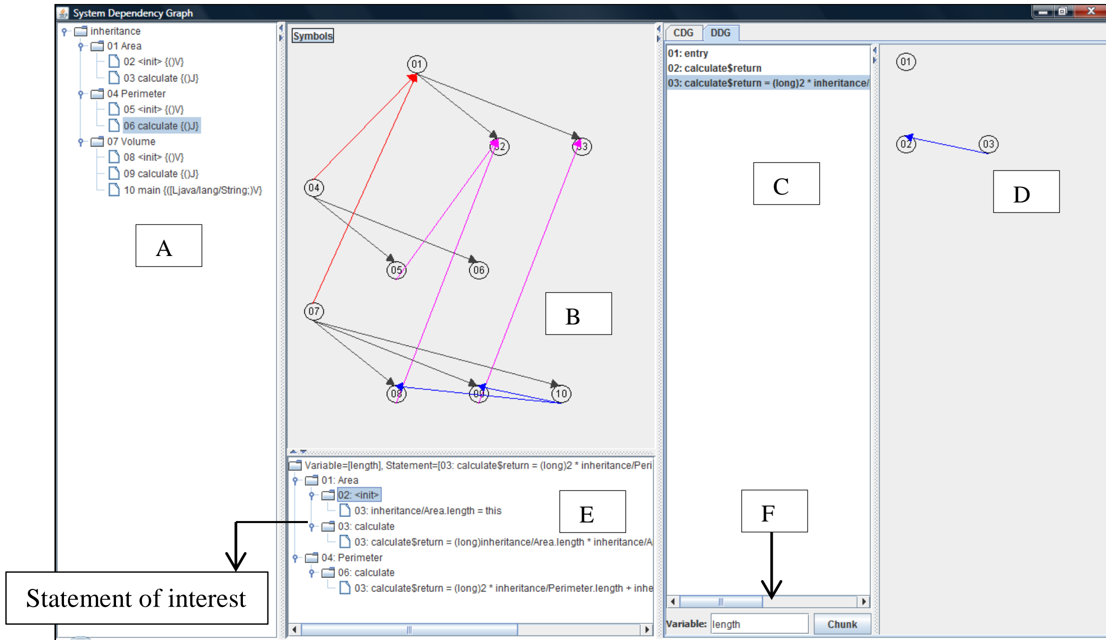


Figure 5.9 Slice/Chunk of the statements of interest

Chapter 6
Conclusion and Future Scope

A new technique based on system dependence graph, has been proposed for object-oriented software to address the problem of "Incomplete Construction (IC)" that can arise due to inheritance and polymorphism.

6.1 Conclusion

The major contribution of this work is given below:

- The key contribution of the technique is to generate the program slice related to incomplete construction (due to unassigned variable).
- The foundation of the proposed technique is System Dependence Graph. With this representation developer can better understand the inheritance relationship within the software.

6.2 Future Work

The proposed technique has focused on the generation of chunk for IC anomaly but still there are the following points that can be explored further.

- The proposed technique can be extended to handle other problems related to Constructor like Anomalous behavior constructor(1), Anomalous behavior constructor(2), *etc*.
- Work can be extended further to solve more challenges of OOPs like private constructor and destructor, *etc*.
- Technique can be explored further extended for concurrent and distributed programming.

[1] V. Berard, "Object-Oriented Concepts", Available at: http://www.ipipan.gda.pl/~ marek /objects /TOA/oobasics/oobasics.html [25th February 2012].

[2] T. Ball, "The Use of Control Flow and Control Dependence in Software Tools", PhD thesis, Computer Science Department, University of Wisconsin-Madison, 1993.

[3] A. P. Mathur, " Foundations of Software Testing", Pearson/Addison Wesley, 2008.

[4] J. Ferrante, K. J. Ottenstein, J. D. Warren, "The Program Dependence Graph and its use in Optimization", ACM Transactions on Programming Languages and Systems, Vol. 9, No. 3, July 1987.

[5] D. Liang and M. Harrold, " Slicing object using system dependence graph", In Proceedings of International Conference on Software Maintenance, pp.358, 1998.

[6] L. D. Larson and M. J. Harrold," Slicing objectoriented software". In Proceedings of the 18th International Conference on Software Engineering, German, March 1996.

[7] K. L. Kumawat, "Prioritization of Program Elements based on their Testing Requirements", Btech thesis, National Institute of Technology, Rourkela, May, 2009.

[8] R. T. Alexander, "Testing the polymorphic relationships of object-oriented programs", PhD Thesis, George Mason University, Information Technology and Engineering, 2001.

[9] M. Weiser, "Program slicing", Proceedings of the 5th international conference on Software engineering, pp. 439-449, March 1981.

[10] M. Weiser, "Programmers use slices when debugging", Communications of the ACM, vol. 25, no.7, pp.446 – 452, July 1982.

[11] M. Weiser , " Program slicing" [J] IEEE Transaction on Software Engg, pp. 352-357 , 1994.

[12] Tip. Frank "A survey of program slicing techniques", Journal of Programming Languages, Vol. 3, no. 3, pp. 121–189, September 1995.

[13] B. Korel and Laski J., "Dynamic program slicing", Information Processing Letters, Vol. 29, no. 3, pp. 155-163, 1988.

[14] B. Korel and Laski J., "Dynamic slicing of computer programs", The Journal of Systems and Software, vol. 13, no. 3, pp. 187-195, 1990.

[15] Andrea de Lucia. "Program slicing: Methods and applications", International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society Press, pp. 142-149, 2001.

[16] S. Horwitz, T. Reps, and D. Binkley, “Inter-procedural slicing using dependence graphs”, ACM Transactions on Programming Languages and Systems, vol. 12, no.1, pp. 26 – 61, January 1990.

[17] H. Agrawal and J. Horgan, “Dynamic program slicing”, In Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation, White Plains, New York, vol.25, no.6, pp. 246 – 256, June 1990.

[18] H. Agrawal, R. A. DeMillo, and E. H. Spafford, “Dynamic slicing in the presence of unconstrained pointers”, In Proceedings of the ACM Fourth Symposium on Testing, Analysis and Verification, pp. 60 – 73, July 1991.

[19] 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, June1993.

[20] M. Kamkar, “Inter Procedural Dynamic Slicing with Applications to Debugging and Testing”, PhD thesis, Linkoping University, Sweden, 1993.

[21] Durga Prasad Mohapatra, Rajib Mall, Rajeev Kumar, “An Overview of Slicing Techniques for Object-Oriented Programs”, INFORMATICA, pp.253-277, June 2006.

[22] J. Zhao, “Dynamic slicing of object-oriented programs”, Information Processing Society of Japan, Technical reportSE-98-119, pp.17-23, May 1998.

[23] D. P. Mohapatra, R. Mall, R. Kumar, “A Novel Approach for Computing Dynamic Slices of Object-Oriented Programs with Conditional Statements”, In Proceeding of India Annual Conference, pp. 478-482, December 2004.

[24] B. Korel and S. Yalamanchili, “Forward computation of dynamic slices”, In proceedings of International Symposium on Software Testing, August 1994.

[25] B. Korel, “Computation of Dynamic Program Slices for Unstructured Programs”, IEEE Transactions of Software Engineering, vol.23, no.1, pp.17-34, January 1997.

[26] Y. Song and D. Huynh, “Forward Dynamic Object-Oriented Program Slicing”, IEEE Symposium on Application- Specific Systems and Software Engineering and Technology, pp.230 – 237, March 1999.

[27] G. Tibor, et al. An Efficient Relevant Slicing Method for Debugging, Software Engineering Notes, Software Engineering ESEC/FSE'99 Springer ACM SIGSFT, pp. 303-321, 1999.

[28] 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.

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

[30] T. Wang, A. Roy Choudhury, "Using compressed byte code traces for slicing Java programs", In Proceedings of IEEE International Conference on Software Engineering, pp. 512 – 521, May 2004.

[31] T. Wang and A. Roy Choudhury, "Dynamic Slicing on Java Bytecode Traces", In proceeding of ACM Transactions on Programming Languages and Systems, vol. 30, no. 2, March 2008.

[32] D.P.Mohapatra, R.Mall, and R.Kumar, "An edge marking dynamic slicing technique for object-oriented programs", In Proceedings of 28th IEEE Annual International Computer Software and Applications Conference, pp. 60 –65, September 2004.

[33] 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.

[34] S. S. Barpanda, "A Graph Coloring Approach to Dynamic Slicing of Object-Oriented Programs", Master's Thesis, Department of Computer Science and Engineering, National Institute of Technology, Rourkela, Orissa, India, 2010.

[35] 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.

[36] F. Ohata, K. Hirose, M. Fuji, and K. Inoue, "A slicing method for object-oriented programs using dynamic light weight information", Proceeding of Eighth Asia-Pacific Software Engineering Conference, China, pp.273-280, December 2001.

[37] L.Huang, Y. Song, "A dynamic impact analysis approach for object-oriented programs," In Proceedings of the Conference on Advanced Software Engineering and Its Applications, Hainan Island, pp. 217–220, December 2008.

[38] S. Pani, P.Arundhati,"An Approach to find dynamic slice for C++", Journal of computer science and engineering, vol.1, no.1, May 2010.

[39] R. T. Alexander, J.Offutt, J. M. Bieman, “Syntactic Fault Patterns in OO Programs”, Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems, November 2002.

[40] Tommy Hoffner, “Evaluation and Comparison of Program Slicing Tools”, Department of Computer and Information Science, Technical report LITH-IDA-R-95-01, Likoping University, Sweden, 1995.

[41] Program Slicing, Available at: http://www.cs.ucl.ac.uk/staff/ucawxe/lectures/3C05-04-05/ProgramSlicing-Essay.pdf [25February 2012].

[42] Raula Gaikovonakula, “Using Program Slicing Metrics for the Analysis of Bug Fixing Processes”, Master’s thesis, Department of Information System, Graduate School of Information, Nara Institute of Science and Technology NAIST, February 2010.

[43] Tool for Static Slicing CodeSurfer, Available at: http: // www.grammatech. com/ products/codesurfer/ [25th February 2012].

[44] Tool for Static Slicing Wisconsin, Available at: http://www.cs.wisc.edu/wpis/ slicing tool/ [25th February 2012].

[45] Tool for Static Slicing Unravel, Available at: http://hissa.nist.gov/unravel/ [25th February 2012].

[46] Bandera Project, Available at: http://bandera.projects.cis.ksu.edu/ [25th February, 2012].

[47] Tool for Static slicing Indus Project, Available at: http://indus.projects.cis.ksu.edu/ [25th February 2012].

[48] Tool for Dynamic Slicing JSlice, Available at: http://jslice.sourceforge.net/ [25th February 2012].

[49] Roger T. Alexendor, A. Jefferson Offutt, and Andreas Stefik, “Testing Coupling Relationship in Object-Oriented Programs”, George Mason University, Vol. 20, No. 4, June 2001.

[50] Peter .H. Schmitt, “Use of Inheritance and Polymorphism in Reliable Software Systems”, Kaslsruhe University, March 2007.

[51] 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 [25th February 2012].

List of Publications

Published:

[1] M. Jindal, V. Arora, "Survey on dynamic slicing of object-oriented programs", IJMAN of International Forum of Researchers Students and Academician, Issue No. 2, Vol. No. 2, May 2012.