Slicing of Object-Oriented Software

Thesis submitted in partial fulfillment of
the requirements for the degree
Of
Bachelor of Technology
In
Computer Science and Engineering

By

Biswaranjan Panda

Roll No: 10606020

Sagardeep Mahapatra

Roll No: 10606033

Ved Prakash

Roll No: 10506053

Under the Guidance of

Prof. D. P. Mohapatra

May, 2010

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela, 769008

Certificate

This is to certify that the project entitled "Slicing of object-oriented software", submitted by Biswaranjan Panda, Sagardeep Mahapatra and Ved Prakash, B.TECH students in the Department of Computer Science and Engineering, National Institute of Technology, Rourkela, India, in the partial fulfillment for the award of the degree of Bachelor of Technology, is a record of an original research work carried out by them under our supervision and guidance. The thesis fulfills all requirements as per the regulations of this Institute and in our opinion has reached the standard needed for submission. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. D. P. Mohapatra
Department of Computer Science and Engineering
National Institute of Technology Rourkela
India – 769008

Acknowledgement

On the submission of our Thesis report, we would like to extend our gratitude and sincere thanks to our supervisor Dr. D.P. Mohapatra, for his constant motivation and support during the course of our work in the last one year. We truly appreciate and value his esteemed guidance and encouragement from the beginning to the end of this thesis. He has been our source of inspiration throughout the thesis work and without his invaluable advice and assistance it would not have been possible for us to complete this thesis.

We would also like to give our most sincere thanks to Dr. Rajib Mall, Professor, Department of Computer Science, IIT Kharagpur for his invaluable guidance and advice on the topic. A special acknowledgement goes to Mr Swarnendu Biswas, a MS scholar at IIT Kharagpur and Mrs. Mitrabinda Ray, a PhD scholar for her guidance throughout the thesis. Finally, our sincere thanks to Prof. B. Majhi for motivating us to perform better and better and providing us with all the resources required to carry out the thesis.

Biswaranjan Panda

Sagardeep Mahapatra

Ved Prakash

Abstract

Software maintenance activities generally account for more than one third of time during the software development cycle. It has been found out that certain regions of a program can cause more damage than other regions, if they contain bugs. In order to find these high-risk areas, we use slicing to obtain a static backward slice of a program. Our project deals with the implementation of different intermediate graphical representations for an input source program such as the Control Dependence Graph, the Program Dependence Graph, the Class Dependence Graph and the System Dependence Graph.

Once a graphical representation of an input program is obtained, slicing is performed on the program using its System Dependence Graph and a two pass graph reachability algorithm proposed by Horwitz, to obtain a static backward slice.

TABLE OF CONTENTS

List of Abbreviations

ANTLR : Another tool for language recognition

CFG : Control flow graph

DDG : Data dependence graph

CDG : Control dependence graph

PDG: Program dependence graph

SDG: System dependence fgraph

CLDG : Class dependence graph

ESDG : Extended system dependence graph

sdom : semi-dominator

idom : immediate dominator

List of figures and tables

Chapter 1

Introduction

Most of the transactions performed in today's world use different types of software solutions. These software solutions are becoming quite complex and their quality have been primarily bounded by the cost and time factors. Also, the focus of building software has seen a dramatic drift from using traditional procedural techniques to object-oriented techniques. Object oriented technique, no doubt modularizes the program, but at the same time, it is very complex and difficult to debug and test for errors. It has been found that almost 50% of the softwares built today go unused because of their inability to meet the above mentioned constraints, which in turn results in a huge loss of time, money and manpower. Software testing activities are hence very essential for the construction of reliable software.

Various methods have been developed to test softwares for errors. These methods apply different approaches toward software testing which use various intermediate forms. Intermediate graph representation of a program is one such convenient representation. It includes various graphs like control flow graph, data dependence graph, control dependence graph, program dependence graph, system dependence graph, etc to represent the program structure and the relations between different program constructs. This representation can be further used in different areas of software engineering that includes activities like slicing, program debugging, software testing, regression testing, etc.

Slicing is an important technique which has a wide range of applications in software testing. Basically, slicing is a technique for simplifying programs by focusing on selected aspects of semantics. It is method of program analysis which is used to extract a set of statements in a program which is relevant for a particular computation. This set of statements is called a program slice. Various type of slicing strategies exist such as forward slicing, backward slicing, static slicing, dynamic slicing, etc. These different slicing techniques have different application domains such as software maintenance, software optimization, program analysis, information flow control, etc.

1.1 Motivation for our project

Usually testing of the software products is carried out in various levels to identify all defects existing in the software product. However, for most practical systems, even after satisfactorily carrying out the testing process, we cannot guarantee that a software product is error free. This situation is caused by the fact that input data domain of most software products is very large. Hence, it is practically impossible to test the software exhaustively with all the sample test cases. It is quite obvious that not all the lines in the source code contribute to the error at a particular location. We therefore need not consider the whole source code in the testing process and only focus on those areas that are more likely to have caused the error. In order to find these high-risk areas, we need to construct an intermediate representation of a sample input program called as the dependence graph, slice the graph obtained and distribute the testing efforts accordingly.

1.2 Objective of our project

Our objective is to construct the intermediate representation of a sample input program and use it to find a static backward slice of any statement in the program.

1.3 Organization of the project

The rest of the project is organized as follows:

Chapter 2

We present the basic concepts related to the intermediate representation of a graph, like the control flow graph, data dependence graph, control dependence graph, program dependence graph, system dependence graph and extended system dependence graph that is used to represent the input program. We also cover some basic concepts of slicing and its different types.

Chapter 3

In this chapter we review some of the related work done in this area.

Chapter 4

In this chapter, we present the different proposed algorithms that are required for the construction of the intermediate graph representation along with our algorithm to compute the system dependence graph. We also discuss a two-pass slicing algorithm proposed by Larsen and Harrold [2] to compute the static backward slice of a statement in an object oriented program.

Chapter 5

In this chapter we give an overview about Eclipse, ANTLR and Graphviz, the tools that we have used in our project and also present the implementation details of our project which are concerned with the construction of the intermediate representation and the static backward slicing of a program and finally discuss the results.

Chapter 6

We conclude the project and discuss the future work that can be done in this area.

Chapter 2

Basic Concepts

In this chapter we discuss the basic concepts and terminologies associated to our work and that are used in later sections.

2.1 Program Representation

In this section, we study about the intermediate representation of a sample program and the methods followed to construct this representation.

2.1.1 Control Flow Graph

A Control Flow Graph is a directed graph with a unique entry node START and a unique exit node STOP, where each node is a statement in the program. There is a directed edge from node P to node Q in the control flow graph if control may flow from block P directly to block Q. Edges in a CFG are of two types. An edge is called a T edge, if control flows along that edge when the predicate at the origin evaluate to true and vice versa [15].

```
x=10;
count=5;
while(count>0){
        if(x<20)
                inc(x);
        count=count-1;
}
```

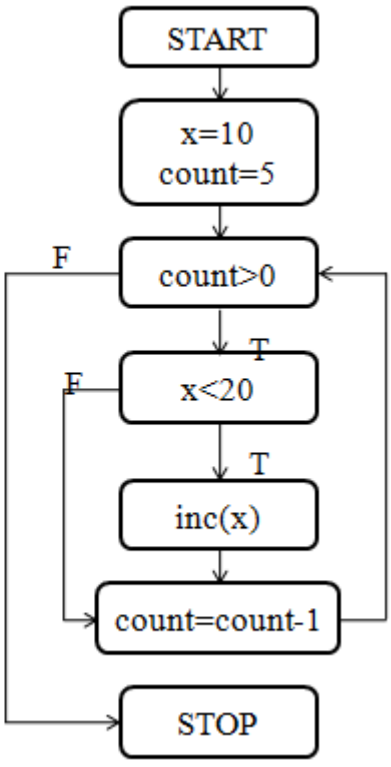

Figure 2.1: Control flow graph for the sample program

2.1.2 Data Dependence Graph

Data dependence over a control flow graph exists from node X to Y if the following conditions are satisfied [15]

- Node X defines variable, say V
- Node Y uses the variable V for computation
- Control can flow from X to Y and along the flow path and there should not be any intervening definition of the variable V.

If node Y is data dependent on node X, then X is called the reaching definition of Y.

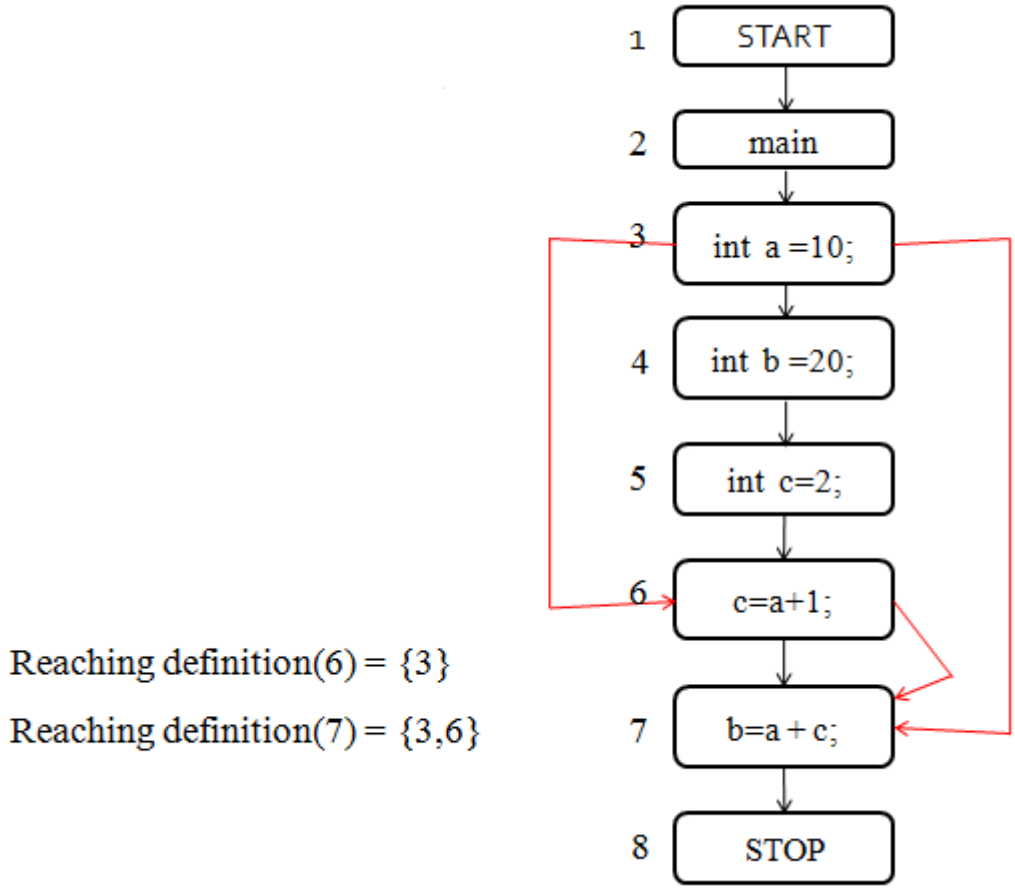

Figure 2.2: Data dependence graph for the sample program

Reaching Definition

If in a program, node Y is data dependent on node X, i.e. a variable defined at X is used at Y, then, X is said to be the reaching definition of Y. For computation of reaching definition,

every node is assigned a unique label, which is usually a number. We also use define some terminologies that will be used for the computation of the reaching definitions [15].

Computation of Reaching Definition [15]

Different sets are needed for the computing reaching definition as mentioned by Steindl [15].

Definition 2.1 Def-set: The definition set of variable x contains as its elements the labels of all definitions that define x.

Definition 2.2 Gen-set: The gen-set of statement S contains as its elements the labels of all definitions that are generated by S.

Definition 2.3 kill-set: The kill-set of statement S contains as its elements the labels of all definitions that are killed by S.

Definition 2.4 in-set: The in-set of statement S contains as its elements the labels of all definitions that reach S.

Definition 2.5 out-set: The out-set contains as its elements the labels of all definitions that leave S.

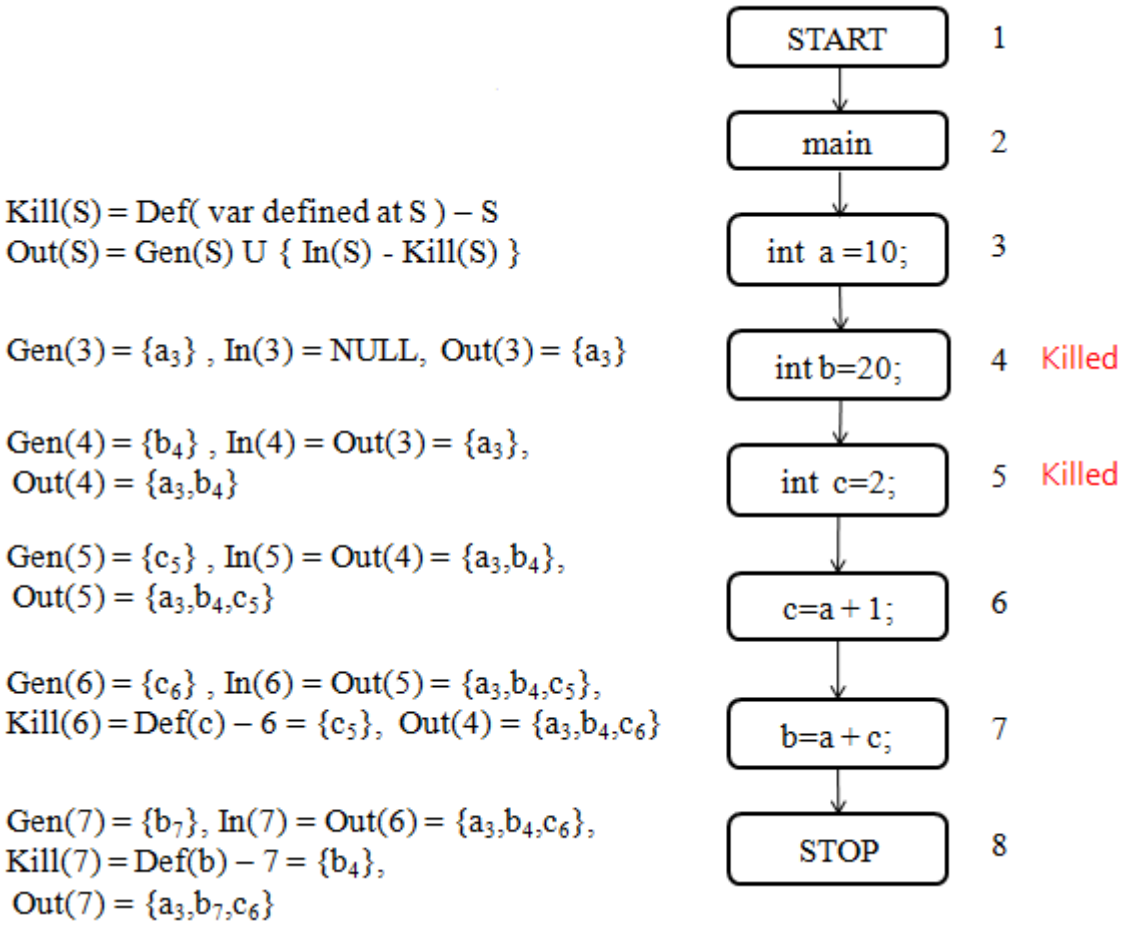

Figure 2.3: Computation of reaching definition for the program in Figure 2.2

2.1.3 Control Dependence Graph

Before presenting the concepts of control dependence, it is essential to understand the concepts of post-dominance, immediate post-dominator and post-dominator tree. We discuss each of them one by one.

Definition 2.6 Post-dominator: In a directed graph with exit node STOP and beginning node START, we say that a node P in the graph post-dominates another node Q in the same graph, if and only if all paths from Q to STOP has to pass through P. We call P, a post-dominator of Q [15].

Definition 2.7 Immediate Post-dominator: We call P the immediate post-dominator of Q, if and only if P is the post-dominator of Q, P is not equal to Q, and there is no other node R in the graph, such that P is a post-dominator of R and that is itself a post-dominator of Q [15].

Definition 2.8 Post-dominator tree: The post-dominator tree of a directed graph G with exit node STOP is the tree that consists of the nodes of G, has the root STOP, and has an edge between nodes P and Q if P immediately post-dominates Q. To construct the post-dominator tree, we need to find out the immediate post-dominators of each node in the control flow graph. This has been illustrated taking one simple example. Following is a CFG and its corresponding post-dominator tree has been shown [15].

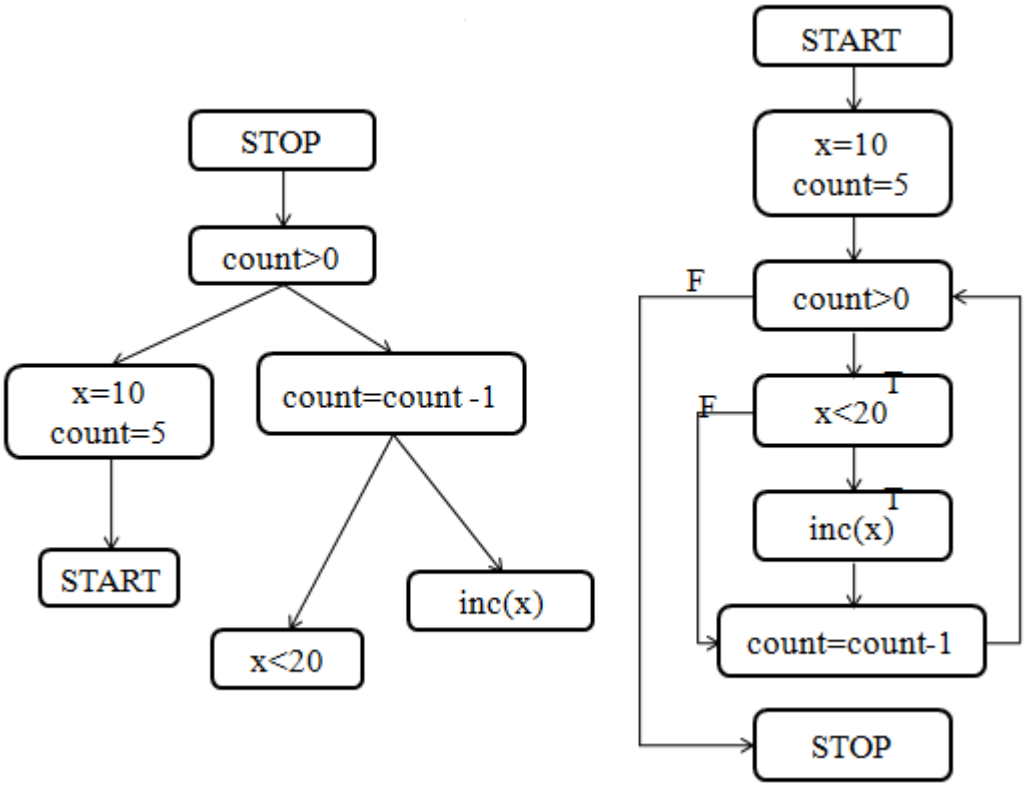

Figure 2.4: Corresponding post-dominator tree for the control flow graph

Construction of the post-dominator tree

- To construct the post-dominator tree, we need to find out the immediate post-dominators of each node in the control flow graph.
- Finding out the post-dominators in the control flow graph is same as finding out the dominators in the reverse control flow graph.
- Immediate dominator of each node is determined by using the concept of semi-dominators.

2.1.4 Program dependence graph

The program dependence graph G [4, 13, 14] of a program P is the graph G = (N, E), where each node n belonging to N represents a statement of the program P. The graph contains two kinds of directed edges: control dependence edges and data dependence edges. A control (or data) dependence edges (m, n) indicates that n is control (or data) dependent on m. Note that the PDG of a program P is the union of a pair of graphs: Data dependence graph and control flow graph of P [6].

2.1.5 System dependence graph

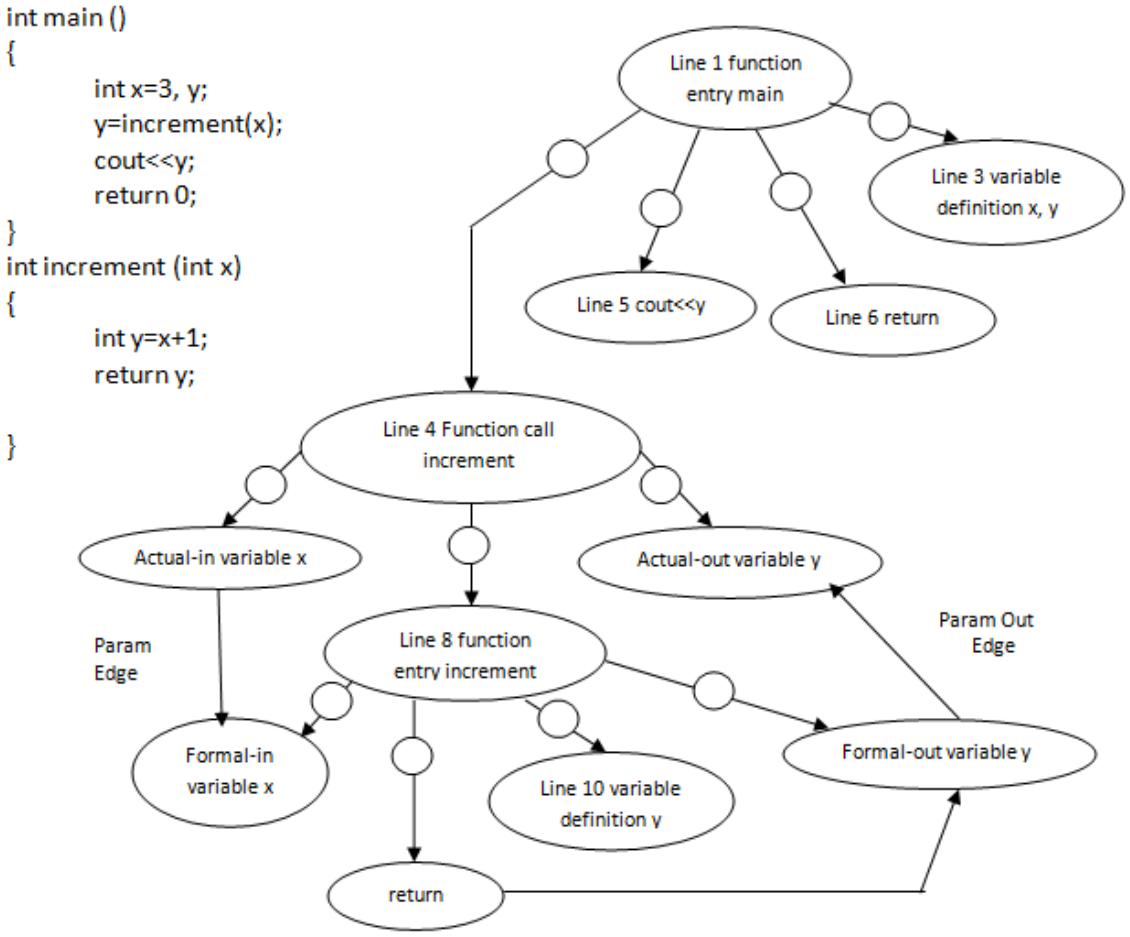

Figure 2.5: Corresponding program dependence graph for the sample program

The PDG cannot handle procedure calls. Horwitz et al [4, 13, 14] introduced the System Dependence Graph (SDG) representation which models the main program together with all associated procedures. SDG is actually a collection of PDGs. For programs without procedure calls, the PDGs and SDGs are similar. For construction of an SDG, first the PDGs of all the procedures are constructed individually and then the SDG is constructed by integrating all the PDGs [6].

SDG takes the help of different types of nodes to model procedure calls and parameter passing [2]. They include the following:

- Procedure call statements are represented by call site nodes in the program.
- Actual-in and actual-out nodes represent the input and output parameters at call site.
- They are control dependent on the call-site nodes.
- Formal-in and formal-out nodes represent the input and output parameters at called procedures. They are control dependent on procedure's entry node.

Different edges are used to link the above nodes in a system dependence graph as proposed by Larsen and Harrold [2]. They are as follows:

- Call edges link the call-site nodes with the procedure entry nodes.
- Parameter-in edges link the actual-in nodes with the formal-in nodes.
- Parameter-out edges link the formal-out nodes with the actual-out nodes.
- Summary edges are added to represent the transitive dependencies that arise due to procedure calls.

The SDG can be extended further to implement object-oriented features. The graph that we obtain is called the extended system dependence graph and has been discussed below.

2.1.6 Extended system dependence graph

The extended system dependence graph is used to represent the programs with object oriented features that include data hiding, inheritance, polymorphism, etc. It is also called as a Class dependence graph (CLDG) [2].

A CLDG captures the control and data dependence relationships that can be determined about a class without the knowledge of calling environments. Each method in a CLDG is represented by a procedure dependence graph. Each method has a method entry vertex that represents the entry into the method. A CLDG also contains a class entry vertex that determines the entry into the class. The class entry vertex is connected to the method entry vertex for each method in the class by a class member edge. Class entry vertices and class member edges let us quickly access the method information when a class is combined with another class or system [2].

In a CLDG, each method entry is expanded by adding formal-in and formal-out vertices. Formal-in vertices are used for each formal parameter that is added and formal-out vertices for each formal reference parameter that is modified by the method. Additionally, formal-in

and formal-out vertices are also added for global variables referenced in the method. Since the class's instance variables are accessible to all methods in the class, we treat them as global to methods in the class and we add formal-in and formal-out vertices for all reference variables referenced in the method. However, the exception to this representation for instance variable is that formal-in vertices for the instance variables in the class constructor and formal-out vertices for the instance variables in the class destructor are omitted [2].

2.2 Slicing

Program slicing is a method of program analysis which is used to extract a set of statements in a program which is relevant for a particular computation. This set of statements is called a program slice. It therefore, computes the statements which affect the value of a variable at a particular point in the program. Program slicing was originally introduced by Mark Weiser as "a method for automatically decomposing programs by analyzing their data flow and control flow starting from a subset of a program's behavior, slicing reduces that program to a minimal form that still produces that behavior. The reduced program called a slice is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of the behavior." The input to the program slicing algorithm is usually an intermediate representation of the sample program that is to be tested, and the output is program slice [15].

2.2.1 Slicing criterion

Slicing is always carried out or computed with reference to a slicing criterion. The slicing criterion is represented as <S, V>. S is the statement whose slice is to be computed and V is the variable that has been used at S [15].

2.2.2 Types of slicing

Program slicing is broadly categorized into the following types.

Static and dynamic slicing

In static slicing, the input program is statically analyzed to compute the program slice i.e. static slicing method considers all possible input values while computing program slices [6,

7, 8]. The input values are not restricted in any manner and therefore predicates may evaluate to either true or false. Since all possible input values are considered, it is a conservative method for computation of program slices. Dynamic slicing is a method for computing a program slice with respect to a particular sequence of execution of a program [9, 10, 11, 12]. Since only a particular execution sequence is considered, the predicate value may either evaluate to true or false. Therefore, only the actual slices are computed for a particular input [15].

Forward and backward slicing

This type of slicing computes the program slice that consists of statements in a program which are affected by the value of a variable at a particular statement in the program. So, if <S, V> is the slicing criterion, then the slice for <S, V> is the set of all the statements that are potentially affected by the value of the variable V at statement S. This type of slicing computes the program slice that consists of statements in a program which affects the value of a variable at a particular statement in the program. So, if <S, V> is the slicing criterion, then the slice for <S, V> is the set of all the statements that potentially affect the value of the variable V at statement S [15].

2.2.3 Difference between static and dynamic slicing

Consider the following example.

```
s1:     main() {
s2:     int x = 10;
s3:     int y = 20;
s4:     x = x + y;
s5:     if (x>y) {
s6:         x = x – y;
s7:         y = 2 * y;
s8:     }
s9:   else {
s10:        y = y / 2;
s11:    }
s12:    }
```

Figure 2.6: A sample program

Let us consider the statement s5. The static slice of s5 consists of s6, s7 and s10. But, the dynamic slice of s5 will consist of s6 and s7 or s10 only depending on whether s5 evaluates to true or false since a dynamic slice is always computed with respect to a particular execution of the program. Consider the above case and assume the value of x to be 10 and the value of y to be 5. Now, since x is greater than y, during the execution of the statement s5 will evaluate to true and hence the slice of s5 will compute of statements s6 and s7 [15].

2.2.4 Difference between forward and backward slicing

Taking the same example of 2.5, we can find the forward slice of s6 to consist of s7, s8 and s11 (if static slicing is used), because these are the statements that can be potentially affected by s6. Similarly, the backward slice of s6 consists of s2, s3 and s4 because they determine the value of s6 [15].

Chapter 3

Related Work

Many theories have been proposed regarding the intermediate representation and also many ways have been defined to obtain the representation. In 1987, Ferrante [4] proposed an algorithm for the construction of the control dependence graph by using the control flow graph and the post-dominator tree for an input program. In 1979, Lengauer and Tarjan [1] introduced a fast algorithm to determine the dominator tree of the control flow graph of a program. Their algorithm to find the dominator tree used the concept of immediate dominators and semi-dominators for every node. Computation of semi-dominator is used as an intermediary step in the immediate dominator computation. They also introduced several properties of semi-dominators and immediate dominators.

Different theories have been proposed for slicing as well. In 1982, Weiser [8] defined slice with respect to a slicing criterion <S, V>, where S is a program point and V is subset of variables at that point. The slices he computed are primarily executable programs and were obtained by removing zero or more statements from the original program. In his proposed algorithm he used data flow analysis of the control flow graph of the program to compute inter-procedural and intra-procedural slices.

Another definition of the slicing criterion was given by Ottenstein and Ottenstein [18] that defined a slicing criterion as <s, v> where s is a program point and v is a variable defined at v. They used a graph reachability algorithm to compute a static slice which consisted of the statements that affect the variable v at the program point s.

Horwitz et al [2] constructed an inter-procedural program representation called the system dependence graph and came up with the two pass static backward slicing algorithm to find out the static backward slice of a statement in an object-oriented program. This algorithm is more precise than the previous one proposed by Ottenstein because it uses the summary information at the call site nodes to account for the calling context of the procedure. In the first pass of the two pass graph reachability algorithm, he traversed along the summary edges to slice across the cal vertices that have transitive dependencies on actual-in vertices. In the

second pass the methods in the program are marked by traversing along the parameter-out edges.

Larsen and Harrold [2], in 1996, enhanced the system dependence graph to represent object-oriented software and used the two phase algorithm of Howritz et al with minor modifications to compute static slices. However, this did not address the dynamic aspects of slicing. A forward slice on a slicing criterion <s, v> is defined as a set of all statements which are affected by the variable v at the program point s.

In 2006, Jehad Al Dallal introduced a method for computing intra-procedural static forward slices by traversing the dependence graphs only once. In this algorithm called as the ComputeAllForwardSlices, he used a function called ComputeAFSlice and each node in the PDG is associated with an empty set before applying the algorithm. After the algorithm is applied, the set associated with a node n consists of the lines of code included in the slice computed at node n . It builds the set associated with each node in the PDG incrementally as the function called ComputeAFSlice is applied recursively.

But the slicing techniques described above were for sequential programs. However, most softwares developed today are concurrent and distributed in nature. For slicing these programs different forms of slicing algorithms which are suitable for slicing concurrent programs are required. However, very less work ha been reported for the same till now.

Chapter 4

Slicing of object-oriented programs

Our objective is to find a static backward slice of a sample program. In order to implement the two pass slicing algorithm proposed by Larsen and Harrold [2], we need to have an intermediate representation of the sample program i.e. the system dependence graph of the program. We first construct the control flow graph of the program. Then, we construct the data and control dependence graphs and merge them to obtain the program dependence graphs. Then we construct the class dependence graph and finally the system dependence graph. We elaborate each step in this chapter. Our work can be mainly divided into two steps:

- Construction of the intermediate representation
- Static backward slicing using the two pass slicing algorithm by Larsen and Harold [2]

4.1 Construction of the intermediate representation

We present some algorithms used for the construction of the intermediate representation.

4.1.1 Algorithm for construction of CFG

To construct the control flow graph we need to determine the true and false edges from every node. Control flows along the true edge if the value of the predicate evaluates to true and vice versa. Following are the steps in constructing the CDG:

Step 1: A Pending stack is taken which is initially set to NULL.
Step 2: When an expression is encountered, a node is created for the expression and it is inserted into the pending stack.
Step 3: When the next new expression is encountered, this node is popped out from the stack and an edge is created between this popped node and the node for the new expression.
Step 4: While dealing with conditional statements, after the end of the ‘if’ block, the ‘if’ node is inserted into the pending stack and if a ‘else’ is encountered, then the pending stack is saved to some temporary variable and assigned to null. And then the false edge of the corresponding ‘if’ block is inserted into it. When the ‘else’ block ends, the temporary stack and the current pending stack are merged.

4.1.2 Algorithm for computation of DDG

The data dependence graph is constructed to represent various data dependencies between the different program constructs. So, construction of the DDG basically consists of an algorithm to determine the various reaching definitions for statements in a program. The following algorithm is used to compute the reaching definition of all the nodes in a CFG:

Step 1: In a first traversal over the control flow graph, one computes the definition set of each variable that has been defined and the gen and kill sets for each statement.
Step 2: In another traversal, one computes the reaching definitions in a syntax-directed manner and inserts links from the usage nodes of variables to all its reaching definitions.

4.1.3 Algorithm for construction of CDG

To construct the CDG, we have used the approach suggested by Ferrante et al. [4]. They introduced the concept of post-dominator tree for the construction of the CDG. A post-dominator tree is equivalent to the dominator tree of the reverse CFG. The algorithm suggested by Lengauer and Tarjan [1] can be used for the construction of the dominator tree. The steps used in the construction of the CDG are as follows:

Step 1: Reverse the control flow graph
Step 2: Construct the dominator tree based on the algorithm suggested by Lengauer and Tarjan [1]
Step 3: Construct the control dependence form the post-dominator tree and the CFG.

4.1.4 Algorithm for construction of ClDG

For the construction of the ClDG, we use an algorithm proposed by Larsen and Harrold [2]:

Step 1: In a ClDG, a node for the class entry vertex is constructed.
Step 2: All members of the class are identified, nodes are constructed for them and they are connected with the class entry vertex.
Step 3: In case of methods belonging to the class, add the class members as formal-in parameters that are used inside the method.

4.1.5 Algorithm for construction of SDG

A program dependence graph cannot handle inter-procedural calls. Hence, it is extended to construct an SDG to facilitate inter-procedural calls. For this, we implement the parameter-in, parameter-out, summary and call edges and formal-in, formal-out, actual-in and actual-out vertices so as to form an SDG. When a function is defined, the formal parameters passed to the function are stored as its formal-in parameters. Similarly when a function is called, the arguments passed constitute its actual-in parameters. The formal-out vertices are added to the system dependence graph for each formal reference parameter that is modified by the method. For every corresponding formal-out vertex there is an actual-out vertex.

The pseudo-code of our algorithm to construct the SDG parameters is as follows:

Algorithm: construct_SDG

Input: formal_in_list, event

Global: global_formal_out_list, final_formal_out_list, affecting_me_list

Output: system dependence graph with formal out and summary edges

```
if (event = function start)
    while ( event = new variable encountered )
        node <- new node
        if  ( type[var] = modified_var )
            create _node ( var )
            add ( node, global_formal_out_list )
        end if
     end while
     while ( node1 = extract_node (formal_in_list) )
          while ( node2 = extract_node ( global_formal_out_list )
               if ( node1.var = node2.var )
                    add ( node2, final_formal_out_list )
               end if
          end while
      end while
end if
if (event = function start)
    while ( event = new variable encountered )
        node <- new node
```

```
        if  ( type[var] = modified_var )
            node1 = create_node ( var )
            node1.affecting_me_list = affecting_me_list
        else
            if ( type[var] = used_var )
                add ( var, affecting_me_list )
            end if
        end if
    end while
    while ( node1 = extract_node (formal_in_list) )
        while ( node2 = extract_node ( global_formal_out_list )
            if ( node1.affecting_me_list = node2.affecting_me_list )
                add ( node2, affecting_me_list )
            end if
        end while
     end while
end if
```

4.2 Two-pass backward slicing algorithm

Every slicing algorithm uses a slicing criterion which is generally represented by <S, V>, where S is the statement and V may be a variable or a function call. If V is a variable, then V must be defined at that statement, else it must be called at S. The static backward slice of a statement in a program is then calculated using a two pass graph reachability algorithm proposed by Horwitz [19]. The first phase of the algorithm identifies vertices that reach S which may be in the same procedure or in another procedure that calls that procedure. We ignore those vertices that reach S through the procedure call in the first phase and identify them in phase 2. Here, we implement the algorithm proposed by Larsen and Harrold [2].

Pass 1: In the first pass of the slicing algorithm, we traverse through all the edges except the parameter-out edges. The summary edges are used to slice across all the call vertices, because summary edges represent the transitive dependencies between the actual-in and actual-out vertices.

Pass 2: In the second pass of the algorithm, we traverse through the parameter-out edges to descend into the different functions or procedures.

Chapter 5

Implementation and results

In this chapter we present the details of our implementation and the results that include different screenshots of the construction of the intermediate representation of the input program and also the average time taken for different sample programs inputted to two pass backward slicing algorithm to compute the static backward slice of a statement. We also present an overview of the tools and some of the data structures we have used.

5.1 Tools used

We use the following tools in order to implement and code the programs and finally to get the intermediate representation.

- Eclipse
- ANTLR
- Graphviz

5.1.1Eclipse

Eclipse is a multi-language software development environment comprising an integrated development environment and an extensible plug-in system. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages as well, including C, C++, COBOL, Python, Perl, PHP, and others. The IDE is often called Eclipse ADT for Ada, Eclipse CDT for C, Eclipse JDT for Java and Eclipse PDT for PHP. The most important feature of Eclipse is its plug-in system. We can integrate different plug-in tools into the eclipse environment and can be used them in the applications. Examples of a few such plug-ins are ANTLR, C/C++ development tool, etc. It is also simple to use as we need not install it. The only thing that we have to do it is to download Eclipse and run the eclipse.exe file. We prefer the language C++ for coding because of the support for object oriented features (which C does not have) and for dynamic memory allocation. Moreover, it is faster than Java. But since CDT is not a C/C++ compiler, we have to

download and install MinGW, a C/C++ compiler which is available as a plug-in for Eclipse to compile our code. Eclipse detects and configures MinGW automatically [5].

5.1.2 ANTLR

ANTLR stands for “ANother Tool for Language Recognition”. It is a top-down parser generator that uses LL (*) parsing. ANTLR takes as input a grammar that specifies a language and generates as output, source code for a recognizer for that language. ANTLR supports code in C, Java, Python, and C#. It provides a single consistent notation for specifying lexers, parsers and tree parsers. This is in contrast with other parser/lexer generators and adds greatly to the tool's ease of use.

ANTLR mainly produces two files a lexer and a parser file. These files may be a java or C or C++ files. By default ANTLR produces java files. To produce the output files in C++ we need to add the line language = "Cpp"; in options of the grammar file [5].

5.1.3 Graphviz

Graphviz is a tool that can pictorially represent a graph. We have used this tool in our project to visualize our final output in a better way i.e. in the form of a pictorial graph instead of a adjacency matrix or adjacency list. The output of the program is converted into a form that is recognizable by graphviz and is written into an output file in the same format. Graphviz reads from the output file in order to generate the graph that the user can visualize [5].

5.2 Data structures used

In our implementation, we start the construction of the control dependence graph from an input control flow graph and then proceed to construct the program dependence graph, system dependence graph and class dependence graphs. Hence, the data structures used in the construction of these graphs are basically those which have been used to implement the control flow graph.

In a control flow graph every program construct has been represented in the form of a node. Every node has some common attributes like the node number, node type, node parent, etc. Also, different program constructs have specific attributes of their own. For example, function call may have specific attributes like actual-in and actual-out parameters, pointers to

the called function, etc. We make a class node which has all these common attributes. The class node has been represented below:

```
class node {
        public:
          int node_no;
          int node_type;
          node *next;
          node *parent;
};
```

Figure 5.1: Class node

In the figure 5.1, the attribute node_no stores the serial count of the program construct as encountered in the control flow graph and the node_type store the type of the program construct (like function, class, global variable, local variable, etc). For implementation of the program construct, we include certain additional attributes as in the following example.

```
Class func_call : private base node {
public:
char *name;
node *called_fun;
node *arg;
};
```

Figure 5.2: Class func_call

In figure 5.2 the attributes called_fun represents the pointer to the function definition for the corresponding function call. The attribute arg represents the actual argument passed through the function call. Similarly, there are other classes like class, formal_param, local_var, etc.

5.3 Screenshots of implementation

Different screenshots of the implementation of the various dependence graphs have been shown taking a few examples.

```
main() {
        int a;
        int b;
        a =10;
        if (a>5)
              b = a + a;
        cout << b;
        }
```

Figure 5.3: A sample program

```
main() {
        int a, b, c;
        a= 10;
        b = 1;
        if (a>5) {
              while (a<20) {
                    a = a+1;
                    b = b * a;
                    }
              }
              cout << b;
        }
```

Figure 5.4: A sample program

```
main() {
        int a = 10, b;
        b = add(a,3);
 }
  int add(int x, int y) {
         x = x + y;
          return x;
  }
```

Figure 5.5: A sample program

```
class A{
      int add(int x, int y) {
           x = x + y;
            return x;
       }
       void display() {
            cout << x;
       }
};
```

Figure 5.6: A sample program

5.3.1 Implementation of CFG

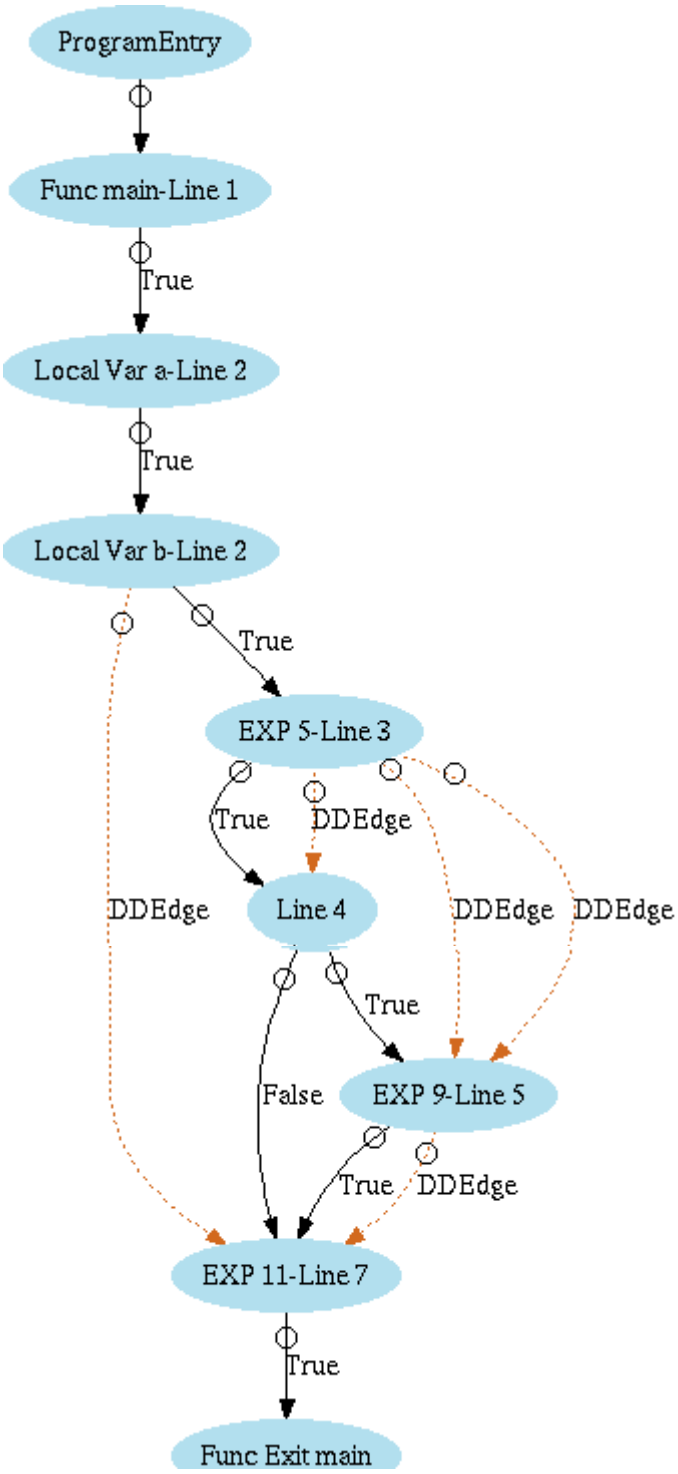

Figure 5.7: Screenshot of implementation of program in Figure 5.3

5.3.2 Implementation of PDG

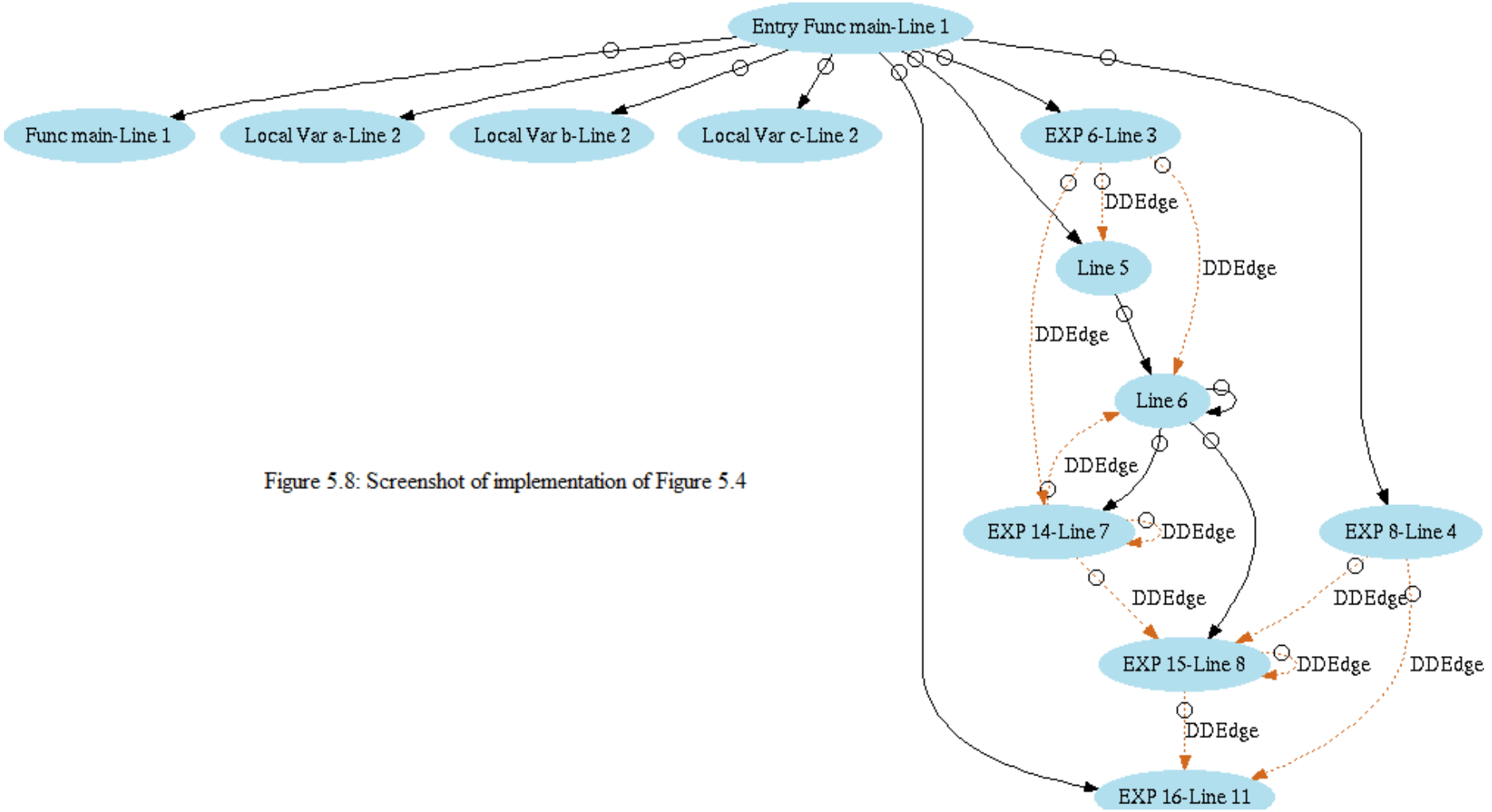

Figure 5.8: Screenshot of implementation of Figure 5.4

5.3.3 Implementation of the SDG

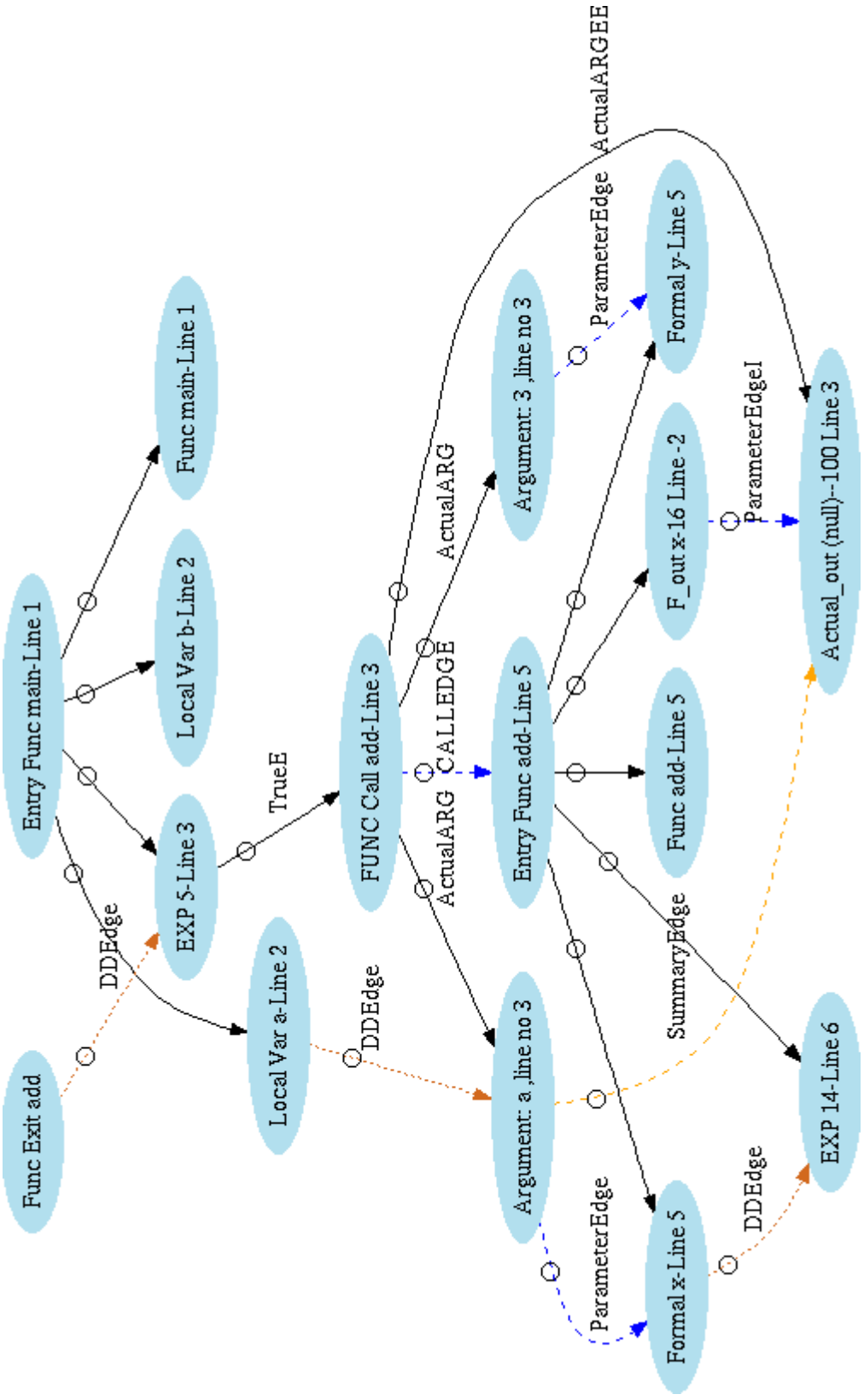

Figure 5.9: Screenshot of implementation of SDG of Figure 5.5

5.3.4 Implementation of the ClDG

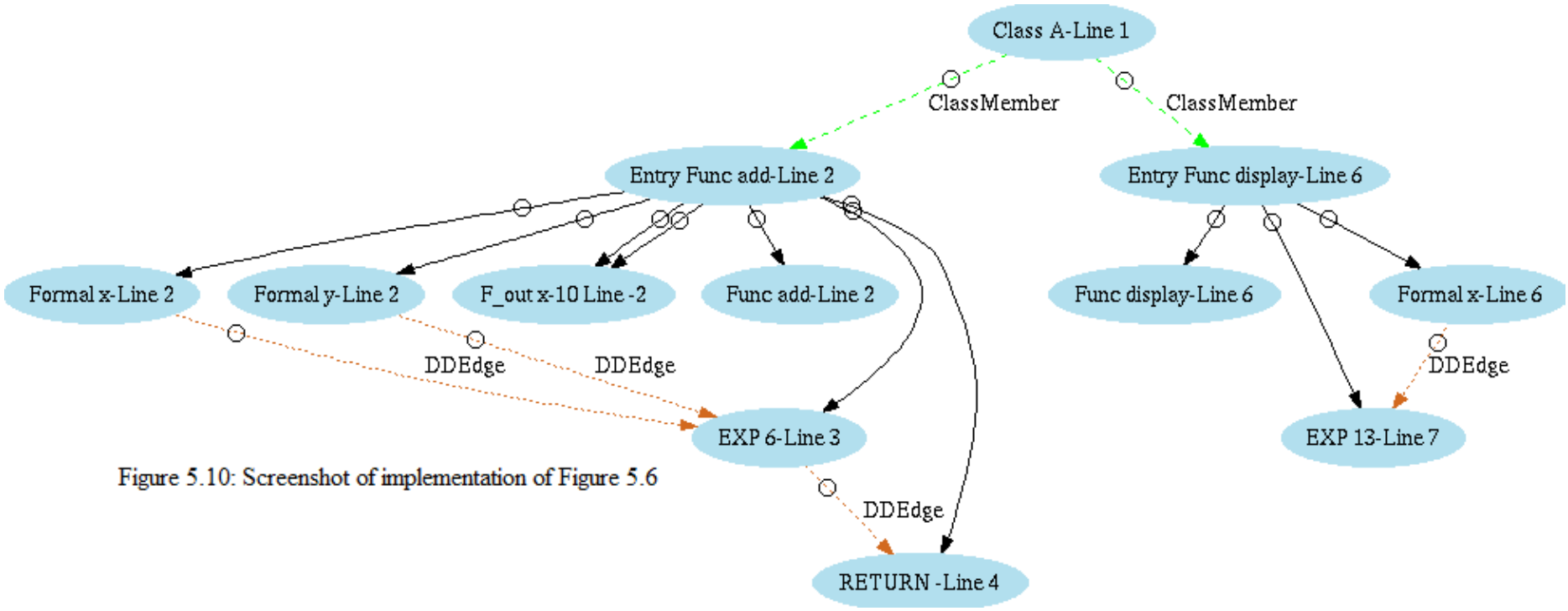

Figure 5.10: Screenshot of implementation of Figure 5.6

5.4 Result of graph reachability algorithm

Serial No.	Number of lines in the program	Average slicing time (micro seconds)
1	10	470.2
2	20	481.2
3	50	563.6
4	70	590.4
5	100	1047.2
6	120	1250.8
7	140	1277.5
8	170	1335.2
9	200	1750.5

Figure 5.11: Table showing average slicing Vs number of lines

5.5 Graph of the implementation

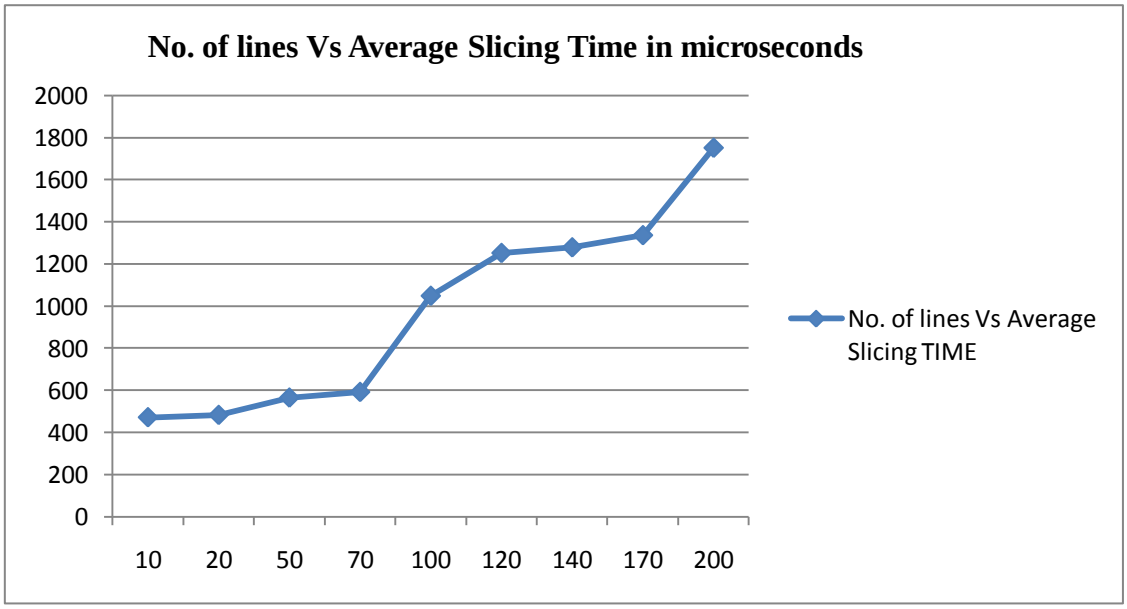

Figure 5.12: Graph of the slicing algorithm

From the graph we can conclude that as the size of the input program increases, the average slicing time also increases. There may be even a sudden increase in the average slicing time as in our case, because of the complexity of the input program.

Chapter 6

Conclusion and future work

6.1 Conclusion

We implemented the structure of a Program Dependence Graph using a graphical user interface. After that we implemented a Class Dependence Graph and a System Dependence Graph as proposed by Horwitz [19] to handle features of a class and inter-procedural calls. Once an intermediate representation is obtained, we implemented the two-pass graph reachability algorithm proposed by Larsen and Harrold [2] to compute a static backward slice for a sample input program and obtained the average slicing times for different sample input programs.

6.2 Future work

The class dependence graph constructed only models how a class entry node is connected to its members. This can further be extended to handle other object-oriented features like polymorphism, inheritance and exception handling. Our work can also be used to handle aspect oriented programs. The intermediate representation can be used to select test cases in regression testing, for software debugging and in many other applications.

References

[1] Lengauer T. and Tarjan R. E., A fast algorithm for finding dominators in a flow graph, ACM Transactions on Programming Languages and Systems, Vol. 1, No. 1, July 1979.

[2] Larsen L., Harrold M. J., Slicing Object-Oriented Software, Proceedings of the 18th international conference on Software Engineering, Pages 495-505, 1996.

[3] Biswas S., Mall R., Sathpathy M., Sukumaran S., A model-based test selection approach for embedded applications, ACM SIGSOFT Software Engineering Notes, Pages 1-9, May 15 2009.

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

[5] Mittal M., Implementation of A Model for Application to Regression Test Case Selection, Mtech thesis, Indian Institute of Technology, Kharagpur, May, 2009.

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

[7] Xu B., Qian J., Zhang X., Wu Z., and Chen L., A Brief Survey of program slicing, ACM SIGSOFT Software Engineering Notes 30, Pages 1-36, February 2005.

[8] Weiser M., Programmers use slices when debugging, Communication of ACM25, Pages 446-452, July 1982.

[9] Zhang X., Gupta R., and Zhang Y., Efficient forward computation of dynamic slices using Reduced ordered binary decision diagrams, International conference of Software Engineering, 2004.

[10] Agrawal H., DeMillo R, A., and Spafford E. H., Debugging with dynamic slicing and

Backtracking, Software Practice and Experience 23, Pages 589-616, June 1993.

[11] Dhamdhare D.M., Gururaja K., and Ganu P. G., A compact education history for dynamic slicing, Information Processing Letters 85, Pages 145-152, 2003.

[12] Korel B., and Rilling J., Dynamic Program Slicing Methods, Information and Software Technology 40, Pages 155-163, 1998.

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

[14] Song Y., and Huynh D., Forward Dynamic Object-Oriented Slicing, Application Specific Systems and Software Engineering and Technology(ASSET'99), IEEE CS Press, 1999.

[15] Steindl C., Program Slicing for Object Oriented Programming Languages, PhD thesis, Johannes Kepler University, Linz, 1999.

[16] Trew T., What Design Policies Must Testers Demand from Product Line Architectures? International Workshop on Software Product Line Testing (SPLIT), Pages 51-57, 2004.

[17] J.Jenny Li, Prioritize code for testing to improve code Coverage of complex software, in proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE05), Pages 75-84.

[18] Ottenstein K. J., Ottenstein L. M., The program dependence graph in a software development environment, Proceedings of ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pages 85-97, July 1995.

[19] Horwitz S., Reps T., Binkley D., Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems, Pages 26-60, January 1990.

[20] Dallal J. A., An Efficient Algorithm for Computing all Program Forward Static Slices, World Academic of Science, Engineering and Technology 16, 2006