Slicing of Aspect-Oriented Software & Its Application to Software Refactoring

Jagannath Singh

Department of Computer Science and Engineering
National Institute of Technology Rourkela

Slicing of Aspect-Oriented Software & Its Application to Software Refactoring

Dissertation submitted in partial fulfillment

of the requirements of the degree of

Doctor of Philosophy

in

Computer Science and Engineering

by

Jagannath Singh

(Roll Number: 512CS601)

based on research carried out

under the supervision of

Prof. D. P. Mohapatra

and

Prof. P. M. Khilar

December, 2016

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Department of Computer Science and Engineering
National Institute of Technology Rourkela

December 30, 2016

Certificate of Examination

Roll Number: *512CS601*
Name: *Jagannath Singh*
Title of Dissertation: *Slicing of Aspect-Oriented Software & Its Application to Software Refactoring*

We the below signed, after checking the dissertation mentioned above and the official record book (s) of the student, hereby state our approval of the dissertation submitted in partial fulfillment of the requirements of the degree of *Doctor of Philosophy* in *Computer Science and Engineering* at *National Institute of Technology Rourkela*. We are satisfied with the volume, quality, correctness, and originality of the work.

P. M. Khilar Co-Supervisor	D. P. Mohapatra Principal Supervisor
Prof. S. Goplakrishna Member, DSC	Prof. S. Chinara Member, DSC
Prof. S. K. Jena Member, DSC	Prof. A. Ananda Rao External Examiner
Prof. S. K. Rath Chairperson, DSC	Prof. D. P. Mohapatra Head of the Department

Department of Computer Science and Engineering
National Institute of Technology Rourkela

Prof. D. P. Mohapatra
Associate Professor & Head

Prof. P. M. Khilar
Assistant Professor

December 30, 2016

Supervisors' Certificate

This is to certify that the work presented in the dissertation entitled *Slicing of Aspect-Oriented Software & Its Application to Software Refactoring* submitted by *Jagannath Singh*, Roll Number 512CS601, is a record of original research carried out by him under our supervision and guidance in partial fulfillment of the requirements of the degree of *Doctor of Philosophy* in *Computer Science and Engineering*. Neither this dissertation nor any part of it has been submitted earlier for any degree or diploma to any institute or university in India or abroad.

P. M. Khilar
Assistant Professor

D. P. Mohapatra
Associate Professor & Head

Dedication

This thesis is dedicated to my father and mother

Signature

Jagannath Singh

Declaration of Originality

I, *Jagannath Singh*, Roll Number *512CS601* hereby declare that this dissertation entitled *Slicing of Aspect-Oriented Software & Its Application to Software Refactoring* presents my original work carried out as a doctoral student of NIT Rourkela and, to the best of my knowledge, contains no material previously published or written by another person, nor any material presented by me for the award of any degree or diploma of NIT Rourkela or any other institution. Any contribution made to this research by others, with whom I have worked at NIT Rourkela or elsewhere, is explicitly acknowledged in the dissertation. Works of other authors cited in this dissertation have been duly acknowledged under the sections ''Reference'' or ''Bibliography''. I have also submitted my original research records to the scrutiny committee for evaluation of my dissertation.

I am fully aware that in case of any non-compliance detected in future, the Senate of NIT Rourkela may withdraw the degree awarded to me on the basis of the present dissertation.

December 30, 2016
NIT Rourkela

Jagannath Singh

Acknowledgment

First, and foremost I would like to thank my supervisors Prof. D. P. Mohapatra and Prof. P. M. Khilar for giving me the guidance, encouragement, counsel throughout my research and painstakingly reading my reports. Without their invaluable advice and assistance it would not have been possible for me to complete this thesis.

I take this opportunity to express my sincere thanks to Prof. S. K. Rath, for his kind support as and when required. I also thank Prof. S. K Jena, Prof. S. Chinara, and Prof. S. Gopalkrishna for serving on my Doctoral Scrutiny Committee and providing valuable feedback and insightful comments.

I am grateful to all the faculty members of the CSE Department for their many helpful comments and constant encouragement. I wish to thank the Software Laboratory staff and all the secretarial staff of the CSE Department for their sympathetic cooperation.

I wish to thank Dr. S. Panda, a senior and friend for helping me with his advices and encouragement. I also thank all my research colleagues, specially Sangharatna Godbolye, for their encouragement and help in several forms.

I gratefully acknowledge the support provided by the National Institute of Technology (NIT), Rourkela.

I am grateful to my family for their inspiration and constant encouragement, especially my elder brother *Krishna Singh*. I wish to thank my son, *Sarthak*, for his sacrifice during these years; he missed me a lot. Without the constant support and encouragement of my wife, *Nitu*, I could hardly have completed this work. Her unending patience, encouragement and understanding have made it all possible, and meaningful. I want to dedicate this thesis to my *Family*.

December 30, 2016 *Jagannath Singh*
NIT Rourkela Roll Number: 512CS601

Abstract

This thesis first presents some program slicing techniques for Aspect-Oriented Programs (AOPs) and then presents a technique for refactoring of software using the proposed slicing technique. Main aim of all the proposed slicing algorithms in this thesis is to compute accurate and precise dynamic slices of AOPs.

In order to compute the slices of aspect-oriented programs, first we extend the System Dependence Graph (SDG) for Object-Oriented Programs (OOPs) to handle AOPs. We have named the extended SDG *Extended Aspect-Oriented System Dependence Graph* (EAOSDG). The EAOSDG successfully represents different aspect- oriented features such as class representation, method invocation, inheritance, aspect declaration, point-cuts, advices etc. The EAOSDG of an aspect-oriented program consists of System Dependence Graph (SDG) for the non-aspect code, a group of Aspect-Oriented Dependence Graphs (ADGs) for aspect code and some additional dependence edges that are used to connect the SDG of the non-aspect code (base code) to ADG of the aspect code. Then, we propose an extended two-phase algorithm to compute the static slices of AOPs, using the proposed EAOSDG. Subsequently, we present a context-sensitive slicing algorithm to compute the dynamic slices of AOPs, using the proposed EAOSDG. The context-sensitivity makes the computed slice more precise and accurate. We have developed a slicer to implement our proposed algorithms. We have compared the performance of extended two-phase algorithm and context-sensitive algorithm, in terms of the average slice extraction time. We have considered five open source projects for comparison of slicing algorithms. We have observed that the context-sensitive algorithm computes the slices faster than the extended-two phase algorithm.

Next, we extends our intermediate representation (EAOSDG) to be able to represent *concurrent aspect-oriented programs*. We have named this intermediate representation Multithreaded Aspect-Oriented Dependence Graph (MAODG). Our MAODG correcly represents the concurrency dependencies in concurrent AOPs. Then, we extend our context-sensitive dynamic slicing technique to handle concurrent AOPs having multiple threads. We have named our algorithm *Context-Sensitive Concurrent Aspect* (CSCA) slicing algorithm. Due to the presence of inter-thread synchronization and communication dependencies, some control and data flows in the threads become interdependent. This interdependency causes difficulty in finding accurate slices of concurrent AOPs. Our algorithm takes the MAODG of the concurrent AOP and a slicing criterion as input and computes the dynamic slice for the given concurrent AOP. We have developed a slicer *Concurrent AspectJ* slicer to implement our proposed CSCA algorithm. We have compared CSCA algorithm with two other existing algorithms using five case studies. The experiment shows that, our proposed CSCA algorithm computes precise slices in less time as compared to the other two existing algorithms.

Further, we propose an approach for dynamic slicing of *distributed* AOPs. We first represent distributed aspect-oriented program using dependence based intermediate representation which we have named *Distributed Aspect Dependence Graph* (DADG). Based on the DADG, we present a slicing algorithm *Parallel Context-sensitive Dynamic Slicing* (PCDS) algorithm for distributed AOPs. We introduce parallelism in our algorithm to make slice computation faster. We have developed a tool called *D-AspectJ* slicer to implement the PCDS algorithm. The proposed slicing algorithm is compared with two other existing algorithms using seven case studies. The experimentation shows that our proposed PCDS algorithm generates smaller slices in less time as compared to the other two existing algorithms.

Finally, we present a technique for software refactoring using program slicing. We use slice-based cohesion metrics to identify the target methods of a software that require refactoring. After identifying the target methods, we use program slicing to divide the target method into two parts. Then, we use the concept of *aspects* to alter the code structure in a manner that does not change the external behavior of the original module. We have implemented our proposed refactoring technique and evaluated its effectiveness through eleven case studies. We have also evaluated the effect of our proposed refactoring technique based on an open source code coverage tool *EclEmma*.

***Keywords*: *Program Slicing*; *Aspect-Oriented Programming*; *Software Refactoring*; *Concurrent Programming*; *System Dependence Graph*.**

Contents

List of Figures

List of Tables

List of Abbreviations

ADG	Advice Dependence Graph
ADGs	Aspect-Oriented Dependence Graphs
ASDG	Aspect-Oriented System Dependence Graph
AO	Aspect-Oriented
AOP	Aspect-Oriented Programming
AOPs	Aspect-Oriented Programs
CAOPs	Concurrent Aspect-Oriented Programs
CASDG	Concurrent Aspect-oriented System Dependence Graph
CDA	Control Dependence Algorithm
CFG	Control Flow Graph
CGCA	Contradictory Graph Coloring Algorithm
CI	Context-Insensitive
CPDG	Concurrent Program Dependence Graphs
CSCA	Context-Sensitive Concurrent Aspect
DADG	Distributed Aspect Dependence Graph
DDDA	Dynamic Data Dependence Algorithm
DDDG	Distributed Dynamic Dependence Graph
DDG	Distributed Dependence Graph
DPDG	Distributed Program Dependence Graph
EAOSDG	Extended Aspect-Oriented System Dependence Graph
IBS	Internet Banking System
IDG	Introduction Dependence Graph
MAODG	Multithreaded Aspect-Oriented Dependence Graph
MDGs	Method Dependence Graphs
NMDS	Node Marking Dynamic Slicing
OO	Object-Oriented
OOPs	Object-Oriented Programs
ORBS	Observation-Based slicing
PADS	Parallel Aspect-oriented Dynamic Slicing
PCDS	Parallel Context-sensitive Dynamic Slicing
PDG	Program Dependence Graph

SDG	System Dependence Graph
SDN	System Dependence Net
TBDS	Trace file Based Dynamic Slicing
tIPDG	threaded Inter procedural Program Dependency Graph

Chapter 1

Introduction

In the present day world, most of the activities are controlled by software or software is helping the routine activities to be more effective. Starting from e-mail to artificial satellite launching, software is developed to handle a wide range of activities. Every task of these applications is controlled by software. The main concern of present day software development organizations is to deliver more reliable and maintainable software. The increasing program complexity generates obstacles in the development of such software. Therefore, researchers have explored many program analysis techniques that help study the behaviour of the programs and reduce the complexity of a program. *Program slicing* is one of such program analysis techniques. Program slicing extracts the statements of a program that may affect or may be affected by a particular point in a program [1]. The collection of such bunch of extracted statements is called a *slice* and the point of interest at which we find the slice is called *slicing criterion*. Typically, a slicing criterion consists of a pair $< s, V >$, where s is the statement number and V is the set of variables present in that statement. The important applications of program slicing include various software engineering activities such as program understanding, testing, debugging, program maintenance, complexity measurement using software metrics, software security, and software refacoring etc.

Program slicing technique was first proposed by Mark Weiser [1] in 1981. According to Weiser, program slicing is a technique to generate a part of the program with respect to a given slicing criterion by deleting zero or more irrelevant statements from the original program. Also, the generated part called as slice must be an executable slice. The slice is computed such that it generates the same output as the original program would have generated with the same input. The slicing technique proposed by Weiser [1] is called static backward slicing. It is called static because it is computing slices for all possible inputs to the given program. Weiser proposed the algorithm for computing the slices by exploring the reachability of the slicing criterion. Initially, program slicing was proposed for procedural programs that basically contain only procedures. After Weiser, several other researchers have developed various program slicing techniques such as, static forward slicing [2], dynamic slicing [3, 4], data-flow equation based slicing [5], backward slicing [6], conditional slicing [7], etc. As the programming practice changes from procedural programming to Object-Oriented Programming (OOP) , all the above mentioned slicing techniques where found insufficient to address the features of OOP. The features like *abstraction*, *inheritance*, *overriding*, *instantiate*, *polymorphism* etc. cannot be handled by the procedural slicing techniques. Therefore new slicing techniques have been developed by several researchers [8–10] to compute slices of Object-Oriented Programs (OOPs).

As the popularity of OOP increased, people started finding some drawbacks in the OOP implementation. One of them is the presence of cross-cutting concerns. Cross-cutting concerns are that parts of the program that are scattered across multiple modules of the program and are also tangled with other basic modules. While OOP is the most common methodology used to manage core concerns, it is not sufficient for managing the cross-cutting concerns. A new methodology was evolved, called *Aspect-Oriented Programming (AOP)*, which can effectively handle the cross-cutting concerns of a software. Along with AOP comes the new features like *pointcuts* and *joinpoints*, which cannot be handelled by the Oject-Oriented (OO) slicing algorithms. Zhao et al. [11] had proposed a slicing algorithm for AOPs. But, the algorithm of Zhao is a static type. According to Korel and Laski [12], the dynamic slices are more useful for interactive applications such as designing and testing and they are smaller in size. Therefore, there is a need for developing more efficient dynamic slicing technique for AOPs which can compute accurate and precise slices.

The introduction of concurrency in a program increases the throughput of the system. In AOP, concurrency is achieved through using *threads*. When an Aspect-Oriented (AO) program does not contains any thread, then each time we execute the program, it will produce the same result for the same input. It means that the output of a program without threads can be predicted, if we know the inputs. But, the presence of threads in a program make it unpredictable. It may produce different results for the same input when the program containing threads executes in different runs. Computation of dynamic slice of any given program depends heavily on the program execution. The non-deterministic nature of a concurrent program makes it very difficult to understand it's execution and hence presents obstacles to compute the slices.

Similarly, the distributed AOPs are also harder to understand and analyze. Execution sequence in a distributed program depends on the sequence of data exchange between the distributed programs. But, the sequence of data exchange between the component programs is not consistent, because all the component programs run on independent computers connect through one network. Hence, dynamic slice generation of the distributed programs is very difficult. However, most of the research work in the program slicing area have focused attention on sequential programs. To the best of our knowledge research reports addressing slicing of *concurrent* and *distributed* programs are scarce in literature [13, 14].

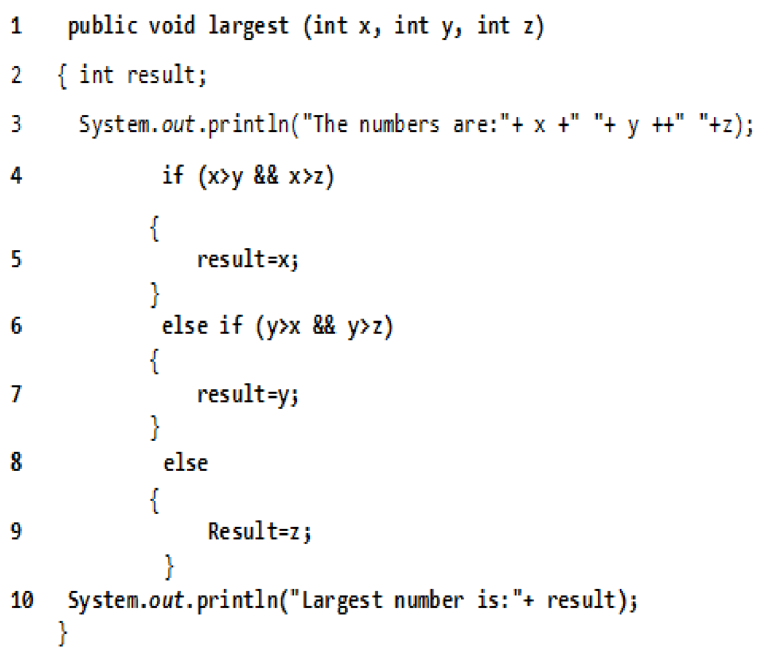
A major goal of any dynamic slicing technique is *preciseness*, since the results are normally used during interactive applications such as program debugging. Preciseness is especially an important concern in slicing aspect-oriented programs, since the size of practical aspect-oriented programs is often very large. The slice computation time of an imprecise dynamic slicer may be unacceptably large for such programs [15]. Generally, slicing starts with the analysis of the code of the given program and then presenting the program using an intermediate representation. Most of the researchers have considered graphs as the intermediate representation. The intermediate graph representation is analyzed by using an algorithm to compute the required slices. So, the efficiency of a slicing technique depends on how efficiently an intermediate graph representation is able to represent the given program.

As already discussed, program slicing is useful in many fields of software development, such as testing, debugging, re-engineering, software refactoring etc. One of the important applications of program slicing is in the field of software refactoring. We have proposed a new software refactoring technique based on program slicing and cohesion metrics. Our experimental results show that after applying the software refactoring technique, the value of cohesion metric for the given program is increasing. When the cohesion of program modules increases, the quality of the software enhances. So, we present software refactoring as an application of our proposed slicing algorithms.

1.1 Categories of Program Slicing

Several slicing techniques were developed in the course of time between 1980 to till date. In the literature, we found that the existing slicing techniques can be classified broadly into the following categories: *backward* or *forward*, *static* or *dynamic*, *intra-procedural* or *inter-procedural*. We found that there are also some slicing techniques that are different from the above types of slicing. These types of slicing include *Context-Sensitive* slicing [16], *Simultaneous* slicing [4], *Conditioned Slicing* [7], *Amorphous* slicing [17–19], *Observation-Based* slicing [20] and some other variations of program slicing [21]. In this section, we discuss all these different types of slicing.

Static Slicing and Dynamic Slicing: In *static slicing*, all the program statements that affect the value of a variable in the slicing criterion are included into the slice. The input values provided to the program may change the program execution and the value of the variable in the slicing criterion. But, in static slicing the slice is computed *for all possible values* of the input variables. Likewise the predicates in a program can take either a true or a false value. In static slicing, we have to consider both the cases and find the slice. It is obvious that when we are considering all the input values and both true and false values of all the predicates, the size of a static slice is likely to be very large. For a large and complex program, the computed static slice will be of large size. Hence, large sized slices are again complex and hard to comprehend. So, the objective of slicing may not be achieved. Consider the Java method *largest()* given in Figure 1.1. The static slice with respect to the slicing criterion $< 10, result >$ is the set of statements $\{1, 4, 5, 6, 7, 8, 9, 10\}$.



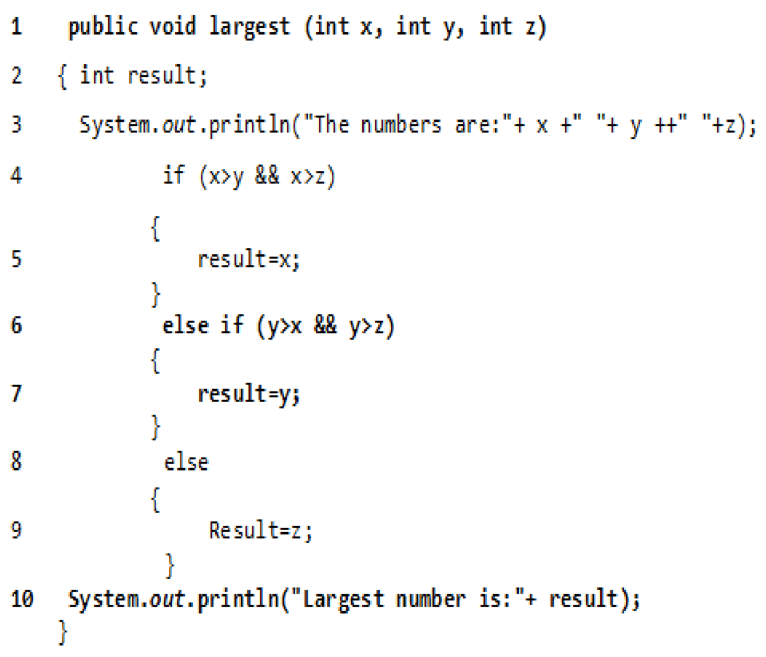
```
1    public void largest (int x, int y, int z)
2   { int result;
3     System.out.println("The numbers are:"+ x +" "+ y ++" "+z);
4             if (x>y && x>z)
             {
5                 result=x;
             }
6             else if (y>x && y>z)
             {
7                 result=y;
             }
8             else
             {
9                  Result=z;
              }
10   System.out.println("Largest number is:"+ result);
    }
```

Figure 1.1: A code snippet showing the static slice in bold with respect to slicing criterion <10,result>

Korel and Laski [12] introduced the concept of dynamic program slicing. In *dynamic slicing*, the slices are computed for a particular execution of the program specific to some input. The sequence of the executed statements in a particular execution is observed and the dynamic slice for that particular execution is computed. A dynamic slice with respect to slicing criterion $< s, V >$ is computed by finding the statements which are executing and also affecting the values of a variable in V at statement s. In the dynamic slicing, the values of the input variables are known and the values of predicates are also fixed. Hence, the extra statements that were included in a static slice due to the unavailability of the fixed values for input variables and predicates, are not present in a dynamic slice. For example, consider a particular execution of the Java method *largest()* with the input value $x = 5, y = 9, z = 2$ given in Figure 1.2. The dynamic slice with respect to the slicing criterion $< 10, result >$ for the particular execution of the program is {1, 6, 7, 10}.

Dynamic slicing is found more useful in complex and large programs [3]. In other words, we can state that dynamic slicing techniques compute *precise slices*. A comprehensive survey on the existing dynamic program slicing algorithms is reported in Korel and Rilling [15] and Xu et al. [22].

Backward Slicing and Forward Slicing: The impact analysis of a statement can be either forward or backward. Depending upon these program analysis, the slices are computed. So according to the direction of program analysis involved in the slicing process, the slicing techniques can be classified as either forward or backward slicing. A backward slice provides



```
1    public void largest (int x, int y, int z)
2   { int result;
3     System.out.println("The numbers are:"+ x +" "+ y ++" "+z);
4            if (x>y && x>z)
            {
5               result=x;
            }
6            else if (y>x && y>z)
            {
7               result=y;
            }
8            else
            {
9                Result=z;
             }
10   System.out.println("Largest number is:"+ result);
    }
```

Figure 1.2: A code snippet showing the dynamic slice in bold with respect to slicing criterion <10,result,(5,9,2)>

the answer to the question: *''which statements will affect the slicing criterion?''*. We start from the slicing criterion and search for the statements of the program that may directly or indirectly affect the value of a variable at slicing criterion. This type of slicing is called backward slicing, because we move from the current statement (slicing criterion) towards backward direction of the program execution [1]. For example, consider the *read()* method given in Figure 1.4. The backward slice with respect to the slicing criterion $< 7, volume >$ includes statements $\{1, 3, 5, 7\}$, as shown in bold letters in Figure 1.4.

On the other side, in *forward slicing*, we start from the slicing criterion and search for the statements that may be affected by the slicing criterion. Forward slicing follows the same direction of the program execution, and hence called forward slicing. In forward slicing, we find the statements of the program that are directly or indirectly dependent upon the slicing criterion and its variables [23, 24]. The forward slice of the example method given in Figure 1.4 with respect to slicing criterion $< 1, h >$ includes statements $\{1, 4, 5, 6, 7\}$, as shown in bold letters in Figure 1.3. A forward slice provides the answer to the question: *''which statements will be affected by the slicing criterion?''*. In this thesis, we always use backward slicing in our proposed slicing approaches.

Intra-procedural and Inter-procedural Slicing: Intra-procedural slicing computes slices within a single procedure. Calls to other procedures are either not handled at all or handled conservatively. If the program consists of more than one procedure, inter-procedural slicing can be used to derive slices that span multiple procedures [6]. For object-oriented programs,

1 public void read (int r, int h)
2 { int area, surface, volume;
3 area = pi * r*r;
4 Surface = 2 * area + 2 * pi * r * h;
5 volume = area * h;
6 System.out.println(“Surface area of cylinder:”+ surface);
7 System.out.println(“Volume of cylinder:”+ volume);
}

Figure 1.3: Forward slice of the method with respect to slicing criterion <1,h> shown in bold

1 public void read (int r, int h)
2 { int area, surface, volume;
3 area = pi * r*r;
4 Surface = 2 * area + 2 * pi * r * h;
5 volume = area * h;
6 System.out.println(“Surface area of cylinder:”+ surface);
7 System.out.println(“Volume of cylinder:”+ volume);
}

Figure 1.4: Backward slice of the method with respect to slicing criterion <7,volume> shown in bold

intra-procedural slicing is meaningless as practical object-oriented programs contain more than one method. So, for object-oriented programs, inter-procedural slicing is more useful.

Context-Sensitive Slicing The main aim of a slicing algorithm is not only to generate slices but also to generate *accurate* slices. The slicing technique suggested by Wieser computes the affected statements for a given slicing criterion by simply considering reachability of the statements. This type of slicing is called context-insensitive slicing and it does not always produce accurate slices. The slicing technique is said to be context-sensitive, if during the traversal of the intermediate graph, the call-site is preserved at the time of entry and exit of a method. It means that during graph traversal of a method call, one must return to the same node from where he/she entered into the method. For details on context-sensitive slicing, the readers may refer [16].

Simultaneous Dynamic Slicing This type of slicing uses the combination of test cases along with slicing [4]. This is called simultaneous slicing because it simultaneously compute one dynamic slice for more than one test cases applied to the program. In dynamic slicing, the slice is computed for only one input at a time. But, in simultaneous slicing, we can compute one slice for a number of input test cases. A simultaneous dynamic slice of a program P with respect to simultaneous slicing criterion C=($\{I_1$,I_2,..., $I_m\}$, S, V) is a syntatically correct and executable program P' that is obtained from P by deleting zero or more statements from P. Here, I_m represents the m^{th} input, S is the statement and V is the set of variables in the slicing criterion. For details on simultaneous dynamic slicing, the readers may refer [4].

Quasi Static Slicing It is a hybrid of static and dynamic slicing. In Quasi slicing the values of some program variables are fixed and the slices are computed by varying values of other variables [25]. This type of slicing is broadly known as conditioned slicing. Quasi-static slicing can be used in that applications in which a set of the program inputs are fixed, and the rest of the inputs are unknown. It is mainly used in debugging and program comprehension. For details on simultaneous dynamic slicing, the readers may refer [25].

Amorphous Slicing Amorphous slicing is based on preserving program semantics [17, 18]. Generally all slicing techniques are syntax preserving in nature. In these techniques, the statements are deleted from the program based on the slicing criterion, so that the syntax of the program remains the same even after slicing. But, in amorphous slicing, any program transformation technique can be used that preserves the semantics of the program with respect to the slicing criterion. The computed slices are not as large as the slices computed by other slicing techniques. The computed slices are simplified form of the program with respect to the slicing criterion. Amorphous slicing is useful in program comprehension, analysis and reuse. For details on amorphous dynamic slicing, the readers may refer [17, 18].

Observation-Based Slicing Generally all the program slicing techniques are developed based on a specific programming language and they work for the progras written in that language only. When the same slicing technique is applied to the programs developed using other programming languages, it may not work properly. The Observation-Based slicing (ORBS) is a language independent slicing technique, which is capable of computing slices for programs developed in multiple languages [20]. In ORBS, a repetitive statement deletion process is adopted and it is validated after each deletion of statement through careful observation of the program behaviour. If the sliced program after deletion of a statement behaves as the original program, then the deletion is accepted. ORBS, in comparison with dynamic slicing, is simple to construct, effective and efficient to handle programs written in different languages. For details on simultaneous dynamic slicing, the readers may refer [20].

1.2 Issues in Program Slicing

In this section, we briefly discuss some of the most important issues that must be addressed while computing dynamic slices of aspect-oriented programs.

- **Intermediate Representation:** Before computing the dynamic slices of an AOP, first an intermediate representation is required. The given program is transformed using the intermediate representation and then a suitable dynamic slicing algorithm can be applied upon it to compute the slices. So, slicing of any AOP is dependent on the intermediate representation. Hence, the intermediate representation must be efficient to represent correctly all the features of AOP such as aspects, pointcuts, advices and introduction.

- **Memory Requirement:** As the size of software is huge, therefore the memory requirement for both the *intermediate representation* and the *dynamic slicing algorithm* should be as small as possible. If the intermediate representation requires more memory to represent an AOP, then the stored data will run out of memory. We have designed our intermediate representations such that they consume little memory. Also, we have shown that the space complexity of our proposed dynamic slicing algorithms is less than the existing ones.

- **Time Requirement:** The time required for computing slices by any dynamic slicing algorithm should be as little as possible, so that it's application can be availed in debugging and software testing. Otherwise, the response time will be too large. We have imparted more attention towards decreasing the slice computation time in all our proposed algorithms. For each dynamic slicing algorithm, we have shown that of our dynamic slicing algorithm is more time efficient than the existing ones.

- **Correctness:** The dynamic slicing algorithm should compute *correct* dynamic slices with respect to any given *slicing criterion*. A slice is said to be *correct* if it contains *all* the statements that affect the slicing criterion. We prove that each of our proposed dynamic slicing algorithm computes *correct* dynamic slices with respect to any given *slicing criterion*.

- **Scalability:** The dynamic slicing algorithms should be developed in such a way that the algorithms can easily be extended to handle large scale programs as the sizes of practical aspect-oriented programs are very large. Our dynamic slicing algorithms can be easily extended to handle large and complex programs.

- **Preciseness:** Only computing the slices is not useful in applications like debugging and testing. A slice must be accurate and precise. A dynamic slice is said to be precise, if it is an executable slice and it contains only that statements that affect the value of the variable at a program point for execution. Generally a precise slice is smaller in size.

1.3 Motivation for Our Work

One of the primary purpose of program slicing is to compute slices that can further be used in debugging [23, 26]. A program slicing technique should compute correct and precise slices so that it can be used to produce efficient results in debugging and testing. Much of the literature on program slicing is concerned with procedural and Object-Oriented software [10]. But, we found in the literature survey that there is a scarcity of papers on slicing of aspect-oriented programs [27].

The existing program slicing techniques for procedural and object-oriented programs are designed to handle the properties of procedural and OO programs, like procedures, classes, objects, methods, inheritance, polymorphism etc. In AOP, there are some extra features such as aspects, pointcuts, advices etc. which cannot be handled with these existing slicing algorithms. Hence, the need of special slicing techniques explicitly for AOPs arises. There are some researchers who have developed slicing techniques for AOPs [28, 29], but these works are not much efficient to handle all the properties of AOPs.

Multithreading is very useful in real-time programming and parallel computing. A multithreaded program runs faster than a single threaded program [30]. When a program is implemented using multithreading, the independent parts of the program can run concurrently. When the concurrency mechanism like thread is embedded with AOPs, then they are called concurrent AOPs (CAOPs). But, there are some challenges in the implementation of CAOPs such as debugging, testing, synchronization among threads, and difficulty in porting the existing code to concurrent code. The CAOPs are so complex that looking at the CAOP, it is very difficult to identify the control flow and data flow in the program. Hence, the complexity of CAOP puts obstacles in program comprehensibility, debugging, and testing. Program slicing is an analysis and transformation technique that reduces the complexity of a program. Research reports dealing with slicing of CAOPs are scarce in the literature [14]. So, there is a imperative necessity to develop suitable intermediate representations and efficient dynamic slicing algorithms for CAOPs.

When a AOP contains many component programs that can be executed in a distributed environment, it is called a distributed AOP. In distributed AOPs, the component programs execute in different computers connected to a common network. The component programs of a distributed AOP communicate through message passing. Message passing is an overhead for any distributed system, because it is very hard to know the sequence of message communications between these component programs. As a result, it is very hard to understand a distributed AOP. If a program is hard to understand, then it will be very difficult to test it and debug the faults in the program. Hence there is a necessity of a program analysis tool that can reduce the complexity of a given distributed AOP and help understand better. The literature in the field of slicing of distributed AOPs is very scarce [13] and there is a need to develop efficient and dynamic slicing techniques to handle the distributed AOPs.

Only computing slices is not useful in debugging and testing, but the computed slices must be accurate and precise. To compute more precise slices, the slicing algorithm must consider the context-sensitivity. Context-sensitive slicing algorithms compute more precise slices for complex programs. The concurrent and distributed AOPs are very complex programs and the existing slicing algorithms compute little precise slices for these programs. By introducing explicit context-sensitive feature into the dynamic slicing algorithm, the efficiency of slicing algorithm can increase.

There are many applications of program slicing, like debugging, testing, reverse engineering, etc. One of the most important and useful application of program slicing is software refactoring. In refactoring of a program, we change the structure of the more complex modules of the program, so that the overall complexity of a given program reduces. But, refactoring of the whole software is a tedious task in terms of time and cost involved in it [31]. So, instead of going for refactoring of all the modules of the software, we have to refactor only that modules which need refactoring. To identify that modules which need refactoring, *slice-based cohesion metrics* [32] can be used. We can compute the cohesion of each module in the software and then check their cohesion metrics values. If some module's cohesion metric value is less than the admissible threshold value, then we need to refactor that module. Program slicing can be used in refactor of the target modules.

Based on the above discussions, the main motivations of the thesis are listed below:

- Very little work has been done in the field of slicing of AOPs. Hence, there is a need of an accurate and precise dynamic slicing algorithm for AOPs. For this, first we have to develop a suitable intermediate representation and then based on the intermediate representation we have to design a slicing algorithm.

- Also, there is a scarcity of slicing techniques for concurrent AOPs. The existing intermediate representations are not efficient to represent all the features of a concurrent AOP. Hence, it is necessary to develop a suitable intermediate representation of concurrent AOPs and to design a dynamic slicing algorithm for concurrent AOPs.
- Slicing of distributed AOPs is not much explored and hence there is a need of developing efficient dynamic program slicing technique for distributed AOPs.
- Computed slices are useful when they are accurate and precise. Context-sensitivity is the property of program slicing algorithm which ensures the accuracy of computed slices.
- The program slicing technique must be implemented to compare the correctness and efficiency with other existing program slicing techniques. A dedicated tool is required, that can generate appropriate intermediate graph for different types of AOPs. Also, using the tool, we can implement our proposed algorithm and other existing slicing algorithms to produce a comparison of accuracy and effectiveness.
- There are many applications of program slicing, one of them is software refactoring. But, there is very few work found in literature which uses program slicing for refactoring of the existing programs. So, there is a need to develop an efficient algorithm for software refactoring using program slicing.

1.4 Objectives of Our Work

We aim at developing efficient and accurate dynamic slicing algorithms for AOPs, CAOPs and distributed AOPs. Also, we want to propose some software refactoring techniques, such that our proposed dynamic slicing algorithms can be used. To address these broad objectives, we identify the following goals:

- We want to develop a dynamic slicing algorithm for aspect-oriented programs, with more accuracy and preciseness. For this we plan to:
    - develop an intermediate representation that correctly represents all the features of an AOP and the various dependencies that exist among the statements of the AOP.
    - propose an efficient dynamic slicing algorithm for aspect-oriented programs based on the above intermediate representation.
- Next, we wish to extend this approach to compute dynamic slices of *concurrent* aspect-oriented programs. For this we plan to:

  - develop an intermediate representation that correctly represents the thread dependencies along with the existing dependencies in an AOP.
  - develop an efficient dynamic slicing algorithm for concurrent aspect-oriented programs using the above intermediate representation.
  - incorporate explicit context-sensitive features into the developed dynamic slicing algorithm to enhance its efficiency.

- Then, we want to propose a technique to compute dynamic slices of *distributed* aspect-oriented programs. For this we plan to:
  - develop an intermediate representation that will accurately represent all the communication dependencies of the distributed AOPs, along with all the basic dependencies existing in AOPs.
  - propose an context-sensitive dynamic slicing algorithm for distributed aspect-oriented programs using the above intermediate representation.
- To develop a partial tool for construction of all proposed intermediate graphs and implementation of all algorithms.
- To compare the performance of our approaches with some of the existing related approaches.
- To develop a technique for software refactoring using our proposed algorithms. For this we plan to:
  - automatically identify the target modules for refactoring using the values of slice-based cohesion metrics.
  - use proposed slicing technique for refactor the target modules.

1.5 Organization of the Thesis

The rest of this thesis is organized into chapters as follows.

Chapter 2 provides the background concepts used in the rest of the thesis. We discuss some concepts and definitions of graph theory which are used later in our proposed algorithms. As all the proposed algorithms are based on intermediate representations, we describe some popular intermediate program representation concepts that are used in existing slicing techniques. Then, we present some concepts of program slicing, and applications. Then, we present an introduction of program refactoring and its advantages. Finally, we discuss the concepts of *precision* and *correctness* of a dynamic slice.

Chapter 3 provides a brief review of the related work relevant to our contribution. We first discuss the work on dynamic slicing of aspect-oriented programs. Then, we describe the work on slicing of *concurrent* aspect-oriented programs. Finally, we discuss the reported work on dynamic slicing of *distributed* aspect-oriented programs.

Chapter 4 presents our dynamic slicing algorithms for simple aspect-oriented programs. We introduce some basic concepts and definitions. We first develop an intermediate representation for aspect-oriented programs to represent all important features of aspect-oriented programs and then present the proposed dynamic slicing algorithms. Then, we present a brief discussion on the implementation of our algorithms. Finally, we compare our dynamic slicing algorithm with some existing algorithms.

Chapter 5 deals with dynamic slicing of *concurrent* aspect-oriented programs. We first introduce some definitions. We develop an intermediate representation for *concurrent* aspect-oriented programs and then present the context- sensitive dynamic slicing algorithm. Then, we give an implementation of our algorithm. Finally, using some case studies, we present a comparison of our proposed algorithm with some existing closely related algorithms.

Chapter 6 describes *distributed dynamic slicing* of aspect-oriented programs running on several nodes connected through a network. We first present some basic concepts and definitions. We develop an intermediate representation for *distributed* aspect-oriented programs. Then, we present a *parallel dynamic slicing* algorithm for distributed aspect-oriented programs. We show that this algorithm computes *correct* dynamic slices. Then, we describe an implementation of our algorithm. Finally, we compare the efficiency of our proposed algorithm with some related slicing algorithms.

Chapter 7 contains the application of program slicing in software refactoring. We first present the use of slice-based cohesion metrics and refactoring. We use these metrics to identify target modules that need refactoring. Finally, we propose a technique to refactor that target modules based on our proposed slicing algorithms.

Chapter 8 concludes the thesis with a summary of our contributions. We also briefly discuss the possible future extensions to our work.

Chapter 2

Background

In this chapter, we present some of the basic concepts that form the basis of this thesis. To keep the content simple, we only describe the concepts in brief and highlight only the essential points. Section 2.1 introduces the concepts of AOP. These concepts are supported by AspectJ, which is an extension of Java programming language, and is used in this thesis for implementing AOP. We have discussed the syntax and features of AspectJ programs. Also, we have discussed the advantages and disadvantages of AOP. In Section 2.2, we presents the different approaches used for program slicing techniques. intermediate graph based slicing. We describe some program representations used by different researchers for program slicing in Section 2.3. In Section 2.4, the applications of program slicing in the field of software development are briefly described. Lastly, in Section 2.5, we present the basic concepts of software refactoring based on program slicing.

2.1 Aspect-Oriented Programming (AOP)

Object-oriented programs (OOPs) were developed at Xerox PARC (by Alan Kay and others) in the 1970s, to represent the prevalent use of objects and messages as the basis for computation. Since then OOP has been the most widely used software development paradigm. Many companies have adopted to OOP methodology for their product development. But, along with the increase in popularity of OOP some drawbacks of the OOP implementation were also noticed. In most large object-oriented software, the mapping of the customer requirements to the program modules that implement these requirements are complex. One requirement may also need several modules to implement it [33]. It means that, a change in one requirement requires to understand and change several modules.

For example, let us consider an Internet Banking System (IBS). This system has some requirements related to new customers such as *registration*, *identity verification* and *address verification*. Then it has some requirements related to accounts such as *minimum balance checking*, *withdraw* and *deposit*. It also requires to manage the customer accounts. All these requirements are core concerns of the IBS, because these are the primary features of a banking system. In Figure 2.1, these core concerns are shown as vertical columns.

Now, as the banking system is a critical system, the system must have some requirements

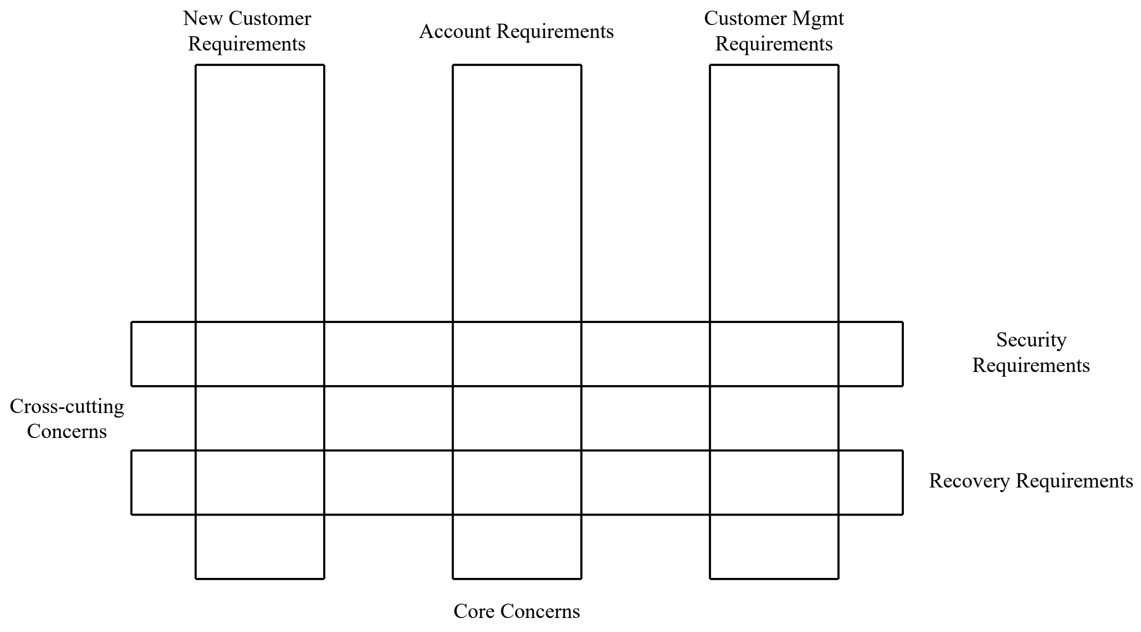


Figure 2.1: Example of cross-cutting concerns

related to the security of the system. Also, it must have recovery requirements to ensure that the data is preserved even on system failures. These additional requirements apart from the core concerns are called cross-cutting concerns, because these concerns are influencing the implementation of all other core system requirements. The cross-cutting concerns are shown in Figure 2.1 as horizontal bars that cross the vertical columns. The cross-cutting concerns are the main point of interest of an AOP.

Definition 2.1: Cross-cutting concerns: Cross-cutting concerns are that parts of a program that are scattered across multiple modules of the program and are also tangled with other modules.

The concept of AOP was developed at Xerox PARC, the same place where OOP was introduced, by Gregor Kiczales et al. [34] in the year 1997. OOP creates a coupling between the core and cross-cutting concerns that is undesirable. Adding new cross-cutting features and even certain modifications to existing cross-cutting functionalities require the modification of the relevant core modules. But, AOP provides separation of cross-cutting concerns from the core modules by introducing a new unit of modularization, called *Aspect*. In AOP, we implement cross-cutting concerns in aspects instead of fusing them in the core modules.

2.1.1 Aspect-Oriented Programming using AspectJ

There are many models available for the implementation of AOP, such as Spring AOP, Aspect C#, and AspectJ. Among all the existing models, AspectJ is the most widely used AOP model. AspectJ is a compatible extension to the Java programming language. We have used AspectJ programming language for developing all the implementations in this thesis.

```
//HelloWorld.java
public class HelloWorld {
public static void main(String args[]){
   display();}
  public static void display(){
  System.out.print("World");
  } }

Output: World
```

Figure 2.2: An example Java program to print *World*

Features of AsepctJ

- *Aspect*: Aspects are like classes in OOP, that contain functionalities. But aspects are different from classes because aspects are meant to compute cross-cutting concerns to be injected into other codes [35, 36].
- *Joinpoints*: Aspects cross-cut objects at only well-defined points, such as at object construction, method call, or member variable access points. Such well-defined points are known as *joinpoints* [36]. Joinpoints include method calls, constructor calls, field accesses, object and class initialization, and others.
- *Point-cut*: The specification for naming a joinpoint is called a *point-cut* [35]. Point-cut is the collection of joinpoints.
- *Advice*: Once the joinpoints are spotted in a program, the intended behavior must be defined [36]. This behavior is called *advice*. An advice can contain anything that an arbitrary Java method can have.
- *Code Introduction*: With code introduction, programmers can add variables and methods into a program entity by using Aspects [36].

HelloWorld example

Let us consider a program HelloWorld.java having one *main()* method and another method called *display()*, as shown in Figure 2.2. The display() method only displays a message ‘‘World”. Now, suppose we want to add some more strings to the output, but without any modification in original program. This requires to write a separate Aspect that will add some string in the output. This job is done by HelloWorld_aspect, as shown in Figure 2.3.

2.1.2 Advantages of Aspect-oriented programming

Aspect-oriented programming is a relatively new technique, but some of the studies show that it is better for modularization of cross-cutting concerns and consequently for accelerating the

```
//HelloWorld_aspect.aj
public aspect HelloWorld_aspect {
  pointcut PC():call (void HelloWorld.display());
  before():PC(){
   System.out.print("Hello! ");
   }
after():PC(){
  System.out.print(": This is AOP ");
   } }

Output: Hello! World: This is AOP
```

Figure 2.3: An example AspectJ program

software development process. The idea is that, well-separated concerns can be more easily maintained, modified, and manufactured. So, the total time of the programmer to develop a software product will be shorter than that of the development time of analogous system, realized without the use of AOP techniques. In addition to this, some more advantages of using AOP are as follows:

1. *Improved design stability*

   In a study by Greenwood et al. [37] on the impact of Aspectual Decompositions on design stability, it was found that the concerns that were modularized using the Aspect-Oriented (AO) techniques, had superior design stability and the modifications tend to remain confined within the target modules.

   The AO design also follows the open-close principle more effectively. That is, a module should be open for extension, but must be closed for modification. This also increases stability of the software design.

2. *Substantial reduction in module size*

   In AO software design, the module size can be reduced considerably and thus makes it easier for developers to implement the modules more efficiently.

3. *Use of AOP in exception handling*

   In the traditional software, many defects are caused due to wrong exception-handling mechanism [38]. The programmers usually treat exception-handling as an ad-hoc process and address them at the last minute, as and when needed. Also programmers rarely reuse exception-handling code. As a result, handling of exceptions generically can result in unreliable software. The main issue that makes exception-handling difficult to manage is that it is difficult to modularize exception-handling using standard OOP languages. Exception-handling can be considered as a cross-cutting concern because they tend to cut across the boundaries of many classes and hence it can be handled more efficiently in AOP [39].

2.1.3 Disadvantages of AOP

Despite it's many advantages, AOP produces some challenges in producing high quality software and it's testing. Some of these are [40]:

- *Aspects do not have independent existence*

  Aspects can be created by keeping in mind the context of another class and also cannot execute on their own. An aspect depends upon the context of another class for its identity and execution.

- *Difficult to determine Control and Data dependency*

  The control and data dependency of an AO program is not apparent from the source code of the aspect or class, because both of them are developed separately. The dependency is solely determined by the weaving process. So, it is difficult to know the control dependence and data dependence before the execution of the program.

- *Debugging and maintenance of AO program*

  In AOP, we inject a block of code that is being run at a given point, i.e. at joinpoint. But, it is very difficult to determine where this block is invoked by just inspecting the source code. If the advice causes some changes, then while debugging an application, it will be very difficult to identify the root cause of a fault [41].

2.2 Program Slicing

In order to reduce the complexity of a program so that it will be easy for testing and maintenance, one approach is to break the whole program into parts. Program slicing is an analysis and transformation technique that uses the dependency relation between the program statements to identify the parts of a program that affect or are affected by a point of interest, called the *slicing criterion*. All the program statements influencing or influenced by the variables mentioned in the slicing criteria are added to the slice. Program slicing was first introduced by Weiser in 1981 [42].

The construction of a program slice starts with the definition of a slicing criterion. A slicing criterion is the set <s,v>, where ‘s’ denotes the statement number and ‘v’ denotes the subset of variables used or defined at ‘s’. In the literature, we found there are two major categories of program slicing approaches. One is the program flow equation based slicing technique, where the dependence information of a given program are represented in number of equations. Then, depending upon the slicing criterion, the equations contributing to its value, are selected and presented as slice. Another category of program slicing is based on reachablity analysis of intermediate graphs. In graph based program slicing techniques, first the dependencies between the statements of a given program are identified. Then, these dependencies are represented in the form of an intermediate graph, representing the statements of the program as nodes, and the dependencies among the statements as edges between the nodes. Then, a reachability analysis is carried out using the intermediate graph to find out the slices. In the next section, we present the different program representations used in program slicing techniques.

2.3 Program Representation

The most popular technique for program slicing is based on representing the program in the form of an intermediate graph and then finding the slices using traversal algorithms. Different types of intermediate representations are used by different researchers to represent a given program and its features. Here, we present some most important types of intermediate program representations found in the literature.

2.3.1 Control Flow Graph (CFG)

A control flow graph for a given program P is a directed graph in which each node represents a statement of P and the edges represent the flow of control in P [43].

Definition 2.1 CFG: Let the set S represents the set of statements of the program P. The CFG of the program P is a directed graph G=(V,E), where V is the set of nodes representing statements of the program P and E is the set of control edges. An edge (x,y) $\in$ E represents the flow of control from the node x to the node y. Each CFG contains two special nodes labeled *Start* and *End* corresponding to the beginning and termination of the program P.

Definition 2.2 Post-dominance: In a CFG, a node i is said to be *post-dominance* by another node j if all the paths from node i to the end node pass through node j, where end node denotes the last statement of the program.

Definition 2.3 Control Dependence Edge: The control dependence edge, $n_1 \overset{cd}{\rightarrow} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$ such that there is a transfer of control from n_1 to n_2.

In a CFG, all the nodes are connected through the control dependence edges. Usually control dependence is used to define the *post-dominance* relationship between two nodes. Let us consider an example Java program for finding summation of numbers from 1 to n, as shown in Figure 2.4. In this program, the *sum()* method is called from *main()* with a parameter value 10. The *sum()* method performs the iterative summation of numbers from 1 to n and sends the result to the callee method. Finally, the computed result is stored in a variable and displayed to the user. The CFG of the example program is given in Figure 2.5. In the CFG, each node represents one statement in the program and edges represent

```
//Summation.java
1. public class Summation{
2.  public static void main(String args[]){
3.    int result;
4.    result=sum(10);
5.    System.out.print(result);
      }

6. public static int sum(int n){
7.    int i,summ;
8.    i=1;
9.    summ=0;
10.     while(i<=n)
11.        { summ=summ+1;
12.        i=i+1;
         }
13.      return(summ);
      } }
```

Figure 2.4: An example Java program for finding summation of ‘n’ numbers

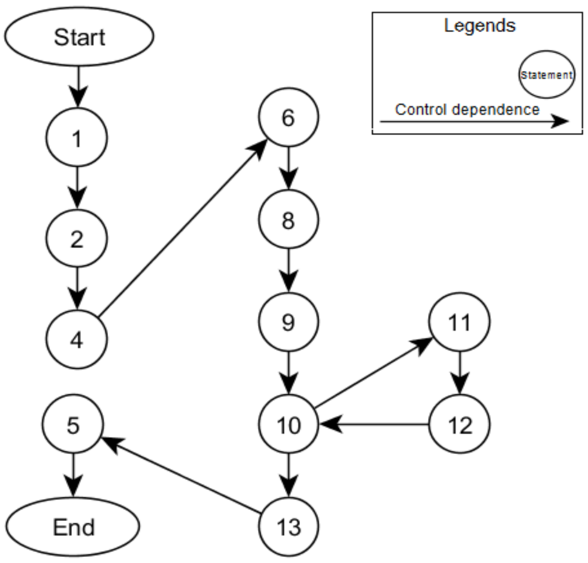


Figure 2.5: CFG for the example program given in Fig.2.4

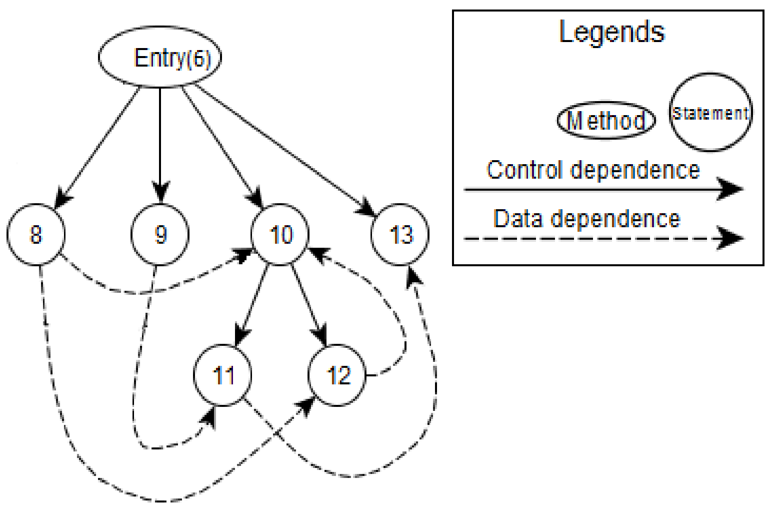


Figure 2.6: PDG for the sum() method of example program given in Fig.2.4

the transfer of control between nodes. The cycle between node 10, node 11, and node 12 represents the *while* loop in the program.

2.3.2 Program Dependence Graph (PDG)

The PDG for a given program P is a directed graph in which the nodes represent the statements of P and the edges represent the dependence relationships between the statements of P [44].

Definition 2.4 PDG: The PDG for a program P is represented as G_P=(V, E) where V is the set of nodes representing statements of the program P and E={control, data}. In a PDG, there are two types of edges; *control dependence edge* and *data dependence edge*.

Definition 2.5 Data Dependence Edge: The data dependence edge, $n_1 \overset{dd}{\rightarrow} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$ such that n_2 is using a variable which is defined at n_1.

The PDG is constructed for only one procedure in the program to show the flow of control and data inside that particular procedure or method. Each PDG has a special node called *Entry* node to represent the method entry point [44]. As shown in Figure 2.6, the PDG has the first node as *Entry* node which represents statement number 6 of the program. Rest of the nodes represent the statements of the program. In Figure 2.6, we show the PDG of the sum() method given in the example program of Figure 2.4. Each node in the PDG is connected with other nodes through control or data dependence edges.

2.3.3 System Dependence Graph (SDG)

In a program, there can be one or more number of procedures. PDG cannot be used to show the dependencies present among all the procedures, as it can only show the dependencies

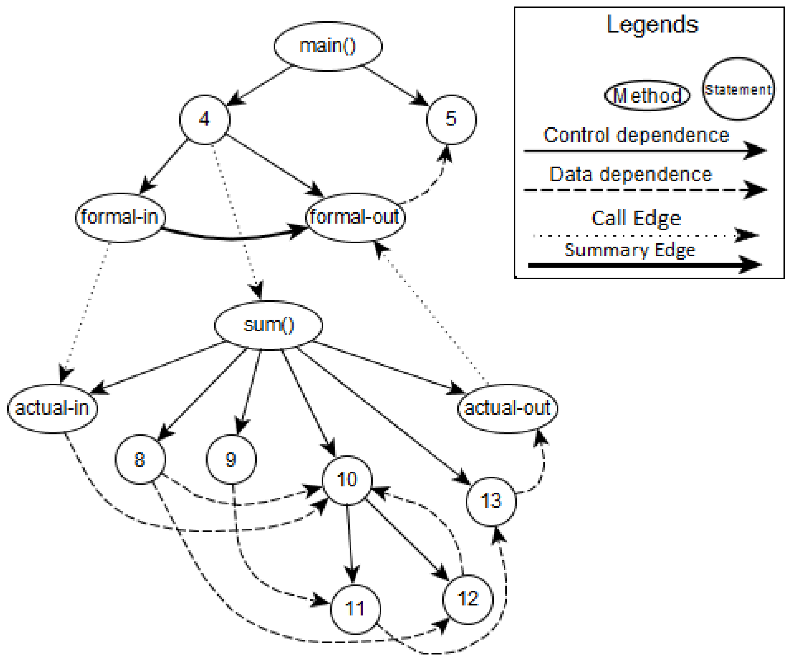


Figure 2.7: SDG for the example program given in Fig.2.4

present within a single procedure. Hence, to represent the whole program as a dependency graph, system dependence graph (SDG) is used. SDG is a collection of PDGs [6]. SDG contains some extra nodes than that of PDG, i.e. *formal-in*, *formal-out*, *actual-in* and *actual-out*. To represent parameter passing, the SDG uses formal parameter vertices: a *formal-in* node for each formal parameter of the procedure; a *formal-out* node for each formal parameter that may be modified by the procedure. On the other side, i.e. the called procedure side, parameter passing is shown through actual parameter nodes: an *actual-in* node for each actual parameter at the call site; an *actual-out* node for each actual parameter that may be modified by the called procedure.

Definition 2.6 Call Edge: The call edge, $n_1 \overset{call}{\rightarrow} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$, n_1 is the method calling node, and n_2 is the method declaration node.

Definition 2.7 Parameter-in Edge: The parameter-in edge, $n_1 \overset{Pin}{\rightarrow} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$, n_1 is the actual parameter, and n_2 is the formal parameter.

Definition 2.8 Parameter-out Edge: The parameter-out edge, $n_1 \overset{Pout}{\rightarrow} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$, n_1 is the return node, and n_2 is the node accepting the value from the called method.

Definition 2.7 Summary Edge: If the formal-out node (n_1) is transitively dependent on the formal-in node (n_2), then there is a summary edge $n_2 \overset{summary}{\rightarrow} n_1 \in E$ from node n_2 to node n_1.

The SDG combines the PDGs procedure corresponding to each by the help of three types

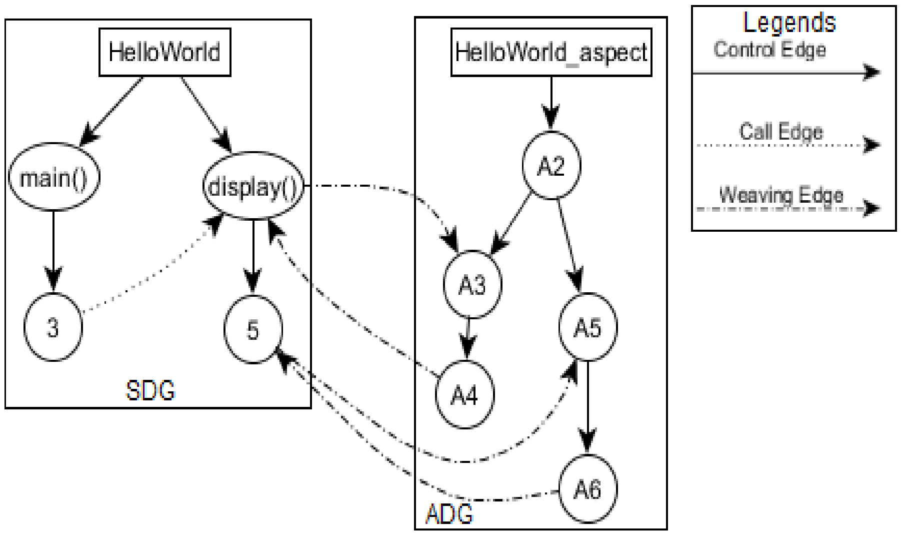


Figure 2.8: ASDG for the example AspectJ program given in Fig.2.2 and 2.3

of edges, i.e. *call* edge, *parameter-in* edge, and *parameter-out* edge. The SDG for the example program of Figure 2.4 is shown in Figure 2.7. The PDGs of the *main()* method and *sum()* method are connected through *call* edge, *parameter-in*, and *parameter-out* edges. Sometimes, there exists a transitive data dependency between the *formal-in* and *formal-out* nodes. This type of dependency is represented by a special edge called *summary* edge, as shown in Figure 2.7.

2.3.4 Aspect-Oriented System Dependence Graph (ASDG)

AOP have additional features compared to procedural or OOP, such as aspects, join points, advices etc. For representing these extra features appropriately, the SDG for OOP is not an efficient option. Therefore, an ASDG is used to represent AOP programs. An ASDG consists of three parts- an SDG for non-aspect part, a group of dependence graphs for aspect part called Aspect Dependence Graphs (ADGs), and some additional dependence edges used to connect the SDG and aspect dependence graph. The ASDG for the HelloWorld AspectJ program given in Figure 2.2 and Figure 2.3, is shown in Figure 2.8. In this figure, the left hand side box represents the SDG for the non-aspect program given in Figure 2.2 and the right hand side box represents the Aspect Dependence Graph (ADG) for the aspect program given in Figure 2.3. These two graphs are combined using the weaving edges.

2.4 Applications of Program Slicing

This section describes the use of program slicing techniques in various applications. Weiser [23] had developed the concept of program slicing for the application of debugging. From this modest beginning, the use of program slicing techniques has now ramified into a powerful set of tools for use in many diverse applications such as program comprehensibility, program verification, computation of software metrics, software maintenance, testing, reverse engineering, parallelization of sequential programs, software integration, software quality assurance, software refactoring etc. [23, 45–52]. A comprehensive study on the applications of program slicing is made by Binkley and Gallagher [53], Lucia [54] and Silva [55]. In the following, we briefly discuss some of these applications of program slicing.

2.4.1 Testing

Software maintenance is required for improving the quality of the software. After maintenance of the software is over, regression testing is carried out to detect the side effects of the changes made during the maintenance [48, 49, 56]. Regression testing is the process of retesting the whole software after any change is made to it. Regression testing of a software is done using large number of test cases that can cover all the changed points in the software. Regression testing is a very time consuming and costly process, because for a minor change in the code, the whole test cases must be rerun. Program slicing in regression testing is used to identify that modules which are affected by a change. Hence, program slicing helps to select only that test cases which are covering the affected modules. So, the number of test cases required for regression testing reduces.

Suppose during maintenance of a program, the value of a variable ‘v’ at statement ‘s’ has changed. If we compute the forward slice with respect to <s,v> and observe that the computed slice is disjoint from the coverage of a particular test case ‘t’, then the test case ‘t’ must not be rerun in regression testing. Lots of work is done in the field of application of program slicing for regression testing [49, 51, 57, 58].

2.4.2 Debugging

Debugging is the process of locating a bug or error and correcting it. Debugging is a very tedious and time consuming task. Mark Weiser [23] observed that in the process of debugging, a programmer mentally slices the program to find the location of a bug, this leads to the development of program slicing. Debugging a large program is a very difficult task. Program slicing is useful in debugging to narrow the search space for a bug. If a program produces wrong value for a variable ‘x’, then we compute a slice of the given program with respect to ’x’ [23, 45]. After computing the slice, the programmer has to search within the slice of the program for the cause of bug, and rest of the code not present in the slice can be ignored. Hence, when a slicer is embedded with a debugger, the discovery of bugs becomes very easy and fast. Some variants of program slicing such as program dicing and program chopping ate also used for debugging. *Program dicing* is used to find the difference between the slices of two variables. It may be the case that the program produces correct result for one variable and the same program produces incorrect result for another variable. Program dicing is used to find a bug in this type of programs [45]. *Chopping* is the process of revealing the statements involved in a transitive dependence from one specific statement (the source criterion) to another (the target criterion). Program chopping [46] is used to restrict the size of slice to the point after it is known that the code is correct. This further reduces the size of a computed program slice.

2.4.3 Software Maintenance

Software maintenance is very complex and time consuming process, because for every modification in the existing program many complex dependence relationships must be considered. Effective software maintenance is accomplished by understanding the dependences in the existing code, so that the changes in the software cannot introduce any new bug. The main problem in software maintenance is the *ripple effect*. During software maintenance, when some code are changed, it will affect other parts of the program. To avoid ripple effect of change during maintenance, it is necessary to know which variables and in which statements will be affected by a modification. This process is supported by program slicing [47].

2.4.4 Differencing

In testing and maintenance, a program changes at several points. After some changes are made to the original program, it is difficult to identify that changing points. A textual comparison on the difference between the changed program with the original program is a straight forward technique. But this is inefficient for large programs. Program slicing is useful in identifying semantic differences between two programs [50]. First, the dependence graphs are build for the *old* and *new* programs. The backward slices for given slicing criteria are generated. Components whose nodes are present in the slices of both graphs, have the same behavior in *old* and *new* programs. The set of nodes present in the slice of *new* graph but not present in the slice of *old* graph, are the components with changed behavior.

2.4.5 Program Integration

Once a module undergoes some changes, it may happen that the changed module not integrate property with rest of the unchanged modules of the software. So program integration is a challenging task. Many program integration techniques are developed in due course of time [59, 60]. One such technique is *Semantic-based* program integration technique [51]. In this technique, first the dependence graphs are build to represent the original program (Base) and its variants (A and B). Then, the slices of the dependence graphs are computed by giving the points of change in the program, as slicing criteria. The computed slices of Base, A and B are merged to form a merged graph. Finally, a program is reconstructed from the merged graph.

2.4.6 Functional Cohesion

Functional cohesion of a module in the software shows the functional independence of a module [61]. A highly functional cohesive software module is one that performs only one function that is atomic in nature. Bieman and Ott [62] proposed a slicing based cohesion measure defined *data slices* that consists of data tokens (instead of statements). A *data token* is a variable with constant definition and reference. Data slices are computed for each output of a procedure (e.g., output to a file, output parameter, assignment to a global variable). The tokens that are common to more than one data slice are the connections between the slices. They are called *glue*. The *glue* binds the slices together. The tokens that are in every data slice of a function are called *super-glue*. *Strong functional cohesion* can be expressed as the ratio of *super-glue* tokens to the total number of tokens in the slice, whereas *weak functional cohesion* may be seen as the ratio of *glue* tokens to the total number of tokens. The *adhesiveness* of a token is another measure expressing the number of slices are glued together by that token.

2.4.7 Parallelization

In a multithreaded program, two or more parts of a program runs simultaneously. Each part is handled by a thread and all the threads execute simultaneously. Program slicing can be used to identify independent parts of a program such that that parts can be assigned to different threads [63]. While computing the slices of a given program, if two slices are not overlapping, then these parts of the given program can be executed parallely.

2.4.8 Reverse Engineering

Reverse engineering concerns the problem of comprehending the current design of a program and the way this design differs from the original design [56]. This involves abstracting the design decisions and rationale from the initial development (design recognition) and understanding the chosen algorithms (algorithm recognition).

Program slicing provides a tool set for this type of re-abstraction [52]. For example, a program can be displayed as a lattice of slices ordered by the *is-a-slice-of* relation. Comparing the original lattice and the lattice after (years of) maintenance can guide an engineer towards places where reverse engineering energy should be spent. Because slices are not necessarily contiguous blocks of code they are well suited for identifying differences in algorithms that may span multiple blocks or procedures.

2.5 Software Refactoring

According to Fowler [64], *''Refactoring is the process of modifying the original structure of the software system to reduce the complexity, but without altering its external behavior''*.

When we perform refactoring, we improve the design of the code after it has been written [64]. In the current practice of software development, we first design and then start coding. But over time, the requirement keeps on changing and hence the code requires modification. As a result of repeated modifications, the integrity of the system, and its structure according to the original design, gradually fade. But, on the other hand, refactoring can take a bad design and rework it into a well design code. Refactoring is also found useful in introducing aspect-oriented features into an existing object-oriented software [65]. Monteiro et al. [66] shown the process of transformation of object-oriented source code into AspectJ code as an application of software refactoring.

2.5.1 Advantages of Software Refactoring

From the literature survey we find several advantages of *Software Refactoring* [31]. Some of them are listed as follows:

1. *Refactoring improves the design of software*

   Generally we change the code to realize short-term goals and as a result of accumulation of these small changes, the design of the program decays. Refactoring helps redesign the code to make the code in accordance with a good design.

2. *Refactoring makes software easier to understand*

   The code for a program must be written in a more descriptive mode and easily understandable format. In general practice, we go on changing and adding code until it gives the desired result. As a result our program is no more easy to understand. Software refactoring re-arranges the code to improve its understandability.

3. *Refactoring helps find bugs*

   Refactoring helps improve the understandability, and hence also helps find bugs. This is because finding bugs in a whole structured code is easier than finding them in an unstructured code.

4. *Refactoring helps develop code faster*

   All the earlier points mentioned above, conclude that refactoring helps write program faster. A good design is essential for rapid software development and refactoring enhances the design of a program. Hence, it helps develop a software faster.

2.6 Summary

In this chapter, we have discussed some basic concepts and definitions that are used later in our proposed approaches. The concept of AOP is discussed in detail along with its advantages and disadvantages. We discussed the implementation of AOP using AspectJ programming language. We have also discussed various existing intermediate program representations used by various researchers. We discussed some important applications of program slicing. Finally, we describe in brief the process of software refactoring and its advantages.

Chapter 3
Review of Related Work

This chapter presents an overview of the basic program slicing techniques and includes a brief history of their development. First, we discuss the work done by previous researchers on dynamic slicing of object-oriented programs. We present a popular two-phase slicing algorithm in details with an example. Then, we briefly discuss slicing of aspect-oriented programs. Next, we have presented the works in the field of slicing of concurrent object-oriented programs. Then, we present the research work done in the field of concurrent aspect-oriented programs. Subsequently, we present the work done in the field of slicing of distributed object-oriented programs and slicing of distributed aspect-oriented programs. Finally, we discuss some work carried out on software refactoring.

3.1 Slicing of Object-Oriented Programs

In slicing of object-oriented programs, developing intermediate representation of the program is an important issue. Present-day software systems are basically object-oriented. Object-oriented features such as classes, inheritance, polymorphism need to be considered carefully in slicing. Due to presence of polymorphism and dynamic binding, the process of tracing dependencies in OOPs becomes complex. Slicing of OOPs has been addressed by several researchers [6, 13, 47, 48, 65, 66, 83, 115].

Horwitz et al. [6] developed system Dependence Graph (SDG) as an intermediate program representation for procedural programs with multiple procedures. As explained in Section 2.3.3, the SDG is built for a given program. For example, let us consider an example program and its corresponding SDG as shown in the Figure 3.1. The program declares two variables x and y, then a method *check()* is called by passing these variables as parameter. If the parameter values is positive, then *check()* returns square of the parameter. If the parameter is negative, then it returns the positive value of the parameter.

```
public class Main {
 public static void main(String args[]){
  int x, y;
  x=1;
  y=-1;
  x=check(x);
  y=check(y);
  System.out.println(x);
  System.out.println(y); }
  }
  public static int check(int i){
      if(i>0)
         i=i*i;
     else
         i=-i;
      return (i);}
```

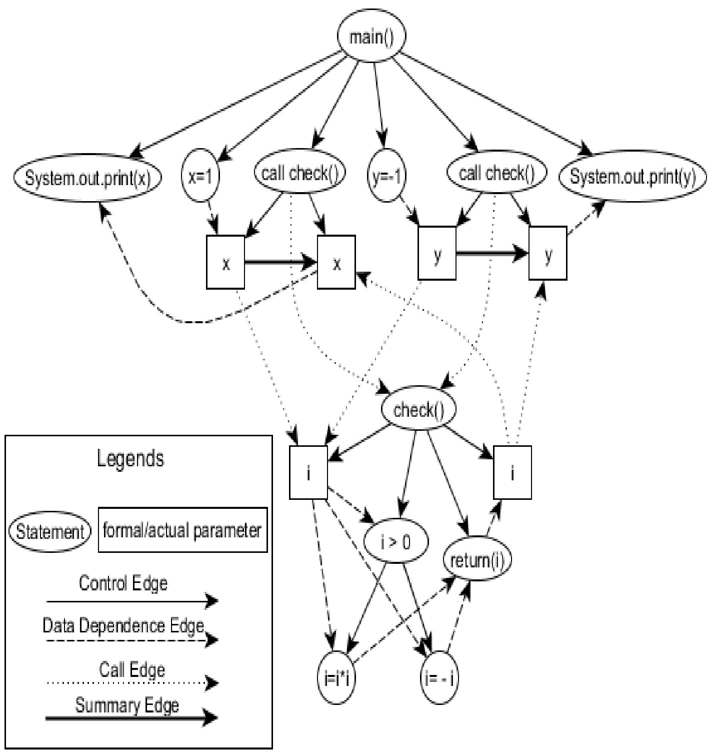


Figure 3.1: An example program and it's SDG

They [6] proposed a two-phase graph reachability algorithm as shown in Algorithm 1, on the SDG to compute the inter-procedural slices. Algorithm 1 is a worklist implementation of the two-phase slicing algorithm. In the first phase, the algorithm starts with a given slicing criterion node and traverses backward along all the edges except *parameter-out* edges. Algorithm 1 uses a worklist data structure W_1, which is initialized with the slicing node. In the first phase, we remove one node from W_1 and check all its incoming edges. All the edges are processed except the *parameter-out* edge. This process continues till W_1 is empty. This algorithm marks all the nodes reached during this traversal and includes them into the slice. Then, in the second phase, the algorithm traverses backward along all the edges except *call* and *parameter-in* edges. In this phase, the algorithm uses a worklist data structure W_2. We process each node in W_2 and check its incoming edges. In this phase, all the edges are considered for traversal except *call* and *parameter-in* edges, because these are already processed in the first phase of the algorithm. The second phase continues till W_2 becomes empty. It includes all the nodes reached during both the traversals into the slice. The union of nodes marked in phase one and phase two gives the slice of the given program with respect to the input slicing criterion.

Algorithm 1 : Two-phase slicing algorithm

INPUT: SDG $G < V, E >$, a slicing criterion s.
OUTPUT: The Slice S for s.
INITIALISE: Worklists $W_1 = \{s\}, W_2 = \{\}, S = \{s\}$.

```
1:  while $W_1 != \Phi$ do                               ▷ phase 1
2:      $W_1 = W_1 - \{n\}$                              ▷ process the next node in $W_1$
3:      for all $m \rightarrow_e n$ do                   ▷ handle all incoming edges of n
4:          if $e \notin \{p_o\}$ then                   ▷ if e is not a parameter-out edge
5:              if $m \notin S$ then
6:                  $S = S + \{m\}$
7:                  $W_1 = W_1 + \{m\}$
8:              end if
9:          else $e \in \{p_o\}$
10:             $W_2 = W_2 + \{m\}$
11:         end if
12:     end for
13: end while
14: while $W_2 != \Phi$ do                               ▷ phase 2
15:     $W_2 = W_2 - \{u\}$                              ▷ process the next node in $W_2$
16:     for all $v \rightarrow_e u$ do                   ▷ handle all incoming edges of u
17:         if $e \notin \{p_i, call\}$ then             ▷ if e is not a parameter- in or call edge
18:             if $v \notin S$ then
19:                 $S = S + \{v\}$
20:                 $W_2 = W_2 + \{v\}$
21:             end if
22:         end if
23:     end for
24: end while
```

Let us take the slicing criterion $<$ 8, x $>$ for the program given in Figure 3.1. Initially the worklists $W_1 = \{8\}, W_2 = \{\}$, and $S = \{8\}$. Phase 1 starts at the slicing criterion and traverses backward through all the edges except the parameter-out edges, while the source nodes of encountered parameter-out nodes are saved in a worklist W_2. The nodes visited in phase 1 of the algorithm is shown by *green* color nodes in Figure 3.2. Phase 2 starts from the first node in worklist W_2 and traverses backward through all the edges excepting call and parameter-in edges. All the nodes visited in phase 2 are colored in *cyan* color nodes in Figure 3.2. Because of the summary edges, there is no need to return from a called procedure back to the callee. The resulting slice consists of all the nodes visited in phases 1 and 2.

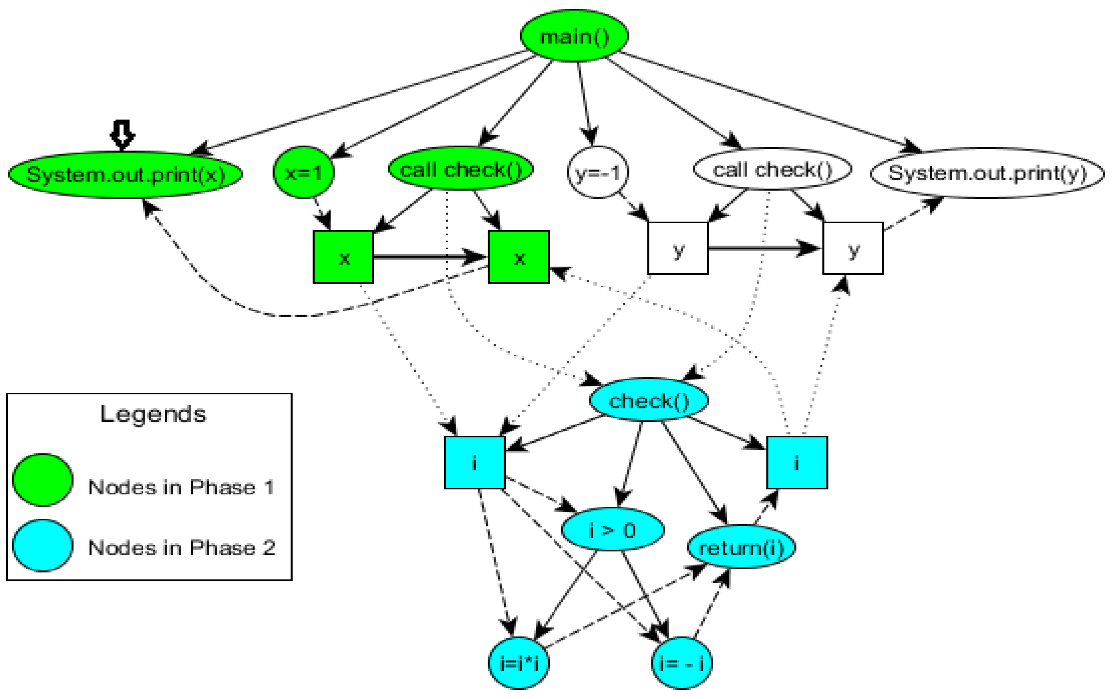


Figure 3.2: Resulting slices using 2-phase algorithm for the example program given in Fig.3.1

Later, Larsen and Harrold [10] extended the SDG of Horwitz et al. [6] to represent object-oriented programs. Their extended SDG incorporates many object-oriented features such as classes, objects, inheritance, polymorphism etc. After constructing the SDG, Larsen and Harrold used the two-phase algorithm to compute static slices of object-oriented programs.

Mohapatra et al. [67] have developed edge-marking and node-marking dynamic slicing techniques for object-oriented programs. Their algorithms are based on marking and un-marking of the edges or nodes of the graph appropriately, as and when dependencies arise and cease.

3.2 Slicing of Aspect-Oriented Programs

Program slicing was initially proposed for procedural programming language and has been extended to cope with the Object-Oriented paradigm by Harrold et al. [10] and Mohapatra et al. [68]. Now-a-days the application of program slicing technique to Aspect-Oriented programs is gaining most of the researchers attention.

A preliminary work in this area has been done by Zhao [69]. He proposed an *Aspect-oriented System Dependence Graph(ASDG)* that is an extension of Object-Oriented SDG. The ASDG consists of a system dependence graph for non-aspect code, some dependence graphs for aspect code and special edges for combining these dependence graphs.

According to his observations, an SDG can be created by combing the method dependence graphs (MDGs) for call sites. He represented the advices in an aspect through Advice Dependence Graph (ADG), the introductions of an aspect as Introduction Dependence Graph (IDG). Both, ADG and IDG are constructed in a manner similar to the construction of MDG. The Aspect Dependence Graph (AsDG) is constructed by combing all MDGs, ADGs and IDGs of an aspect-oriented program. He also introduced some special edges for AOP such as *aspect membership edge* and *coordination arc*. But the work by Zhao [69] suffers from some limitations. In this paper, he had not mentioned any particular slicing algorithm and its implementation. Also, the important features such as around advice and pointcut representations, are not handled in this approach. Finally, the proposed slicing technique is a static program slicing technique.

Braak et al. [70] extended the ASDG proposed by Zhao [69] to include inter-type declarations in the graph. Each inter-type declaration was represented in the form of a field or a method as a successor of the particular class. They introduced *Aspect/Class membership edge* to connect the members of an aspect or class to the *Aspect* or *Class* nodes. They represented one of the important feature of AOP i.e. *around advice* in their ASDG, which was not handled by Zhao [69]. The two-phase slicing algorithm of Horwitz et al. [6] is used to find a static slice of Aspect programs. The ASDG is created statically and the two-phase algorithm used in this paper is also static. So, the proposed slicing technique is a static approach which computes slices of larger size.

Mohapatra et al. [71] proposed a dynamic slicing technique for AOPs. They have proposed a Dynamic Aspect-oriented Dependence Graph (DADG) for representing the features of AOPs. This is a dynamic graph that is built at run time. They compute the execution sequence of a program and represent the executed statements in the form of DADG. Based on the DADG, they have proposed a *Trace file Based Dynamic Slicing (TBDS)* algorithm. The TBDS algorithm is based on the traversal of DADG in a random manner from the slicing node to all reachable nodes. Finally, they map the selected nodes of DADG present in a slice, into the statements of the given program. The limitation of this approach is that DADG does not represent many features of an AOP such as method call, aspect, around advice, introduction etc. Also, the proposed slicing technique is based on the execution trace of a program. Hence, for each execution of a program, a new DADG is constructed, which is much time consuming. No specific slicing algorithm is proposed that can handle the AOP features more efficiently.

Sahu et al. [71] constructed the Extended-ASDG (EASDG) as the intermediate representation of Aspect-oriented programs. Their EASDG differs from the ASDG of Zhao [69] in two ways: each point-cut in the AspectJ program is shown explicitly and also the weaving process is represented in EASDG. The construction of EASDG starts with the construction of SDG for non-aspect code of the AOP. Then the aspect dependence graph (ADG) is constructed. The ADG is the combination of four graphs i.e. advice dependence graph, introduction dependence graph, pointcut dependence graph and method dependence graph. After constructing the EASDG, they have applied a node-marking algorithm on the EASDG to compute dynamic slice of aspect-oriented programs. While executing the given program, if a statement of the program executes then its corresponding node in EASDG is marked. When the control moves out of the current method then the marked nodes are unmarked. The limitation is that Sahu et al. [71] have not mentioned the around advice in their EASDG construction algorithm. Also, the marking and unmarking of nodes depending upon the execution of individual statements of the program at run time, makes the slice computation more time consuming.

Raheman et al. [27] has overcome some of the drawbacks of Zhao by computing dynamic slices and adding a new type of edge called ‘‘ Weaving Edge” into the Extended Aspect-oriented System Dependence Graph (EASDG) proposed by Zhao [69]. But this approach does not compute an executable slice. They have added point-cut declaration but not considered the aspect class declaration in the EASDG.

3.3 Slicing of Concurrent Object-Oriented Programs

Very intensive work have been done in the field of slicing of concurrent OOPs, by different researchers [16, 72–75]. Probably, the earliest approach for slicing of concurrent OOP was the work of Zhao [69]. Zhao presented a dependence based representation called the *system dependence net* (SDN) which extends the previous dependence based representations [9] to represent various dependence relationships in concurrent object-oriented programs like Compositional C++ (CC++) programs [30]. An SDN of a concurrent object-oriented program consists of a collection of dependence graphs each representing a main procedure, a free standing procedure, or a method in a class of the program. It also consists of some additional arcs to represent direct dependencies between a call and the called $procedure/method$ and transitive inter-procedural data dependencies. To represent interprocess communications between different methods in a class of a concurrent object-oriented program, they have introduced a new type of program dependence arc named as *external communication dependence arc* into the SDN. An SDN can be used to represent either object-oriented features or concurrency issues in a concurrent object-oriented program.

Based on the SDN, a two-phase algorithm is used to compute static slices of concurrent object-oriented programs. In CC++, synchronization between different threads is realized by using a single assignment variable. Threads that share access to a single assignment variable can use that variable as a synchronization element. Their system dependence net is an extension of the SDG of Larsen and Harrold [10] and therefore can be used to represent many object-oriented features in a CC++ program. To handle concurrency issues in CC++, they used an approach proposed by Cheng [76] which was originally used for representing concurrent procedural programs with a single procedure each. However, their approach, when applied to concurrent Java programs suffers from some problems due to the fact that the concurrency models of CC++ and Java are essentially different. While Java supports monitors and some low level thread synchronization primitives, CC++ uses a single assignment variable mechanism to realize thread synchronization. This difference leads to different sets of concurrency constructs in both languages, and therefore requires different techniques to handle concurrency issues in computing slices.

In another work, Zhao [72] has proposed a slicing technique for concurrent Java programs. He designed a dependence-based representation called Multithreaded dependence graph (MTDG) for concurrent object-oriented programs. The MTDG is an extension of SDG proposed by Larsen and Harrold [10]. In MTDG, he had added two new types of dependencies, synchronization dependency and the communication dependency. They had used Horwitz's 2-phase slicing algorithm [6], to compute the slices using MTDG. But, this paper does not address anything about the implementation issues.

Another useful slicing technique for concurrent programs is presented by Krinke [16]. This algorithm takes into account the time-sensitive information to compute slices. The algorithm computes precise and accurate slices for concurrent programs using threaded Interprocedural Program Dependency Graph (tIPDG). Krinke has identified a new type of dependency called interference dependency in a concurrent program. In these cases, one variable declared in one thread can be used in another thread. It is shown that if interference is present in the program, then the traditional two-phase algorithm is not suitable to compute precise slices. But this slicing technique is a static technique. The author has not considered dynamic aspects.

Chen and Xu [77] have developed *concurrent control flow graphs* (CCFG) and *concurrent program dependence graphs* (CPDG) to represent concurrent Java programs. Based on the CPDG, they proposed a static slicing algorithm for concurrent Java programs [77]. In their algorithm, they have considered the fact that the inter-thread data dependence is not transitive. But, they have not considered the dynamic slicing aspects.

3.4 Slicing of Concurrent Aspect-Oriented Programs

In the literature, we found very few work related to slicing of concurrent AOPs. The paper by Ray et al. [14] is a closely related work with our proposed slicing technique. Ray et al. [14] have used an intermediate graph called *Concurrent Aspect-oriented System Dependence Graph (CASDG)* in their slicing technique. CASDG is created for each current execution trace, starting from scratch. Next time for a different execution trace, another new CASDG is formed. In our approach, we use the previously available information of the MAODG. They extended the existing *Node Marking Dynamic Slicing (NMDS)* algorithm, proposed by Mohapatra et al. [71]. Our slicing technique is based on context-sensitivity.

3.5 Slicing of Distributed Object-Oriented Programs

Duesterwald et al. [78] have proposed a dynamic slicing algorithm for distributed C programs. In computing the slices, they have first developed an intermediate representation called *Distributed Dependence Graph (DDG)*, which represents the communication dependence between the statements in addition to the control and data dependence. The control dependence is identified and added to the DDG statically i.e. during compilation of the program. The target program is instrumented with some additional codes for finding the data and communication dependence information. So that dynamic information can be collected while the program is in execution. An execution trace-based re-execution of the program is also used to collect the dynamic data and communication dependence information. Their proposed dynamic slicing algorithm is a distributed algorithm that works parallely on individual parts of the distributed program. The main disadvantage of this approach is the use of trace file, which is more time consuming. The implementation and practical applicability of the proposed algorithms are not discussed. Also, it does not handle the features of OOP and AOP.

Kamkar et al. [79] have used an intermediate graph called *Distributed Dynamic Dependence Graph (DDDG)*, which is based on the execution trace file. The data and communication dependences are identified at the run time of the program. DDDG does not provide any representation for concurrency within the program. After construction of the DDDG, a sequential graph traversal algorithm was used to find the nodes present in a slice. But, the size of the DDDG is large, because they have used a ‘‘Complete statement instance-based” approach for construction of DDDG. In their approach, for each instance of a statement in the execution trace one node is created. It makes the graph size exponential and the graph generation time very large. Also, their proposed slicing algorithm is sequential, which takes more time to compute the slices.

Li et al. [80] have developed an approach for predicate-based dynamic slicing of distributed programs. Apart from the traditional slicing, a predicate-based slicing finds that parts of the program that influence the predicates. They have used the concept of the global predicate, which is a logical formula defined over local variables used in parallel programs. The global predicates are found to be useful in capturing the abstract design requirements and defining program behaviour. They have developed two slicing algorithms. One is the *coarse-grained* dynamic slice, and the other is a *fine-grained* dynamic slice. Their algorithms are based on the partially ordered multi-set (POMSET) model. Their proposed algorithms are capable of computing dynamic slices of distributed programs that use message passing for communication. But, the authors have neither considered communication through shared variables nor object-orientation aspects in their proposed approach. Also, their proposed algorithm is not able to handle the distributed AOP features.

Mohapatra et al. [81] have proposed a dynamic slicing algorithm for distributed object-oriented programs. First, they construct the *Distributed Program Dependence Graph (DPDG)* for a given distributed OOP. The graph is constructed statically, followed by marking and unmarking of the nodes at run-time to capture the dynamic dependencies. In the DPDG, an additional node called C-Node is created to represent the communication dependencies in the program. The C-Node does not represent any statement of the program, but it is just a logical node created to show the communication dependencies in a distributed object-oriented program. They have proposed a slicing algorithm called Distributed Dynamic Slicing (DDS) algorithm. The DDS algorithm marks an edge when the dependency related to the edge arises, and unmarks the edge when that dependency ceases. Their approach computes precise slices for distributed OOPs. But, the marking and unmarking of edges make the slice computation process more time consuming. Also, the DPDG does not represent the non-determinism behavior of the distributed programs. The size of DPDG is large because it stores the relative local slices on each node of the graph. In this slicing approach, the additional AOP features are not considered.

Barpanda et al. [82] have extended the theory of state restriction for handling the different dependencies in a dependence graph. The DPDG of Mohapatra et al. [81] has been used by Barpanda et al. as an intermediate representation. Then, they have adopted the popular graph coloring technique [83] for computing the slices of a distributed object-oriented program. The graph coloring technique for a given graph is to color all nodes of the graph with the minimum number of colors, such that no two nodes sharing the same edge can have the same color. Barpanda et al. [82] have modified the graph coloring technique to *Contradictory Graph Coloring Algorithm (CGCA)*. They have considered the chromatic number of the graph to be 1, i.e. two nodes sharing the same edge can be colored with one color. They have compared the efficiency of their CGCA with the DDS algorithm proposed by Mohapatra et al. [81]. CGCA implemented on 15 programs and shown that the algorithm is both time and space efficient than DDS algorithm. But, the authors have not discussed the implementation issues of their proposed slicing algorithm. They have used the DPDG as the intermediate graph representation, which does not represent the non-deterministic behavior of a distributed program. Also, the AOP features are not considered in DPDG; hence it unsuitable for slicing of AOPs.

Cheng [76] presented an alternate dependence graph-based algorithm for computing dynamic slices of procedural distributed and concurrent programs. The author used *Program Dependence Net* (PDN) as the intermediate representation. The PDN representation of a concurrent program is basically a generalization of the initial approach proposed by Agrawal and Horgan [84]. The PDN vertices corresponding to the executed statements are marked, and the static slicing algorithm is applied to the PDN sub-graph induced by the marked vertices. So, if a statement in a while loop is executed in some iteration, then the corresponding vertex is marked and included in the slice. But, if that statement is not executed in some other iteration, then that marked vertex is not removed from the slice. So, this approach yields inaccurate slices for programs having loops.

3.6 Slicing of Distributed Aspect-Oriented Programs

Only one work [13] is carried outon slicing of DAOPs, by Ray et al. They have used an intermediate graph called Distributed Aspect-oriented Program Dependence Graph (DAPDG) to represent the aspect-oriented features. This is the extension of the Distributed Program Dependence Graph (DPDG) proposed by Mohapatra et al. [81]. One special node *Comm-Node* is used to represent the communication dependence between statements of the distributed AOP. DAPDG is constructed by creating the DPDG for non-aspect code and Aspect Dependence Graph (AsDG) for aspect code, separately. Then, these intermediate graphs are combined using weaving edges to construct DAPDG. DAPDG is created for each current execution trace, starting from scratch. Next time for a different execution trace, another new DAPDG is formed. In our approach, we use the previously available information of the DADG.

Ray et al. [13] proposed a Parallel Aspect-oriented Dynamic Slicing (PADS) algorithm that extends the existing node-marking algorithm, proposed by Sahu et al. [71]. PADS algorithm has three steps, as shown below:

- Statically construction of DAPDG.
- Updation of the DAPDG at run-time to show dynamic dependencies.
- Computation of dynamic slice.

But, the PADS algorithm is a sequential algorithm then, not a parallel algorithm. Hence, the speed of slice computation is slow. Also, the PADS algorithm does not represent the non-determinism behavior of the distributed AOPs. Size of DAPDG is large because it stores the relative local slice on each node of the graph. The work of Ray et al. [13] is most closely related to our proposed slicing technique for DAOPs.

3.7 Software Refactoring: An Application of Program Slicing

Not much work has been done in the field of refactoring of software using slice-based metrics. In this section, we compare our proposed technique with some of the existing work. Wang et al. [85] have developed a tool called SEGMENT, that inserts blank lines into the given method to increase the readability of the program. The authors tried to identify the important points in a program where there is a need of vertical space between the lines of code to improve readability. But, their technique does not make any change into the internal structure of the program so as to improve the program complexity. Our approach identifies that methods of a given program which are more complex and reduces their complexity by splitting them into a number of methods.

Bavota el al. [86] have proposed an Extract Class refactoring method based on graph theory. They have used structural and semantic analysis to identify the relationships between the methods of a class. One semi-automated tool is developed to improve the cohesion of a class by identifying refactoring opportunities. Here the target class for refactoring is to be identified by the software engineers. The developed tool does not provide any support for the detection of more complex classes. Also, the proposed approach does not consider the class inheritance while performing the class refactoring, hence it may cause compilation error or an unexpected change in the behaviour of the program. In our proposed refactoring technique, we provide an approach to find the complex methods in a given software by calculating their slice-based cohesion metrics. Also, we do not change the overall behaviour of the program, so as to avoid the compilation error or an unexpected error.

Mohsin et al. [87] discussed code restructuring by using program slicing. In that work, they have shown that program slicing can be used for decomposition of modules. They found that by decomposition, the coupling is lowered to 40 % and cohesion has grown to 70% more than before. But this work does not address the most crucial point, i.e. how to decompose a module. In this thesis, we discuss in detail the decomposition process of a module.

Monteiro et al. [66] presented an approach for refactoring of object-oriented programs and conversion into aspect-oriented programs. They collected 17 refactoring techniques to identify the cross-cutting concerns from the programs. In the first phase, all the cross-cutting concerns were moved into aspects, leaving behind only the core object-oriented programs. In the next phase, the refactoring techniques are again applied onto the newly created aspects to remove the duplicate codes. That work concentrated on application of refactoring, not on any new refactoring technique. In our work, we present a new refactoring technique. Our proposed technique can be applied recursively on a program till the cohesion metrics of all it's methods improve upto the threshold values.

Monteiro et al. [88], have used *Code Smells* to identify the modules where refactoring is needed. But, code smells do not provide precise criteria when refactoring are belated. Instead, code smells suggest symptoms of the presence of bugs in the code. Their approach does not give any quantitative information regarding the cohesion metrics. But, in our technique, we have used slice-based metrics to identify the methods where refactoring must be applied. Slice-based metrics are quantitative and they can be compared with the benchmark values for cohesion metrics.

Sward et al. [89] have proposed that cohesion, coupling, and cyclomatic complexity (CC) can be used to determine the target module that needs refactoring. With detail example, they have shown that refactoring the existing module into two modules, reduces the coupling and CC. They have proposed that the modules could be sliced with respect to a set of slicing criteria. It reduces the average complexity of the module by the same amount as that it had been sliced with respect to individual slicing criterion. In our example program, we have adopted this technique. But, in Sward's approach [89] the authors have not mentioned clearly how to compute the cohesion, coupling, and CC values. Also, they have not given any idea on how to identify the target modules for refactoring, and what should be the benchmark values for cohesion, coupling, and CC. We have addressed all these issues in this thesis.

Applying the refactoring process on a given software will enhance the design if a collection of different refactoring techniques are applied to the software in a proper sequence. Generating a proper sequence of refactoring is an NP-hard problem. Lee et al. [90] have developed a genetic algorithm based technique to generate the optimized sequence of refactoring to be applied on the target software. Our proposed refactoring technique should be applied on a software after all the remaining refactoring sequences are followed, because our proposed technique works on the individual methods to further enhance the complexity of the software.

3.8 Summary

In this chapter, we have briefly reviewed some work on slicing of object-oriented programs relevant to our research. We have discussed the work on slicing of simple aspect-oriented programs. We have also presented the recently reported results on slicing of concurrent object-oriented programs and slicing of concurrent aspect-oriented programs. Then, we have discussed some research work on slicing of distributed object-oriented and slicing of aspect-oriented programs. Some useful work is discussed in the field of software refactoring.

Chapter 4
Dynamic Slicing of Aspect-Oriented Programs

Aspect-Oriented Programming (AOP) is a new programming paradigm proposed for cleanly modularizing the scattered and tangled code known as cross-cutting concerns (such as exception handling, synchronization, and resource sharing, etc. [34]. The presence of such cross-cutting concerns in a standard language constructs (such as Java) usually results in poorly structured code. AOP controls the scattering and tangling of such code, thereby improving the structure of the program and making it easier to develop and maintain. Due to various specific features of AOP, existing representations for procedural and object-oriented programs cannot be used directly for aspect-oriented programs. Therefore, we need to develop a new intermediate representation for AOP for better program comprehension.

Before developing a slicing algorithm, a suitable intermediate graph must be designed to represent any given AO program. We have proposed an intermediate graph called Extended Aspect-Oriented System Dependence Graph (EAOSDG) to represent AOPs. In this chapter, tow type of slicing algorithms are proposed. First, an Extended Two-Phase slicing algorithm is proposed, which traverses the EAOSDG of a given AO program and generates slices depending on the given slicing criterion. The second slicing algorithm also works on EAOSDG with an intention to preserve the method calling context. All these slicing algorithms are explained in this chapter and a comparative study is presented to find out the best algorithm out of the two proposed algorithms.

In this chapter, first we present the concept of context-sensitive and context-insensitive slicing. We have explained the technique with examples. Then, we present the intermediate program representation used to represent AOPs. Next, we present our first proposed slicing algorithm, which is an extension of two-phase slicing algorithm. Subsequently, we propose another dynamic slicing algorithm with context-sensitive property. Finally, we present the implementation of our proposed algorithms and comparison with other research works.

4.1 Basic Concepts

In this section, we present the concept of *context-sensitive* during the computation of slices and its advantage over context-insensitive slicing.

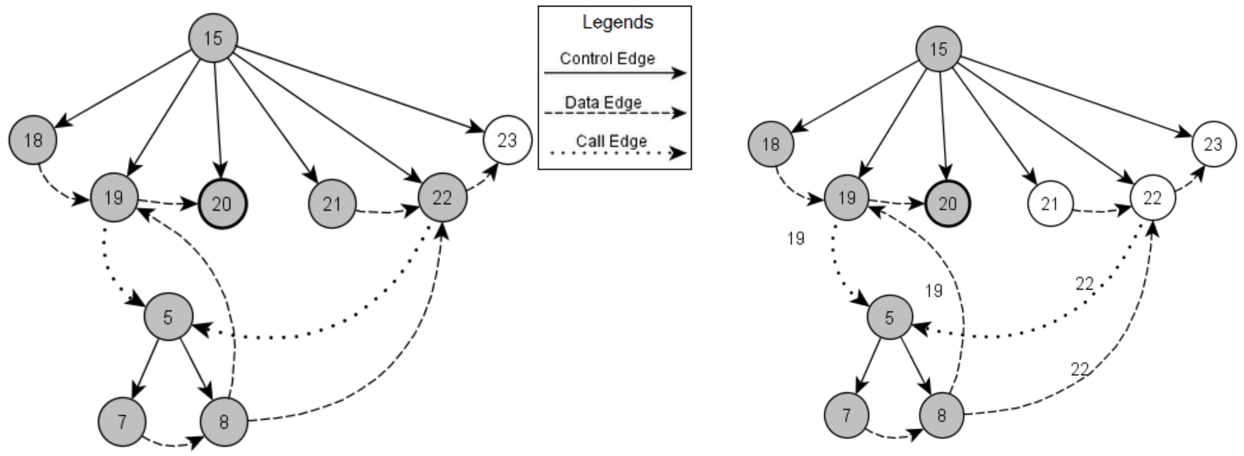
4.1.1 Context-insensitive Vs Context-sensitive slicing

The slicing algorithm must compute accurate and precise slices. The slicing technique suggested by Wieser [42] computes the affected statements for a given slicing criterion by simply considering reachability of the statement. This type of slicing does not always produce accurate slices. For example, consider the code segment given in Figure 4.1. This code segment is for calculating the area of two squares and displaying the results. The *square()* method is called two times from main, i.e. at line number 19 and line number 22. The system dependence graph of the code segment is shown in Figure 4.2a. In the dependence graph shown in Figure 4.2a, we compute a slice with *node 20* as slicing criterion. We traverse the graph in the backward direction starting from node 20. During the traversal, we visited node 19, node 18 and also node 8. Node 8 belongs to square() method. Now, we start traversing backward. Inside the square() method, we have visited node 7 and node 5. *While going backward from node 5, we find two call edges, one from node 19 and another from node 22. It is obvious to include node 19 into the slice, but as we are not putting any constraint on the return call edges, we have to include node 22 also into the slice.* As a result, the final slice will contain node 22 and node 21, which are *extra nodes*. This type of unconstrained slicing approach is called *context-insensitive* slicing. The nodes included in the resultant slice are shown as shaded nodes in Figure 4.2a. The slices generated by the context-insensitive slicing algorithm are inaccurate and larger in size, because many extra nodes are included in the generated slices.

Context-sensitive slicing puts constraints on the process of slice computation [16]. In the example program shown in Figure 4.1, suppose we want to compute context-sensitive slice where the calling sites are preserved. It implies that during backward traversal of a graph representing a program, when we move from the called method to the callee method, we must return to the same callee method through which we have entered into the called method. This is called *context-sensitive slicing*. To implement context-sensitive slicing, we have used labels on call edges, as shown in Figure 4.2b. So while entering into *square()* method from node 19 and start processing node 8, we store the label of the edge, i.e. 19. Then, we proceed backward and visit node 7 and node 5. Now, when returning from the called method (node 5) to the callee method (node 19 and node 22), we look at the stored label. We consider only that edges, whose labels match with the labels stored previously. As a result, we process only node 19 and exclude node 22. The nodes included in the context-sensitive slice are shown as shaded nodes in Figure 4.2b, which does not include nodes 21 and node 22.

```
...
...
5    public int square(int a) {
6    int result;
7    result = a * a;
8    return result;
}
...
...
15   public static void main(String args[])
16   { int side;
17   int area;
18   side=10;
19   area= square(side);
20   System.out.println("Area of Square 1 ="+area);
21   side=5;
22   area= square(side);
23   System.out.println("Area of Square 2 ="+area);
24   }
```

Figure 4.1: An example of simple method call



(a) A context-insensitive slice (b) A context-sensitive slice

Figure 4.2: Slices of the program with slicing criterion $< 20 >$

```
//primebetween.java
import java.util.*;
1 public class primebetween {
2   private static int n;
3   public static void main(String [] args) {
4        n = Interger.parseInt(args[0]);
5        System.out.println("Prime numbers:");
6        primeList(n); }
7   public static void primeList(int n) {
    int i;
    boolean j;
8        i = 1;
9        while(i < n + 1) {
10          j = prime(i);
11          if(j)
12             System.out.println("\t" + i);
13          i++; } }
14  public static boolean prime(int n) {
       int i,r;
15       i = 2;
16       while(i < n) {
17          r = n - ( n / i ) * i;
18          if(r == 0)
19             return(false);
20          i++; }
21       return(true);
    } }

\\primeAspect.aj
22 public aspect primeAspect {
23  pointcut checkprime():
call(boolean primebetween.prime(int));
24  pointcut chechkvarn():
get(int primebetween.n);
25  before(): checkprime(){
26     System.out.println("Prime() is called");
  }
27  after() returning: checkprime(){
28     System.out.println("Returned Normally");
  } }
```

Figure 4.3: An Example AspectJ program

4.2 Intermediate Program Representation

We have proposed an intermediate representation called *Extended Aspect-Oriented System Dependence Graph* (EAOSDG), that represents the features of the Aspect-Oriented Programs. The EAOSDG is a directed graph, $G = (V, E)$, where V is the set of nodes and E is the set of edges. EAOSDG contains the trivial edges such as *control* edge, *data* edge, *call* edge, *parameter-in/out* edge etc., as discussed in Chapter 2. Some new types of edges (such as *weaving edges*) are used to connect the aspect and non-aspect parts of the program. The *weaving edge* represents the dependency between the aspect and non-aspect parts of the program.

Let us consider the sample program shown in Figure 4.3. We have used AspectJ for writing the program, because AspectJ is a widely used programming language for developing aspect-oriented programs. The sample program prints the list of prime numbers upto the specified integer given by the user. The sample program consists of 2 parts; aspect part and non-aspect part. The aspect part has two pointcuts; one for capturing the call of prime() method and another to capture the use of variable *n*. The EAOSDG of the sample program, given in Figure 4.3, is shown in Figure 4.4. Each node $n \in V$ corresponds to the bytecode version of the statements of the AOP written in AspectJ [1]. First, the SDG for non-aspect part is constructed. Then, for representing the aspect part ADG is constructed. The EAOSDG is the combination of these two types of dependence graphs. *Weaving* edges are used to combine different SDGs and ADGs and construct EAOSDG. The EAOSDG shown in Figure 4.4 contains the following set of edges already defined in Chapter 2 (Background). Some extra edges are used that are defined below:

Definition 4.1. ***Class Membership Edge*****:** The class membership edge, $n_1 \overset{class}{\rightarrow} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$ such that n_2 is either an attribute or operation of the class node n_1.

Definition 4.2. ***Weave-In*****:** Weave-In edge, $n_1 \overset{Weave-in}{\rightarrow} n_2 \in E$, connects the non-aspect part with the aspect part of EAOSDG.

Definition 4.3. ***Weave-Out*****:** Weave-Out edge, $n_1 \overset{Weave-out}{\rightarrow} n_2 \in E$, connects the aspect part with the non-aspect part of EAOSDG.

4.2.1 EAOSDG Construction Algorithm

EAOSDG is generated using a series of steps as explained in Algorithm 2. First the nodes are created for each statement of the program. For a method call statement, one or more extra nodes are created to represent the *actual-in* and *actual-out* parameters. Similarly, for a method entry node, appropriate number of nodes are created to represent the *formal-in* and *formal-out* parameters. After, creation of nodes, *control dependence* and *class membership edges* are added depending on the nodes and the respective usages of the edges. Then, the *call edges* are added between the calling node and method entry node of corresponding callee method. Subsequently, *param-in/param-out edges* are added between *actual and formal* parameters. After that, the transitive dependency between *parameter-out* and *parameter-in* nodes are identified, a *summary* edge is draw between these nodes. Similarly, the creation

[1]eclipse.org/aspectj/

of SDG, we want to represent aspect-part as ADG. The pointcut nodes for the aspect part of the programs, the *weaving (In/Out) edges* are added to connect the aspect and non-aspect part of the program.

Algorithm 2 EAOSDG Generation Algorithm

INPUT: Aspect-oriented program.
OUTPUT: An EAOSDG $G < V, E >$.

```
1: Create individual nodes for each statement of the programs.
     • If the node is a method node, then add actual-in and actual-out nodes.
     • If it is a method entry node, then create formal-in and formal-out nodes.
2: Add Control Dependency, Data Dependency, and Class Membership Edge in between
   nodes by analysing the programs.
     • Add a Control Dependent edge, $n_1 \overset{cd}{\to} n_2$, if $n_1$ transfers the control to $n_2$.
     • Add a Data Dependent edge, $n_1 \overset{dd}{\to} n_2$, if $n_2$ is data dependent on $n_1$.
     • Add Class Membership edge, $n_1 \overset{class}{\to} n_2$, if $n_2$ is either an attribute or operation of
       the class node ($n_1$).
3: Add Call Edges and Param-In/Param-Out Edges between the nodes in the graph.
     • Add a Call edge, $n_1 \overset{call}{\to} n_2$, if $n_2$ is the method declaration node and $n_1$ is the
       corresponding method calling node.
     • Add a Param-In edge, $n_1 \overset{P_{in}}{\to} n_2$, if $n_1$ is the actual parameter and $n_2$ is the formal
       parameter.
     • Add a Param-Out edge, $n_1 \overset{P_{out}}{\to} n_2$, if $n_1$ is the return node and $n_2$ is the node
       accepting the value.
4: Add Summary Edge between nodes if the Param-Out node is transitively dependent on
   the Param-In node.
5: Create nodes for statements pointcuts in the Aspect part of the program.
     • Add call edges between pointcut nodes and advices.
6: Add Weaving Edge to connect the Aspect and Non-Aspect part of the program.
     • Add a weave-Out edge, $n_1 \overset{Weave-Out}{\to} n_2$, if $n_1$ is the before advice node and $n_2$ is
       the corresponding method entry node.
     • Add a weaving edge, $n_1 \overset{Weave-In}{\to} n_2$, if $n_2$ is the after advice node and $n_1$ is the
       corresponding method entry node.
```

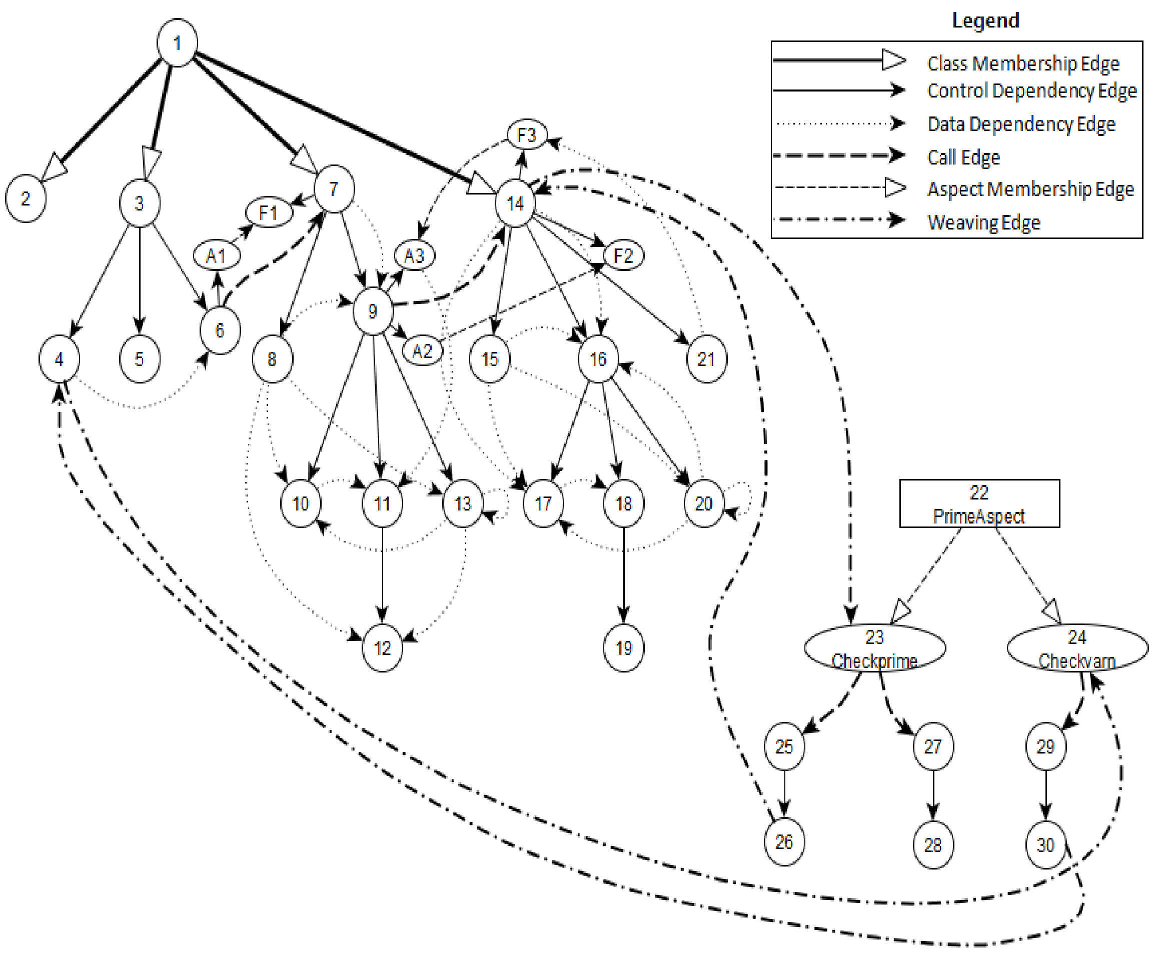


Figure 4.4: EAOSDG of the aspect-oriented program given in Figure 4.3

4.3 Proposed Algorithm 1: Extended Two-Phase slicing algorithm

In this section, we discuss the proposed approach to compute static slices of AOPs based on the intermediate graph Extended Aspect-Oriented System Dependence Graph (EAOSDG). The procedure to construct EAOSDG is presented in Section 4.2. Below, we present our slicing algorithm and then we have explained its working with suitable example.

4.3.1 Proposed Algorithm

The two-phase slicing algorithm given by Horwitz [6] and used by Zhao[91] cannot handle the aspect part of the program properly. The two-phase slicing algorithm just backward traverses the SDG in two different phases which arguably handles the procedural and object-oriented parts of the program respectively. The aspect part of the program is not handled properly by this algorithm.

We extend the two-phase slicing algorithm of Horwitz [6] by adding one more phase, to enable slicing of bytecode based graphs. This algorithm finds a static slice for a given slicing criterion ‘s’, which comprises of that program statements that affect the value of the slicing criterion. The extended two-phase algorithm works in three steps, as given below:

- Phase 1: In the first phase, the algorithm traverses backward, taking into consideration the slicing criterion, along all edges except *parameter-out* edges and *weaving* edges, and marks that vertices in EAOSDG that are reached during the first phase of traversal.
- Phase 2: In the second phase, the algorithm traverses backward from all the vertices that were marked during the first phase along all edges except *call*, *parameter-in* and *weaving* edges and marks the reached vertices in the EAOSDG.
- Phase 3: In the third and last phase, this algorithm traverses backward from all the vertices which were marked by the first and second phases, along the *weaving* edges to reach the aspect part of the program.

The final slice is the union of all the vertices marked during the phase 1, phase 2 and phase 3 traversal of EAOSDG.

4.3.2 Working of Algorithm

For explaining the working of our algorithm, we have considered the example program given in Figure 4.3. The generated EAOSDG for the example program is shown in Figure 4.4. The disadvantage of the extended two-phase algorithm is that it will not work for slicing node present in the aspect part of the EAOSDG. Now, suppose we consider statement *19*, which represents the statement $'return(false)'$, as the slicing criterion. We use three worklists, i.e.,

Algorithm 3 Extended Two-Phase Slicing Algorithm

INPUT: An EAOSDG $G < V, E >$, a slicing criterion s.
OUTPUT: The Slice S for the slicing criterion s,
INITIALISE: $W_1 = \{s\}, W_2 = \{\}, W_3 = \{\}, S = \{s\}$.

```
1:  while $W_1 != \phi$ do                                  ▷ phase 1
2:      $W_1 = W_1 - \{n\}$                     ▷ process the next node in $W_1$
3:      for all $m \rightarrow n$ do                 ▷ handle all incoming edges of n
4:          if $m \notin S$ then
5:              $S = S + \{m\}$
6:              if $e \notin \{p_o, weav\}$ then    ▷ if e is not a parameter-out or weaving edge
7:                  $W_1 = W_1 + \{m\}$
8:              else if $e \in \{p_o\}$ then
9:                  $W_2 = W_2 + \{m\}$
10:             else
11:                 $W_3 = W_3 + \{m\}$
12:             end if
13:         end if
14:     end for
15:     for all $n \rightarrow m$ do
16:         if $m \notin S \&\& e \in \{weav\}$ then          ▷ if e is an outgoing weaving edge
17:             $W_3 = W_3 + \{m\}$
18:         end if
19:     end for
20: end while
21: while $W_2 != \phi$ do                                  ▷ phase 2
22:     $W_2 = W_2 - \{n\}$                     ▷ process the next node in $W_2$
23:     for all $m \rightarrow n$ do                 ▷ handle all incoming edges of n
24:         if $m \notin S$ then
25:             $S = S + \{m\}$
26:             if $e \notin \{p_i, call, weav\}$ then ▷ if e is not a parameter-in, call or weaving edge
27:                 $W_2 = W_2 + \{m\}$
28:             else if $e \in weav$ then
29:                 $W_3 = W_3 + \{m\}$
30:             end if
31:         end if
32:     end for
33:     for all $n \rightarrow m$ do
34:         if $m \notin S \&\& e \in \{weav\}$ then          ▷ if e is an outgoing weaving edge
35:             $W_3 = W_3 + \{m\}$
36:         end if
37:     end for
38: end while
39: while $W_3 != \phi$ do                                  ▷ phase 3
40:     $W_3 = W_3 - \{n\}$                     ▷ process the next node in $W_3$
41:     for all $m \rightarrow n$ do                 ▷ handle all incoming edges of n
42:         if $m \notin S$ then
43:             if $e \notin \{p_i, call\}$ then          ▷ if e is not a parameter-in or call edge
```

```
44:                 $S = S + \{m\}$
45:                 $W_3 = W_3 + \{m\}$
46:             end if
47:         end if
48:     end for
49: end while
        return $S$
```

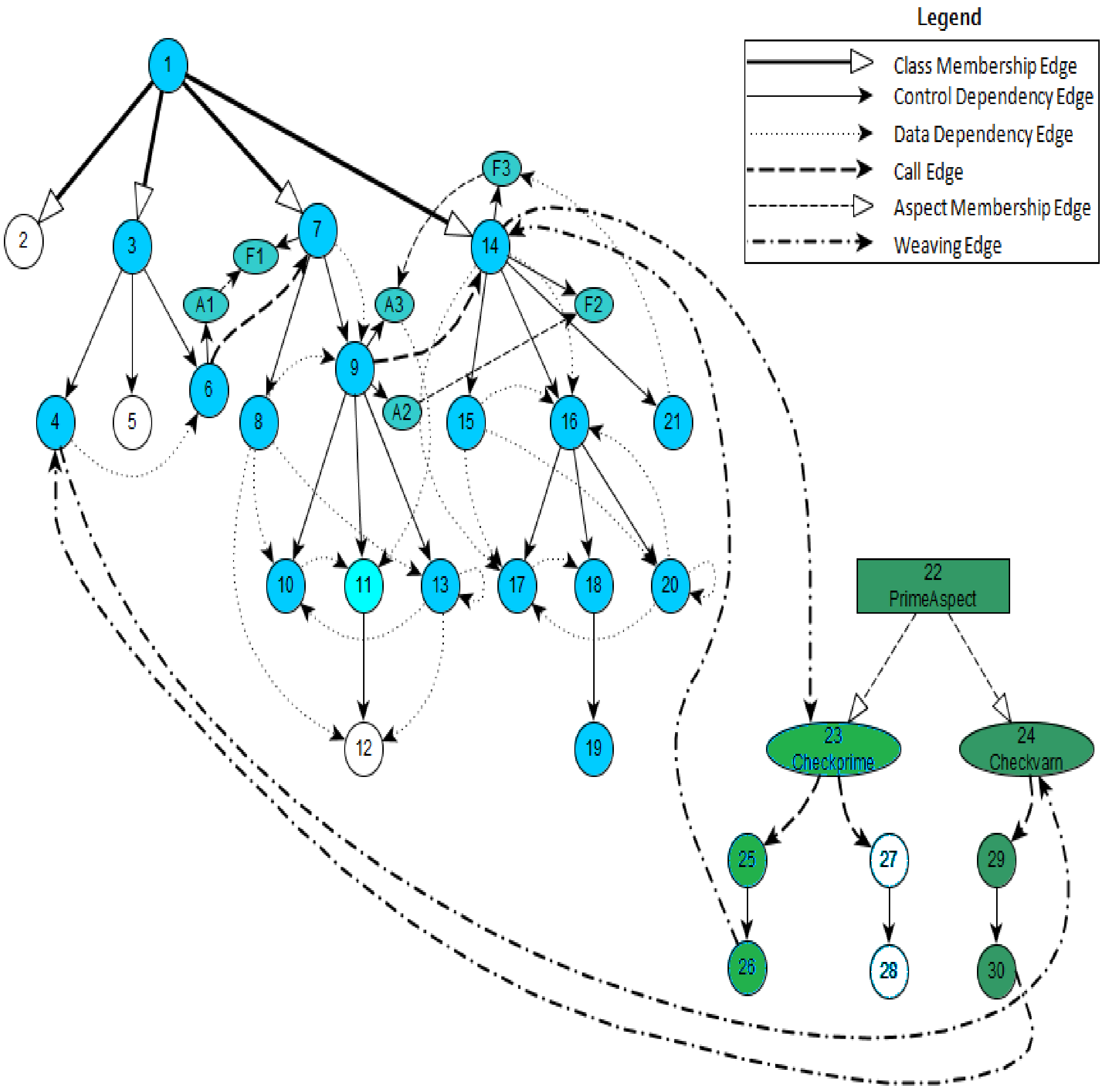


Figure 4.5: Sliced EAOSDG of the example program given in Figure 4.3 w.r.t. slicing criterion $< 19 >$

W_1, W_2 and W_3, in the three phases respectively. Thus the initial state of the data structures used in our approach, is given as follows:
$S = \{19\}$
$W_1 = \{19\}$
$W_2 = \phi$
$W_3 = \phi$

In Phase 1, we pop one node at a time from W_1. If it is not present before, then we add that node into S. Here, we check for all incoming edges to the current node. Then, we add the source nodes of these edges into W_2, if the edge is a *parameter-out* edge. We will use this worklist W_2 in Phase 2 of our approach. If the edge is a *weaving* edge, then we add the source node into worklist W_3, that will be used in Phase 3 of our approach. Else, we put the source node into W_1 itself. Then, we check for the outgoing *weaving* edges from the popped node and add the destination nodes of that edges into worklist W_3. This process is continued till W_1 is empty.
After phase 1 we have:
S = {19,18,17,16,15,14,20,F2,F3,21,A2,7,F1,A1,1}
$W_1 = \phi$
$W_2 =$ {A3,9,6}
$W_3 =$ {14,4}

In phase 2, we pop one node from W_2, add the node into S (if it is not present before) and check for all incoming edges to the current node. If the edge is a *weaving* edge, then add the source nodes of these edges into worklist W_3. Otherwise, we check whether the edge is a *parameter-in/call* edge or not. If it is not a *parameter-in/call* edge, then we add the source node into worklist W_2. Then, we check for the outgoing *weaving* edges from the popped node and add the destination nodes of that edges into W_3. This process is repeated till W_2 is empty. *We show the newly added nodes into the slice in the present phase as bold faces.*
After phase 2, we have:
S = {19,18,17,16,15,14,20,F2,F3,21,A2,7,F1,A1,1, A3,9,8,10,13,6,4,3}
$W_1 = \phi$
$W_2 = \phi$
$W_3 =$ {14,4}

In phase 3, we pop one node from worklist W_3, and add the node into S (if it is not present before) and check for all incoming edges onto the present node. If the edge is not a *call* edge or *parameter-in* edge, the source node is added into worklist W_3. Similar process is carried out till W_3 becomes empty.
After phase 3, we have:
S = {19,18,17,16,15,14,20,F2,F3,21,A2,7,F1,A1,1, A3,9,8,10,13,6,4,3,30,29,24,26,25, 28,27,23,22}
$W_1 = \phi$

$W_2 = \phi$
$W_3 = \phi$

Hence, for the given slicing criterion, 19, the computed slice is computed as follows:
$S = \{$19,18,17,16,15,14,20,F2,F3,21,A2,7,F1,A1,1, A3,9,8,10,13,6,4,3,30,29,24,26,25, 28,27,23,22$\}$
The marked nodes that comprise the slice are shown as shaded nodes in Figure 4.5.

4.3.3 Correctness of Extended Two-Phase Slicing Algorithm

In this section, we sketch the proof of correctness of our extended two-phase slicing algorithm.

Theorem 4.4. *Extended Two-Phase slicing algorithm aways finds correct slice with respect to a given slicing criterion.*

Proof. An algorithm is said to be correct if it satisfy the properties of completeness, correctness, and finiteness. First, we show that our proposed extended two-phase slicing algorithm is complete. During computation of slices of AOPs, we construct EAOSDG for the given AOP. EAOSDG contains following types of edges: *class*, *control*, *data*, *call*, and *weaving*. In phase-1 of our algorithm, we start backward traversal from a given slicing criterion. During the phase-1 of extended two-phase slicing algorithm, when we find any *class dependence* edge , *control dependence* edge or *data dependence* edge, we add the new source nodes into *slice*. Apart from these edges, if we find any *parameter-in* edge or *call* edge then we add the new source nodes in *slice*. When we find any *parameter-out* edge or *weaving* edge, we add the new nodes in the respective worklists for future processing. After phase-1 is over, in the phase-2 our algorithm process all *parameter-out* edges found during the phase-1 traversal of EAOSDG. Similarly, in phase-3, all *weaving* edges are processed. In our proposed slicing algorithm all types of edges of an EAOSDG are processed one after other in three different phases. Hence, the extended two-phase algorithm is complete.

To proof that our proposed algorithm computes correct slices, we assume that the given EAOSDG is correct. Now, suppose s is the slicing node in a given EAOSDG, if there exist any type of dependency of s on another node p of EAOSDG, then there must be an edge from p to s. While computing slice, we start with the slicing node s, as there is an incoming edge from p to s we add the new node p in the *slice* and similarly move backward traversing all incoming edges in an EAOSDG. In our proposed slicing algorithm, we handle all types of incoming edges while traversing EAOSDG from s, in three phases. If any dependency exist between node s and node p and s is already inserted in *slice*, then only the node p will be added in *slice*. So, we can ensure that our proposed slicing algorithm compute correct slices for a given correct EAOSDG.

Our proposed slicing algorithm is based on a worklist W which holds nodes of a given EAOSDG. First, we add the slicing node in the worklist W and repeatedly delete nodes from W and add new dependent nodes in W. The algorithm iterates till the worklist W not becoming empty. As the number of nodes and edges in an EAOSDG is finite, the worklist W will become empty after finite number of iterations. Hence, we can say that our proposed slicing algorithm terminates after finite iterations in finite time. □

4.3.4 Complexity Analysis

In the following we discuss the space and time complexity of the extended two-phase slicing algorithm.

Space complexity: The EAOSDG is a graph stored using a modified adjacency list, which has nodes and edges as objects of different classes. For each statement of the given program, one node is created in its EAOSDG. If the number of statements in a given AOP is n, then its EAOSDG must have atleast n nodes. Then nodes in EAOSDG may be more than n because some extra nodes are created for showing *actual-in*, *actual-out*, *formal-n*, and *formal-out*. But, these extra nodes are little in comparison to nodes in a large EAOSDG, hence can be neglected. Since the number of nodes in the graph is n and the number of edges is e, the space complexity of storing the graph is of order $O(ne)$.

Time complexity: Extended two-phase algorithm has three phases. For each phase, there is an inner and an outer loop. Let the number of edges in the EAOSDG is e and number of nodes be n, then the inner loop runs for e times and the outer loop runs for n times in the worst case scenario. As all the phases have the same number of iterations, the worst case time complexity of the extended two-phase algorithm is $O(ne)$.

The extended two-phase algorithm computes static slicing using EAOSDG. The dynamic slices are very useful in testing, debugging, reverse engineering etc. and smaller in size. So, we want to develop a dynamic slicing algorithm for AOPs. In the next section, we have proposed a dynamic slicing algorithm.

4.4 Proposed Algorithm 2: Context-Sensitive Dynamic Slicing Algorithm

As discussed above, the dynamic slices are more useful than static slices. In this section, we present a dynamic slicing algorithm for AOPs. For computing dynamic slices for a given program, first we have to construct the dependence graph dynamically. So, we have proposed an algorithm for dynamic construction of EAOSDG for a given AOP. Also, we want to develop a context-sensitive slicing algorithm for AOPs for which we add labels on each edge of EAOSDG except *data dependence* edges and *control dependence* edges. Next, we present the dynamic EAOSDG construction algorithm.

4.4.1 Dynamic EAOSDG Construction

Before implementation of context-sensitive slicing algorithm, we need to know that call-site [16] of each edge except data and control dependence edges. Our proposed *Dynamic EAOSDG Construction Algorithm* (i.e. Algorithm 4) first calls *Control Dependence Algorithm (CDA)* for creation of nodes in EAOSDG and drawing control dependence edges between the nodes. Then *Dynamic Data Dependence Algorithm (DDDA)* is called to analyse run-time data dependencies between nodes of an EAOSDG and draw corresponding data dependence edges. Also, the *call* edges, *weave-in/out* edges are labeled with the call-sites, from where they are linked. Labeling of edges is required for or proposed context-sensitive slicing algorithm.

Algorithm 4 Dynamic EAOSDG Construction Algorithm

```
1: INPUT:  P- Input AOP
2: OUTPUT: $EAOSDG$- Modified EAOSDG
3: Invoke Control Dependence Algorithm (CDA)
4: Invoke Dynamic Data Dependence Algorithm (DDDA)
```

Control Dependence Algorithm (CDA)

The pseudo code for the *Control Dependence Algorithm* (CDA) is given in Algorithm 5. CDA takes the AOP as input and creates nodes for each statement of input AOP. Then it identify the control dependencies between the statements of AOP and draw *control* dependence edges between appropriate node of EAOSDG. After adding all control dependence edges it adds *call* edges, *parameter-in* edges, and *parameter-out* edges between appropriate call nodes and method entry nodes of EAOSDG. Then to maintain call-site of each method call, all the *call* edges, *parameter-in* edges, and *parameter-out* edges are labeled. After adding all method call related edges, weaving edges are added. The weaving edges are added between non-aspect part of the EAOSDG and aspect part of EAOSDG. Appropriate labels are given to each weaving edges. The EAOSDG constructed by CDA does not contains data dependence edges, so the DDDA is called for adding runtime data dependencies between nodes of EAOSDG.

Dynamic Data Dependence Algorithm (DDDA)

In Algorithm 6, we present the pseudo code of *Dynamic Data Dependence Algorithm* (DDDA). DDDA is responsible to identify the runtime data dependencies between statements of AOP and draw appropriate *data dependence* edges between nodes of EAOSDG.

Algorithm 5 Control Dependence Algorithm (CDA)

```
1: INPUT:  P- Input AOP
2: OUTPUT: $EAOSDG$- Modified EAOSDG suitable for computing context-sensitive
   slice
3: for each executable statement or predicate $\in$ P do
4:     Create a node in the EAOSDG
5:     Create separate call nodes and actual-in/out nodes for each call sites
6:     Create method entry nodes and formal-in/out nodes for each method entry nodes
7: end for
8: \\ Add control dependence edges
9: if node $j$ controls the execution of another node $k$ then
10:     add control dependence edge $j \to k$
11: end if
12: \\ Add call dependence edges
13: if node $n$ is a call node and $m$ is the entry node for the method called at $n$ then
14:     add call dependence edge $n \to m$
15:     insert a label $n$ on the call edge, i.e. $n \overset{n}{\to} m$
16:     add parameter_in edges between actual-in and formal-in nodes
17:     add parameter_out edge between formal-out and actual-out nodes
18:     insert label $n$ on each parameter edge
19: end if
20: \\ Add weaving edges
21: if $v$ is the entry node of an advice target method and $w$ is the starting node of before
    advice then
22:     add weaving edge $v \to w$
23:     insert a label $v$ on the weaving edge, i.e. $v \overset{v}{\to} w$
24: end if
25: if node $x$ is the last node of before advice then
26:     add weaving edge $x \to v$
27:     insert a label $v$ on the weaving edge, i.e. $x \overset{v}{\to} v$
28: end if
29: if node $c$ is the last node of an advice target method and $d$ is the starting node of after
    advice then
30:     add weaving edge $c \to d$
31:     insert a label $c$ on the weaving edge, i.e. $c \overset{c}{\to} d$
32: end if
33: if node $g$ is the last node of after advice then
34:     add weaving edge $g \to c$
35:     insert a label $c$ on the weaving edge, i.e. $c \overset{c}{\to} c$
36: end if
37: if node $a$ is the call node of an advice target method and $b$ is the starting node of around
    advice then
38:     add weaving edge $a \to b$
39:     insert a label $a$ on the weaving edge, i.e. $a \overset{a}{\to} b$
40: end if
41: if node $c$ is the last node of around advice then
42:     add weaving edge $c \to a$
43:     insert a label $a$ on the weaving edge, i.e. $c \overset{a}{\to} a$
44: end if
```

```
45: if node $p$ is a proceed() node inside around advice then
46:     add weaving edge between $p$ to the target method entry node
47:     add another weaving edge between the last node of the target method and node $p$
48: end if
```

It first execute the given program by providing the input values to find the run-time data dependencies. The *input variables* are such variables in a program, whose values must be provided during run-time by the user, like command line arguments in Java. To capture these dynamic data dependencies, we execute the given AOP. During the execution, DDDA maintains a variable $C(v)$ for each variables used in the program. As the program executes, $C(v)$ contains the recent statement number where the variable v is defined or changed. At the end, DDDA adds *data dependence* edges between the node represents $C(v)$ and nodes where the variable v is used. At the end of DDDA a dynamic EAOSDG is constructed.

We have constructed the dynamic EAOSDG for the example program given in Figure 4.3. First we have called CDA for constructing the EAOSDG's node and adding *control* dependence edge, *call* edges, *parameter-in* edges, *parameter-out* edges, and *weaving* edges. Then, DDDA is called for adding dynamic *data dependence* edge in EAOSDG. Dynamic EAOSDG for the example program given in Figure 4.3 is shown in Figure 4.6.

Algorithm 6 Dynamic Data Dependence Algorithm (DDDA)

```
1: INPUT: P- Input program
2:           I- Values for input program variables
3: OUTPUT: $DynamicEAOSDG$
4: for each variable $v \in$ P do
5:     initialize $C(v) = \phi$
6: end for
7: while (! P terminate) do
8:     for each statement $s \in$ P do
9:         execute statement $s$ of P associated with I
10:        for each variable $v \in s$ do
11:            if $v$ is defined at $s$ then
12:                $C(v) = s$
13:            end if
14:            if $v$ is used at $s$ AND $C(v) \neq \phi$ then
15:                add a data dependence edge $C(v) \rightarrow s$ to $MAODG_P$
16:            end if
17:        end for
18:    end for
19: end while
```

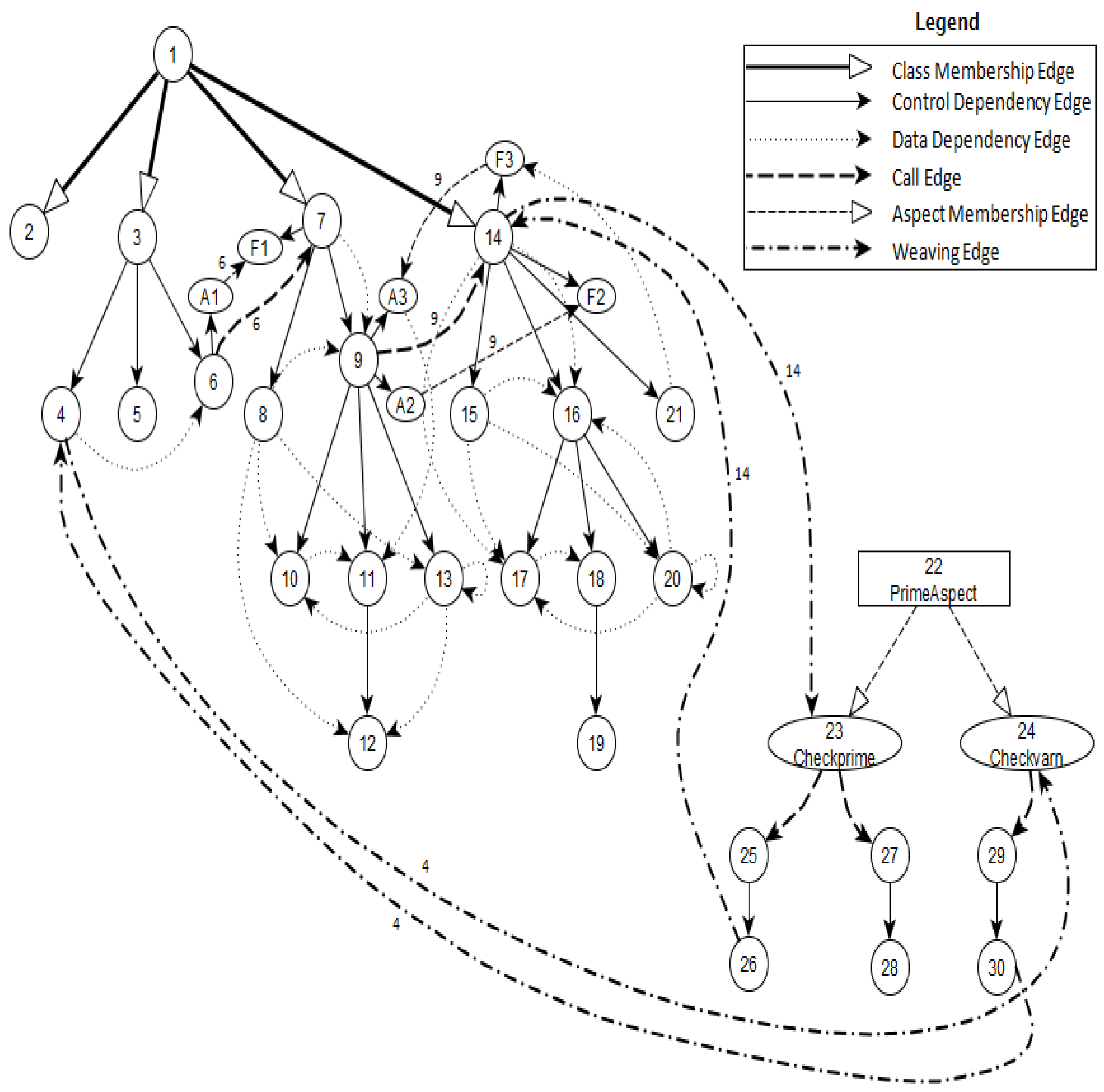


Figure 4.6: Dynamic EAOSDG of the example program given in Figure 4.3 that displays call-sites

Theorem 4.5. *Dynamic Data Dependence Algorithm (DDDA) finds correct data dependence in a given program.*

Proof: In this section, we sketch the correctness proof of DDDA. The main properties of an algorithm are finiteness, effectiveness, and termination. DDDA consists of mainly two loops. The number of iterations in first for loop is the number of variables in a program. In a program, the number of variables is fixed and hence, the for loop will iterate for a fixed number of times. The second loop is a while loop, which iterates for each statement of the program. As the number of statements in a program is finite, so the while loop will have finite number of iterations. This proves that DDDA will terminate after a finite number of iterations. Suppose $v_1, v_2, ...v_i, ...v_n$ are the variables present in the input program. DDDA first assigns a cache $C(v_i)$ for each variable v_i and initializes it with ϕ. Before the use of a variable, it must be defined. Let the statement p defines a variable v_i, so DDA will change the cache value of the variable as $C(v_i) = p$. After execution of some statement of the program, suppose at statement q, variable v_i is used. Then there exists a data dependence between the statement where v_i is defined and the current statement q which uses v_i. In DDDA, a data dependence edge is drawn between the node representing $C(v_i)$ and the node representing statement q. So, it proves that DDDA is effective to handle data dependencies present in a program. Finally, as the algorithm has two loops and both of them are finite, so after finite number of iterations, the algorithm will terminate. This proves that DDDA terminates within finite time. $\square$

4.4.2 Proposed Slicing Algorithm

The proposed context-sensitive slicing algorithm is responsible for maintaining context-sensitivity during computation of slices. It takes the EAOSDG of an AOP, as input. It produces a list L which includes all the nodes included in the computed slice, as output. It maintains a stack, i.e. SC, for preserving call-context. The algorithm is based on a worklist. First, the worklist W is initialised with the slicing criterion node ‘s’. The slicing algorithm runs until the worklist W becomes empty. Repeatedly one node is removed from W and added into L and all the incoming edges to this node are examined.

Let *cs* be the label of one incoming edge into the slicing criterion node ‘s’. First the algorithm checks for all incoming edges. If the current edge is a call or parameter in edge, then the algorithm checks the status of the corresponding call-context stack SC. If SC is empty, then the source node of the current edge is inserted into worklist W. If SC is not empty, then the top element of SC is fetch and matched with the label of current edge, i.e. *cs*. If both of them match, then only the source node of the current edge is inserted into W and the top element of SC is removed. If the current edge is found to be a parameter out edge, then the source node of the edge is inserted into W and SC. Similarly, the weaving edges are handled by the context-sensitive slicing algorithm.

4.4.3 Working of Algorithm

Let us take node 19 as the slicing criterion, as shown in Figure 4.6. Initial values of $W = \{19\}, SC = \{\}$ and $L = \{\}$. Now, let us start traversing backward from node 19. The status of various data structures when processing different nodes during construction of the slice is given in Table-4.1. During traversal, first we reached at node 18 and the connecting edge from node 19 to node 18 is a control dependence edge. As it is not a call, or weaving edge, so no stack will be used, only we have to add node 18 into W. In the next step, node 18 is removed from W and added into L and its incoming edges are checked. One control dependence edge (from node 16) and one data dependence edge (from node 17) are found. Both the new nodes associated with these two edges must be added to W, as shown in the

Algorithm 7 Context-Sensitive Slicing Algorithm

```
1: INPUT:  $EAOSDG$ = (V,E)
2:         Slicing criterion s, s$\in$ V
3: OUTPUT:  L- the list of nodes in the computed slice for s
4: W = $\{s\}$ // initialize the worklist with s
5: L = $\{\}$ // the slice set
6: SC = $\{\}$ // the stack for maintaining context-sensitivity
7: repeat
8:     W= W $\setminus \{n\}$
9:     L= L $\bigcup \{n\}$
10:    for all $m \xrightarrow[e]{} n$ do // handle all incoming edges of $n$
11:        Let $CS_e$ is the call-site of edge $e$
12:        if  $e \in \{parameter\_in, call, weave\_in\}$ && e has not been marked then
13:            if $SC[TOP] == \phi$ then
14:                W= W $\bigcup \{m\}$
15:                mark $e$
16:            end if
17:            if $SC[TOP] == CS_e$ then
18:                W= W $\bigcup \{m\}$
19:                mark $e$
20:                POP(SC)
21:            end if
22:        else if  $e \in \{parameter\_out, weave\_out\}$ && e has not been marked then
23:            W=W $\bigcup \{m\}$
24:            SC.PUSH(m)
25:            mark $e$
26:        else//intra-procedural edges
27:            if  e has not been marked  then
28:                W=W $\bigcup \{m\}$
29:                mark $e$
30:            end if
31:        end if
32:    end for
33: until W $== \phi$
```

Table 4.1: Status of data structures during working of CS slicing algorithm

W	Stack SC	L
19	-	-
18	-	19
17,16	-	19,18
16,14,15	-	19,18,17
14,15	-	19,18,17,16
15,9,1,26	9,14	19,18,17,16,14
9,1,26	9,14	19,18,17,16,14,15
1,26,7,8	9,14	19,18,17,16,14,15,9
26,7,8	9,14	19,18,17,16,14,15,9,1
7,8,25	9,14	19,18,17,16,14,15,9,1,26
8,25,6	9,14,6	19,18,17,16,14,15,9,1,26,7
25,6	9,14,6	19,18,17,16,14,15,9,1,26,7,8
6,23	9,14,6	19,18,17,16,14,15,9,1,26,7,8,25
23,3,4	9,14,6	19,18,17,16,14,15,9,1,26,7,8,25,6
....	.....	....

third row of Table-4.1. Similarly, we delete one node at a time from W and add to L and simultaneously change the status of stack SC. When W become empty, the algorithm stops and the list of nodes included in the slice are present in L.

4.4.4 Correctness of Context-Sensitive (CS) slicing algorithm

In this section, we sketch the proof of correctness of our context-sensitive slicing algorithm.

Theorem 4.6. *Context-Sensitive (CS) slicing algorithm aways finds correct slice with respect to a given slicing criterion.*

Proof. An algorithm is said to be correct if it satisfy the properties of completeness, correctness, and finiteness. The input to CS slicing algorithm is EAOSDG for an AOP. In EAOSDG, there are two categories of edges present, i.e. inter-procedural edges and intra-procedural edges. Our CS algorithm starts traversal from the given slicing criterion node and checks all incoming edges. If it finds, *call* edge, *parameter-in* edge or *weave-in* edge, then it fetch the top element from stack SC and matches it with the label on current edge. When there is a match, the new node is added into slice. During traversal of EAOSDG when it finds any *parameter-out* edge or *weave-out* edge, then the algorithm push the label on current edge in stack SC. For any other intra-procedural edges, the CS algorithm adds the unmarked node in slice. As, the CS algorithm handles all types of edges that may present in an EAOSDG, hence it is complete.

To prove that our proposed algorithm computes correct slices, we assume that the given EAOSDG is correct. Now, suppose s is the slicing criterion node in the given EAOSDG.

If there exists an edge from any other node p of the EAOSDG to node s, it means that node s depends on node p for its execution. If the edge between node p and node s is inter-procedural type edge, i.e. from one method to another method, then the algorithm uses stack SC to maintain calling context-sensitive during traversal. If the edge between node p and node s is an intra-procedural type edge, i.e. within the same method, such as *control dependence* edge or *data dependence* edge, then the new node p will be added to the slice. As, the context-sensitive property of slice computation is implemented through stack, it can be concluded that our slicing algorithm computes correct slices for AOPs.

Our proposed slicing algorithm is based on a worklist W which holds nodes of a given EAOSDG. First, we add the slicing node in the worklist W and repeatedly delete nodes from W and add new dependent nodes in W. The algorithm iterates till the worklist W not becoming empty. As the number of nodes and edges in an EAOSDG is finite, the worklist W will become empty after finite number of iterations. So, we can conclude that our CS slicing algorithm satisfy finiteness. This proves the Theorem. □

4.4.5 Complexity Analysis

In the following we discuss the space and time complexity of the Dynamic Context-Sensitive slicing algorithm.

Space complexity: The CS slicing algorithm works on the input EAOSDG, as in case of extended two-phase algorithm. In Section 4.3.4, we have already determined the space complexity of EAOSDG. Hence, if the number of nodes in the EAOSDG is n and the number of edges is e, the space complexity of storing the graph is of order $O(ne)$.

Time complexity: Dynamic context-sensitive slicing algorithm is based on insertion and deletion from a worklist W. For finding the worst case time complexity, suppose all the nodes of EAOSDG are connected. Let the number of nodes in EAOSDG in n. While processing any node s, we have to traverse rest $(n - 1)$ nodes. As a result the outer loop of CS slicing algorithm runs $n * (n - 1)$ times. So, the worst case time complexity of the dynamic context-sensitive slicing algorithm is $O(n^2)$.

4.5 Implementation and Results

In this section, we briefly describe the implementation of our proposed slicing approaches. The main motivation for our implementation is to verify the correctness and the preciseness of our algorithms. For construction of EAOSDG of a given AOP, we have developed a tool. First, we describe the overview of our developed tool. Then, we have applied our proposed slicing techniques on five open source projects using our slicing tool. Finally, we present the output of our experiment.

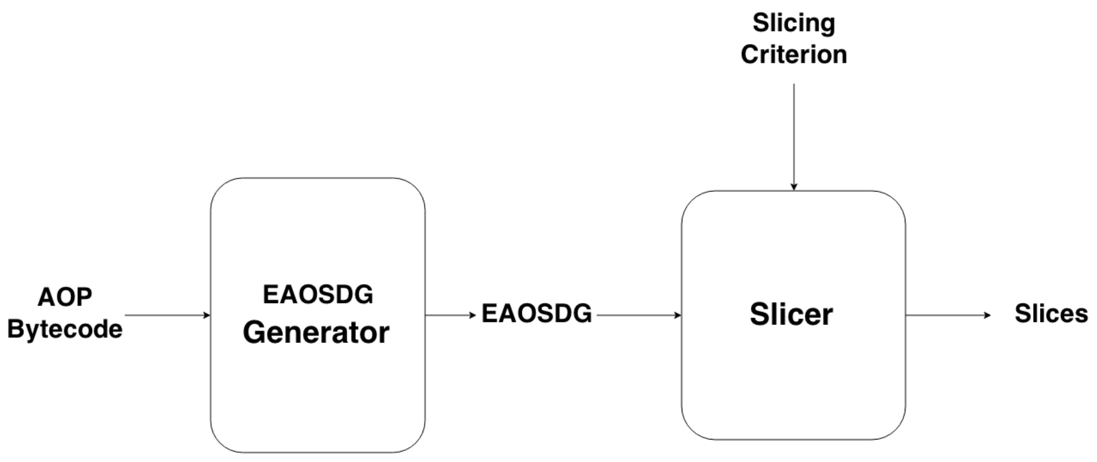


Figure 4.7: Framework of the tool

4.5.1 Overview of Tool

We developed a tool for automatic construction of EAOSDG for an input AOP and to compute slices using our proposed slicing algorithms. The framework of the tool is shown in Figure 4.7. Developed slicing tool has two parts: *EAOSDG Generator* and *Slicer*. Our tool takes the bytecode of an AOP. First, bytecode of the given AOP is given to *EAOSDG Generator* package. This package is an extension of *Java System Dependence Graph API* [2].

- **EAOSDG Generator :** This part of the tool extracts the information of all the classes and methods of the program from the bytecode and sends it for matrix generation. The different packages present in *EAOSDG generator*, are summarized in Table 4.2. The *internal* package analyses the byte code and generates information about instructions, methods, classes, aspects etc. The *graph.internal* package uses the fetched information by *internal* package and generates the data-structure for *method dependence graph (MDG)* and *aspect dependence graph (ADG)*. The *com.graph* package checks the information provided and finds the dependencies between different parts of the program. It maps all the dependencies and parameters of the program and then stores it according to the data structures defined by *com.util.datastructure* package. We have developed a package called *main* for collecting all information from different packages and construct the EAOSDG and display it using an image editor. The sketch of *main* package is shown in Figure 4.8. We have to provide the complete path of the input AOP bytecode in *inputFolderPath*. EAOSDG generator construct the graph and stores the graph in the folder whose path is given in *outFolderPath*.

[2]http://www4.comp.polyu.edu.hk/ cscllo/teaching/SDGAPI/

```
//package main;
import com.graph.jsdg.*;
import convertor.data_store;
import java.io.File;
import java.io.IOException;
import java.lang.Object;

public class Main {
  public static void main(String[] args) {
        try {
        String inputFolderPath="E:\\primenumber\\bin";
        String outFolderPath="E:\\Output";
        AspectSDGraph lvx = new AspectSDGraph(inputFolderPath);
    ...
    ...

    } catch (Exception ex) {
    ex.printStackTrace();
  }
}

}
```

Figure 4.8: Sketch of *main* package of EAOSDG Generator

- **Slicer :** After construction of EAOSDG for an input AOP, we want to compute slices using our proposed algorithms. We have developed a package *Slicer* that takes input the EAOSDG file and *slicing criterion* from the user.

4.5.2 Case Studies

We have downloaded five open-source programs for our experiment from the available open-source repositories. In the absence of adequate number of open-source aspect-oriented programs, some of the experimental programs (such as *Elevator* and *ATM Simulation*) are developed as laboratory assignments. We constructed different EAOSDGs for these programs and computed the time required by the tool to generate these EAOSDGs. Also, the number of nodes and edges generated in the respective EAOSDG are shown in Table 4.3. This table contains the details of five case study project such as, *Project name*, *project description*, *LOC*, details of EAOSDG, and EAOSDG generation time.

4.5.3 Experimental Results

In this chapter, we have proposed two slicing algorithms for AOPs, i.e. Extended Two-Phase slicing algorithm and Context-Sensitive slicing algorithm. We want to compare the performance of these two algorithms to know which algorithm gives more accurate results

Table 4.2: Package Description for our EAOSDG generation tool.

Package Name	**Usage**
com.asm.internal	This package is used for representing the internal classes which operate with ASM framework.
com.asm.internal.util	This package is used for storing the utility classes which operate with ASM framework.
com.graph	This package is used for storing the common attribute of a Graph.
com.graph.element	This package is used for storing the basic element of a Graph.
com.graph.internal	This package is used for storing the internal representation of a Graph.
com.graph.Iterator	This package is used for storing the different iterator for different searching algorithm.
com.graph.pdg	This package is used for storing the procedural dependence graph related things.
com.graph.sdg	This package is used for storing the system dependence graph related things.
com.util	This package is used for storing the common utility classes.
com.util.datastructure	This package is used for storing the common data structure classes.

and in little time. Based on the EAOSDGs generated for the different case study programs, slices are computed using the proposed slicing algorithms. Different number of slices for different programs are computed depending upon the input slicing criterion.

The details of the findings of our experiment is shown in Table 4.4. This table shows comparison of the two algorithms based on average slicing time. It is observed from Table 4.4 that Context-Sensitive (CS) slicing algorithm computes slices 4% to 37% faster than Extended Two-Phase slicing algorithm. Figure 4.9 shows the comparison of average slicing time vs LOC for each proposed algorithms. It can be observed from Figure 4.9 that Extended two-phase slicing algorithm always takes larger time to compute slices than CS slicing algorithm.

4.5.4 Threats to validity

Below, we present some of the threats to the validity of our proposed approach.

1. As the slicing technique proposed in this chapter is only for AspectJ platform, it may not work satisfactorily for other aspect-oriented programming languages such as Aspect C++ and Aspect C#.

2. Through the five case studies, we have tested our proposed slicing technique for computing precise and correct slices of projects upto 3856 LOCs. We believe, the

Table 4.3: Case study projects details

Sl. No	**Project Name**	**Project Description**	**LOC**	**No. of Nodes**	**No. of Edges**	**EAOSDG Generation Time**
1	Server - Client-1[3]	This project uses socket programming to create a server-client connection in between two systems	119	155	195	118 ms
2	Elevator-1	This project simulates elevator system. This version is an ordinary AspectJ program	540	583	997	302 ms
3	ATM Simulation	This project simulates the ATM system on a distributed environment	887	944	1650	1391 ms
4	Tetris Project	This is a very popular game, where we arrange blocks	1027	1566	2317	1672 ms
5	GoF Patterns-1	This is the AspectJ implementation of GoF design patterns	3856	4137	3752	2671 ms

Table 4.4: Comparative study of proposed slicing algorithms

Sl. No.	Project	LOC	No. of Slices	Average Slicing Time (in ms)	
				Ext. 2-Phase	CS
1	Server-Client-1	119	8	1.75	1.10
2	Elevator-1	540	14	2.48	1.82
3	ATM Simulation	887	20	4.73	4.51
4	Tetris Project	1027	23	5.02	4.73
5	GoF Patterns-1	3856	22	5.98	5.56

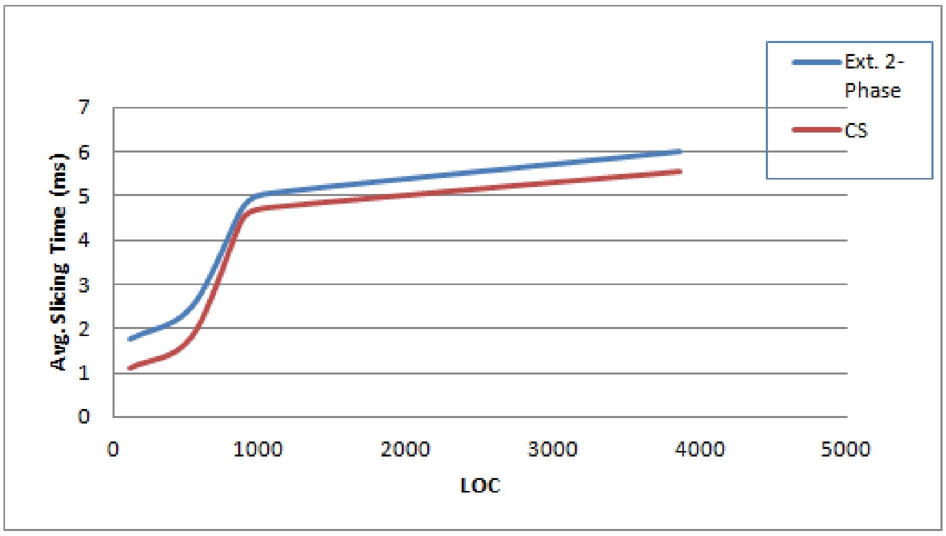


Figure 4.9: Comparison of performance

other larger projects can be handled through our slicing technique.

3. The proposed slicing technique is based on construction of EAOSDG, and if the intermediate representation changes, then our slicing technique may not work properly.

4.6 Comparison with related work

Most of the work in the existing literature manually generates the SDGs to compute the program slices, as they are silent about the graph generation process. Also, very little work has been reported in slicing of AOP. Zhao [91] for the first time computed the slices for AOPs. He has proposed an intermediate graph called *Aspect System Dependence Graph (ASDG)* to represent an AOP. ASDG cannot represent *around* advice of any AOP. In the absence of all features adequate number of different dependencies, the intermediate graph used to compute the slices do not correctly distinguish the aspect and non-aspect parts of the program.

Braak [92] also gave an approach for aspect slicing based on an intermediate graph that is also manually generated. Unlike the slicing algorithm extended by Zhao [91] and Braak [92], our proposed extended two-phase algorithm extends the slicing algorithm in [6] by introducing a third phase of traversal along the weaving edges. In the first two phases of the proposed slicing algorithm, we slice the non-aspect part of the input program and traverse the weaving edges in the third phase to slice the aspect parts.

Mohapatra et al. [71] has propose a *Trace file Based Dynamic Slicing* (TBDS) algorithm for AOPs. As the name suggest, this approach is based on execution trace based. Storing execution trace file is an overhead for slicing process. Also, needs to update trace file for each execution of the program. They have use an intermediate graph called Dynamic Aspect-oriented Dependence Graph (DAODG) for given AOP. The DAODG is not able to represent all features of AOP such as pointcuts and around advice. In our proposed approach, we have considered all these important features during computation of slices.

Ray et al. [27] have extended the work of Mohapatra et al. [71]. They have represented pointcuts in the intermediate graph. But, for each execution of the given program, a new graph has to build. This slows down the process of slicing. In our proposed slicing approach, we have used EAOSDG which is generated at the time of execution of given program. Hence, there is no extra time require to store execution trace and then construct the dependence graph.

4.7 Summary

In this chapter, we have first proposed an intermediate representation called Extended Aspect-Oriented System Dependence Graph (EAOSDG) for representing AOPs. Then, based on the EAOSDG we have proposed two slicing algorithm for AOPs, i.e. Extended Two-Phase slicing algorithm and Context-Sensitive (CS) slicing algorithm. We have developed a prototype tool for automatic generation and computation of slices for a given AOP. We have compared the performance of proposed slicing algorithms by taking five case studies. From the experiment, we found that, the context-sensitive slicing algorithm computes more precise slices in comparison to the other algorithm. Also, context-sensitive slicing algorithm takes significantly little computation time to generate the slices in comparison to the other slicing algorithm.

The algorithms presented in this chapter cannot handle concurrency issues in AOPs. In the next chapter, we extended our framework to consider concurrency issues in AOPs.

Chapter 5
Dynamic Slicing of Concurrent Aspect-Oriented Programs

A multithreaded program contains two or more parts that can run simultaneously or concurrently. Each such part of the program is called a thread. Multithreading is very useful in real-time programming and parallel computing [93]. A multithreaded program runs faster than a single-threaded program. The initiation of a thread can occur inside a program in any sequence, but the termination of these threads must follow the same sequence [72, 94]. Synchronization is the process to handle the creation and termination of threads in right sequence. We can maintain a separate module that handles all thread synchronization issues, so that overall complexity of the concurrent program will reduce [56]. But, the synchronization module is scattered through several related modules that invoke threads. When any module or concern is affecting many modules, then it is called cross-cutting concern. OOP is not efficient to handle the cross-cutting concerns in a software [91]. AOP is a new programming paradigm developed to handle these types of cross-cutting concerns [91]. When an AOP is developed to handle thread synchronization issues, then these types of AOPs are called concurrent AOPs (CAOP).

The computed slice of the CAOPs should be correct and precise. Finding a precise slice in a CAOP is very difficult. In the existing literature, we found that there are very few work done in the field of slicing of CAOP [14]. In this chapter, we propose a dynamic slicing algorithm for CAOPs using context sensitive approach. We first propose an intermediate graph called Multithreaded Aspect-Oriented Dependence Graph (MAODG) to represent CAOPs. MAODG represents the features of a CAOP. Then, we have proposed an algorithm to compute the dynamic context sensitive slice of CAOPs using MAODG. We have compared the performance of our slicing algorithm with the context-insensitive slicing algorithm and Ray et al.'s [14] algorithm. For comparison, we have considered slice size and slicing time.

This chapter is organized as follows:

In Section 5.2, we present the description of concurrency model of AOP, used in this thesis. Next, in Section 5.3 we present the details of our proposed intermediate representation, MAODG, for CAOPs. Then, in Section 5.4 our proposed context-sensitive dynamic slicing algorithm for CAOP is described. We have implemented our slicing approach and developed a concurrent AspectJ slicer. In Section 5.5, we discuss the details of our implementation and the results. In Section 5.6, we compare our proposed work with some existing related work. In Section 5.7, we summarize the chapter.

5.1 Basic Concepts

In this section, we present some basic concepts, which are most relevant and important for understanding our proposed technique. First, we briefly discuss the slicing technique for sequential programs and concurrent programs. Then, we present the concurrency model of AspectJ which is used in our proposed program slicing technique.

5.1.1 Slicing of Aspect-Oriented Programs without threads

All the existing slicing techniques are graph-based slicing techniques. Zhao [29] proposed a new intermediate representation for AOPs. He named this intermediate representation Aspect-oriented System Dependence Graph (ASDG), which consisted of three parts: an SDG for non-aspect part, a dependence graph for aspect part and some additional dependence edges to connect the SDG and the dependence graph of the aspect part. Trivial SDG consists of dependencies like control, data, call etc. The ASDG includes some additional types of edges apart from the types of edges present in SDG. Some of them are (i) aspect-membership edge, (ii) coordination edge etc. Aspect membership edges are drawn between the aspect declaration node and the pointcut nodes or advice nodes. The connection between the non-aspect SDG and the aspect graph is made through coordination edges. For example, the AOP shown in Figure 5.1 checks whether a given number is prime or not. In this program, the *before* advice prints the called method name before executing the method and the *after* advice prints ‘‘Returned Normally” after returning to the callee method. The ASDG for the example AOP is shown in Figure 5.2. *In the ASDG, the nodes represent the line numbers of the program*. Below, we give the formal definitions of some of the edges used in ASDG.

Now, for finding the slice for a given AO program, we need to apply a slicing algorithm on the generated ASDG. In our survey, we found that most of the research work including the work of Zhao [29], have used the two-phase slicing algorithm developed by Horwitz [6]. In the first phase, the algorithm starts with the given slicing criterion node and traverses backward along all the edges except parameter_out edges. All the nodes reached during this traversal are marked and included in the slice. Then, in the second phase, the algorithm traverses backward along all the edges except call and parameter_in edges. All the nodes reached during both the traversals are included in the slice. The union of nodes marked in phase one and phase two gives the slice of the given AO program with respect to a slicing criterion.

Taking node 23 of the ASDG in Figure 5.2, as the slicing criterion, we have computed

```
        primebetween.java

 import java.util.*;

1   public class primebetween {
2        private static int n;
3        public static void main(String [] args)
         {
4                Scanner sc = new Scanner(System.in);
5                n = sc.nextInt();
6                boolean t=prime(n);
7                if(t)
8                 System.out.println("Prime");
                 else
9                 System.out.println("Not Prime");
         }

10       public static boolean prime(int m)
         {
11          int count = 0;
12          for(int i=2; i<=m/2; i++)
             {
13            if(m%i==0)
14            count++;
             }
15        if(count==0)
16          return true;
          else
17          retrun false;}
        primeAspect.aj

18  public aspect primeAspect{
19  pointcut checkprime():
                call(boolean primebetween.prime(int));
20  before():checkprime()
         {
21       System.out.println("Prime method called");
         }
22  after()returning:checkprime()
         {
23       System.out.println("Returned Normally");
         }}
```

Figure 5.1: An example aspect-oriented program

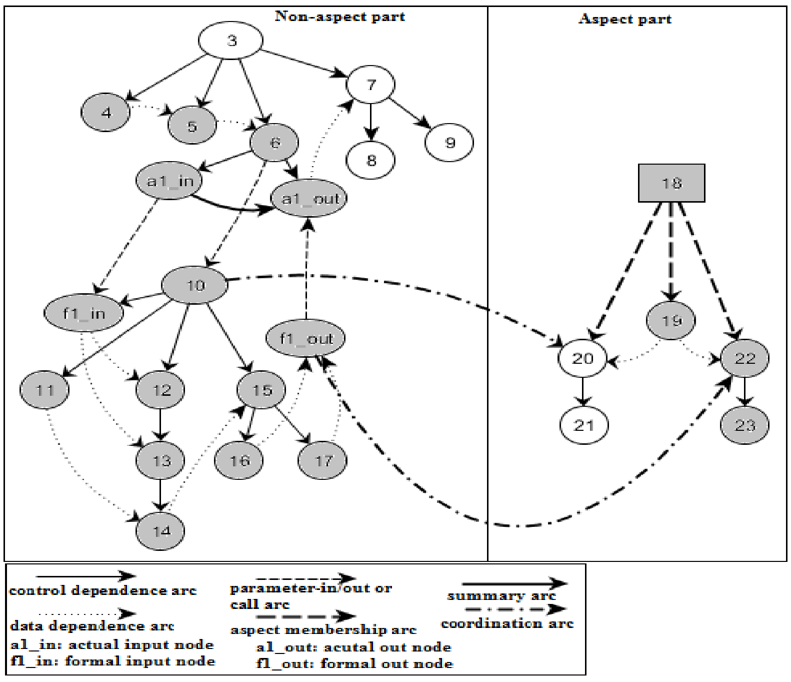


Figure 5.2: ASDG for the example aspect-oriented program given in Figure 5.1

the slice using the two-phase algorithm [6]. The resultant slice is shown in Figure 5.2, with shaded nodes included in the slice. Initially the worklists $W_1 = \{23\}, W_2 = \{\}$, and $S = \{23\}$. Phase 1 starts from the slicing criterion and traverses backward along all the edges except the parameter_out edges, while the source nodes of the encountered parameter_out nodes are saved in a worklist W_2. This phase visits the light gray nodes. Phase 2 starts from all the nodes present in the worklist W_2 and traverses backward along all the edges except call and parameter_in edges. Because of the summary edges, there is no need to return from a called procedure back to the callee. The resultant slice consists of all the nodes visited in phases 1 and 2.

Algorithm 8 : Two-phase slicing algorithm

INPUT: ASDG $G < V, E >$, a slicing criterion s.
OUTPUT: The Slice S for slicing criterion s.
INITIALISE: Worklists $W_1 = \{s\}, W_2 = \{\}, S = \{s\}$.

```
1:  while $W_1 ! = \Phi$ do                          ▷ phase 1 starts
2:      $W_1 = W_1 - \{n\}$                          ▷ process the next node in $W_1$
3:      for all edge $e$ between $m \to n$ do        ▷ handle all incoming edges of n
4:          if $e \notin \{p_o\}$ then               ▷ if e is not a parameter_out
5:              if $m \notin S$ then
6:                  $S = S + \{m\}$
7:                  $W_1 = W_1 + \{m\}$
8:              end if
9:          else $e \in \{p_o\}$
10:             $W_2 = W_2 + \{m\}$
11:         end if
12:     end for
13: end while
14: while $W_2 ! = \Phi$ do                          ▷ phase 2 starts
15:     $W_2 = W_2 - \{u\}$                          ▷ process the next node in $W_2$
16:     for all edge $e$ between $v \to u$ do        ▷ handle all incoming edges of u
17:         if $e \notin \{p_i, call\}$ then         ▷ if e is not a parameter_in or call edge
18:             if $v \notin S$ then
19:                 $S = S + \{v\}$
20:                 $W_2 = W_2 + \{v\}$
21:             end if
22:         end if
23:     end for
24: end while
```

5.1.2 Slicing of Aspect-Oriented Programs in presence of threads

Threads need particular attention during slicing. The presence of interference between threads in a program, makes the dependence relationship non-transitive. Hence, there cannot be any summary edges in the SDG for the concurrent programs [95]. As a result, when we use the two-phase algorithm to compute the slices of a concurrent program, it may generate incorrect and imprecise slices. So, we need to develop a new slicing algorithm that takes care of threads as well as the interference dependencies, if any, in CAOPs. We have designed a method to represent the interference dependency using a special edge in the graph called *Interference edge*. As the interference dependency arises during run-time and we cannot predict the run-time behaviour in advance, we add the interference edge for all possible scenarios during static analysis and graph generation. The construction of intermediate graph for CAOPs is presented in Section 5.3 and computing context-sensitive slices in the presence of threads, is explained in Section 5.4.

5.1.3 Concurrency model of AOP

There exists many programming languages for sequential implementation of AOPs. AspectJ is one of them. In AspectJ, the modularized cross-cutting concerns are weaved with the base code of the program to generate a sequentially executable program. Concurrency is an important feature to improve the performance of a program. In our literature survey, we found very few work [96–98] presenting models for concurrent AOP. Douence et al. [98] have proposed a Concurrent Event-based AOP (CEAOP) model which is the extension of sequential Event-based AOP (EAOP). In an EAOP, the aspects are defined in terms of events which occur during execution of a program. The CEAOP model supports concurrent aspects along with concurrent base program. As mentioned in [98], the CEAOP model is only an initiation of building a full concurrent model for AOP. Also, for the implementation of the proposed concurrent model, they have given only a JAVA prototype, which is little useful in developing large software.

The concurrent model proposed by Allan et al. [97] is based on the execution trace matching [99], for covering sequences of joinpoints. In their proposed model, there can be several sequences of joinpoints that run concurrently. But, while an advice executes, the base program is paused. So, this model can handle concurrent base programs, not concurrent aspects. Another concurrency model is proposed by Andrews [96] for AOP *reflection*. *Reflection* is the ability of a program to check and modify its own structure at run-time. In the proposed AOP reflection, Andrews has designed a concurrent model where the base program can run concurrently along with the metalevel code introduced through reflection. Their work only presents a sketch of the AOP reflection concurrent model. Also, their model is not directly related to the concurrent aspect model.

These existing models are not fully developed or evaluated models for CAOP. So, we have used a simple pointcut model of AspectJ and examined all possible designs of AOPs having threads. In our concurrent model, we have used the *Thread* class library provided by Java, as AspectJ is an extension of Java programming language. The standard operations on a thread such as *start()*, *stop()*, *join()*, *suspend()*, *resume()*, etc. are used along with AspectJ.

We have observed that only the presence of thread in AOP does not make it concurrent. An example program is shown in Figure 5.3. In this program, the *base* class calls two

```
public class base implements Runnable

 public void run()
  {
      for(int i=0;i<5;i++)
      System.out.println("Jai");
  }
 public static void main(String args[])
  {
      method1();
      method2();
  }
  public static void method1()
   {
      Thread th1=new Thread(new base());
      th1.start();
   }
  public static void method2()
   {
      Thread th2 =new Thread(new base());
      th2.start();
   }
  }
```

(a)

```
public aspect aspt {
 pointcut pc1():call(void base.method1());
  before():pc1()
   {
     for(int i=0;i<5;i++)
     System.out.println("thread1");
   }
 pointcut pc2():call(void base.method2());
  before():pc2()
   {
     for(int i=0;i<5;i++)
     System.out.println("thread2");
   }
 }
```

(b)

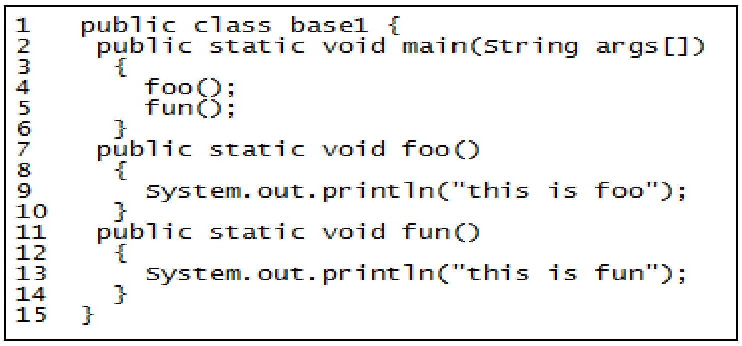
Figure 5.3: An example (a) concurrent base program with (b) sequential advices in an AspectJ program

methods and inside each method we create one thread. Within the thread we print *Jai* five times using a for loop. We have declared an aspect called *aspt* which contains two *before* advices for two methods of the *base* class. After executing this example program, we observed that when the thread is present in the base program, and the aspect part does not have any threads, then advices cannot run concurrently.

We found that when the advices contain threads, then only the advices can run concurrently. Even if the target methods in the base program do not contain threads and they execute sequentially, the advices that contain threads run concurrently, as shown in Figure 5.4. In this program, *base1* class has two methods that print the name of the method called. We have declared an aspect called *aspt1* that contains two *before* advices for the two methods of the *base1* class. These advices contain threads. Hence, we conclude that, when in an AOP, the advices contain threads, then it can be called as CAOP.

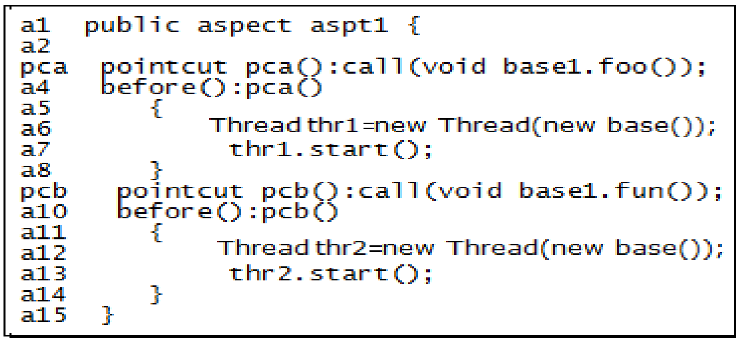
5.1.4 Slicing of Concurrent AOPs

In the literature, we have found a work done by Ray et al. [14] which is the closely related work with our proposed slicing technique. Ray et al. [14], have used an intermediate graph called Concurrent Aspect-oriented System Dependence Graph (CASDG) in their slicing technique. CASDG is created for each current execution trace, starting from scratch. Next



```
1    public class base1 {
2     public static void main(String args[])
3      {
4        foo();
5        fun();
6      }
7     public static void foo()
8      {
9        System.out.println("this is foo");
10     }
11    public static void fun()
12     {
13       System.out.println("this is fun");
14     }
15   }
```

(a)



```
a1   public aspect aspt1 {
a2
pca   pointcut pca():call(void base1.foo());
a4    before():pca()
a5       {
a6           Thread thr1=new Thread(new base());
a7            thr1.start();
a8       }
pcb    pointcut pcb():call(void base1.fun());
a10    before():pcb()
a11      {
a12          Thread thr2=new Thread(new base());
a13           thr2.start();
a14      }
a15   }
```

(b)

Figure 5.4: An example (a) sequential base program with (b) concurrent advices in an Aspect in an AspectJ program

time, for a different execution trace another new CASDG is formed. They [14] have extended the existing Node Marking Dynamic Slicing (NMDS) algorithm for OOPs, proposed by Mohapatra et al. [28] to concurrent AOPs. In this chapter, we have compared our proposed slicing algorithm with Ray et al.'s algorithm [14].

5.2 Multithreaded Aspect-Oriented Dependence Graph (MAODG)

We propose an intermediate graph named *Multithreaded Aspect-Oriented Dependence graph* (MAODG), to represent CAOPs. The MAODG is an extension of SDG (System Dependence Graph) for OOPs, proposed by Larsen and Harrold [10]. MAODG is a collection of method dependence graphs (MDG), thread dependence graphs (TDG) and aspect dependence graphs (ADG) and some special types of dependence edges. MDG is used to represent the methods in the program and intra-method control and data dependencies. TDG is used to represent each thread present in a concurrent AOP. ADG is used to represent the aspects, point-cuts and advices in an AOP. We have used several dependence edges to connect all the MDGs, TDGs, and ADGs. These dependencies are defined as follows:

Definition 5.1. Thread dependence edge: A thread dependence edge $s \xrightarrow{thread} t$ exists, if s

is a thread calling node and *t* is the entry node of its *run()* method. This is also termed as $thread\_start$ edge.

Also there is a thread dependency between the last statement of a thread and the immediate next statement of the main thread from where this thread is called. This is called $thread\_return$ edge. This dependency can be determined during run-time only.

Definition 5.2. Class/Aspect membership edge: Class membership edges are drawn between the class node and all its data member nodes and method declaration entry nodes. Aspect membership edges are drawn between aspect node and its pointcut nodes. For simplicity of the MAODG, we have used the same notation to represent both the Class and Aspect membership edges in MAODG, as shown in Figure 5.7. The dependence between the pointcuts and advices are represented by control dependence edge, as the execution an advice is controlled by its pointcut. Also, this will reduce the complexity of MAODG.

Definition 5.3. Weaving edge: A weaving edge $p \xrightarrow{weave} q$ exists, if p is a method call node and q is its corresponding advice definition start node. It is also known as $weave\_start$ edge. When a weaving edge is used to represent the return from an advice to its corresponding method call node, then this type of weaving edge is called $weave\_retrun$ edge.

Definition 5.4. Interference dependence edge: Suppose there are two nodes n_1 and n_2 of two different threads T_1 and T_2 respectively. If a variable v is defined at n_1 and used by n_2, and both the threads T_1 and T_2 execute in parallel, then an interference dependence edge $n_1 \xrightarrow{interference} n_2$ is drawn between node n_1 and node n_2.

5.2.1 Weaving the aspect and non-aspect parts in MAODG

The dependence graphs for non-aspect part of the AOP are constructed independently. Then, the ADG for aspect part of the AOP is created separately. Now, in-order to complete the MAODG construction, we need to add weaving edges between the SDG of non-aspect part and the ADG. According to the AspectJ program execution process, *the before advice runs just after the call statement and before the called method's body execute*. Therefore a weaving edge from call node in the SDG should be weaved with the entry node of the *before* advice in the ADG and then from last statement of *before* advice to the called method entry node. Similarly, in AspectJ, *the after advice runs just after the last statement of the called method and before the control is transferred to the callee method*. Hence, the weaving edge must be drawn between the last node of the called method and the entry node of *after* advice, and also between the last node of *after* advice and the call node.

5.2.2 MAODG Construction Algorithm

To construct the MAODG for a given CAOP, our algorithm first calls the CCA (Concurrent Control-dependence Algorithm) for constructing the MAODG without any data dependencies. This type of MAODG is incomplete and referred as partial MAODG and denoted by $MAODG_P$, because of the absence of data dependence edges. Then, it invokes the DDA (Dynamic Data-dependence Algorithm), which adds dynamic data dependencies into the partial MAODG.

Algorithm 9 Dynamic MAODG Construction Algorithm

```
INPUT: P- Input program, I- Input set for P
OUTPUT: The MAODG for P

1: Invoke Concurrent Control-dependence Algorithm (CCA) for constructing $MAODG_P$
   having nodes and control dependence edges.
2: Invoke Dynamic Data-dependence Algorithm (DDA) for dynamic construction of
   $MAODG$ having data dependence edges.
```

Concurrent Control-dependence Algorithm (CCA)

The pseudo code for CCA is given in Algorithm 10. CCA creates the partial MAODG of the given program. It first creates a node for each statement or predicate present in the program. Some additional nodes are generated to satisfy the syntax of SDG, like actual parameter-in/out, formal parameter-in/out nodes, etc. Next, control dependence edges are inserted between the nodes if one node is controlling the execution of another node. Then, call dependence, thread dependence and weaving edges are identified and added sequentially into the graph. The resultant partial MAODG of the example program given in Figure 5.4 is shown in Figure 5.5.

Dynamic Data-dependence Algorithm (DDA)

The pseudo code for *Dynamic Data-dependence Algorithm (DDA)* is given in Algorithm 11. After the construction of partial MAODG for the given program, we execute the given program by providing the input values to find the run-time data dependencies. The *input variables* are such variables in a program, whose values must be provided during run-time by the user, like command line arguments in Java. To capture these dynamic dependencies, we execute the example program given in Figure 5.4 and observe that the following statements are executed: [2, 4, a4, a6, a7, b3, b5, b6, 7, 9, 5, a10, a12, a13, b3, b5, b6, 11, 13]. The nodes that are involved in the execution are represented by shaded nodes in Figure 5.5. *Dynamic Data-dependence Algorithm* uses dynamic analysis to determine actual data dependencies during run-time. We add data dependence edges (if any) between the shaded nodes only. The extra non-shaded nodes which are present in the partial MAODG are removed, as these nodes will not affect the computation of dynamic slice with respect to the given input. *This reduces the space complexity and time complexity of the graph generation process*. After removing the extra nodes, we find the complete MAODG of the given program for a particular input. The complete MAODG for the example program given in Figure 5.4 is shown in Figure 5.7.

In the MAODG shown in Figure 5.7, it can be observed that the *labels* of some nodes such as node $b5$, are not matching with the program statements, because our tool works on

Algorithm 10 Concurrent Control-dependence Algorithm (CCA)

```
1: INPUT: P- Input program
2: OUTPUT: $MAODG_P$- Partially constucted MAODG
3: for each executable statement or predicate $\in$ P do
4:     Create a node in the MAODG
5:     Create separate call nodes and actual-in/out nodes for each call site
6:     Create method entry nodes and formal-in/out nodes for each method entry node
7: end for
8: \\ Add control dependence edges
9: if node $j$ controls the execution of another node $k$ then
10:     add control dependence edge $j \rightarrow k$
11: end if
12: \\ Add call dependence edges
13: if node $n$ is a call node and $m$ is the entry node for the method called at $n$ then
14:     add call dependence edge $n \rightarrow m$
15:     insert a label $n$ on the call edge, i.e. $n \xrightarrow{n} m$
16:     add parameter_in edges between actual-in and formal -in nodes
17:     add parameter_out edges between formal-out and actual-out nodes
18:     insert label $n$ on each parameter edge
19: end if
20: \\ Add thread dependence edges
21: if node $t$ is a thread start node and $r$ is the entry node for the run method called at $t$ then
22:     add thread dependence edge $t \rightarrow r$
23:     insert a label $t$ on the call edge, i.e. $t \xrightarrow{t} r$
24: end if
25: if node $y$ is the last node of the run method for the run method called at $t$ then
26:     add thread dependence edge $y \rightarrow t$
27:     insert a label $t$ on the thread edge, i.e. $y \xrightarrow{t} t$
28: end if
29: \\ Add interference dependence edges
30: if $n$ is a node in the thread $t1$ which defines a variable $var$ and $m$ is a node in another
    thread $t2$ which uses the variable $var$ then
31:     add interference dependence edge $n \rightarrow m$
32: end if
33: \\ Add weaving edges
34: if $v$ is the entry node of an advice target method and $w$ is the starting node of before
    advice then
35:     add weaving edge $v \rightarrow w$
36:     insert a label $v$ on the weaving edge, i.e. $v \xrightarrow{v} w$
37: end if
38: if node $x$ is the last node of before advice then
39:     add weaving edge $x \rightarrow v$
40:     insert a label $v$ on the weaving edge, i.e. $x \xrightarrow{v} v$
41: end if
42: if node $c$ is the last node of an advice target method and $d$ is the starting node of after
    advice then
43:     add weaving edge $c \rightarrow d$
```

```
44:         insert a label $c$ on the weaving edge, i.e. $c \xrightarrow{c} d$
45:     end if
46:     if node $g$ is the last node of after advice then
47:         add weaving edge $g \rightarrow c$
48:         insert a label $c$ on the weaving edge, i.e. $c \xrightarrow{c} c$
49:     end if
50:     if node $a$ is the call node of an advice target method and $b$ is the starting node of around
        advice then
51:         add weaving edge $a \rightarrow b$
52:         insert a label $a$ on the weaving edge, i.e. $a \xrightarrow{a} b$
53:     end if
54:     if node $c$ is the last node of around advice then
55:         add weaving edge $c \rightarrow a$
56:         insert a label $a$ on the weaving edge, i.e. $c \xrightarrow{a} a$
57:     end if
58:     if node $p$ is a proceed() node inside around advice then
59:         add weaving edge between $p$ to the target method entry node
60:         add another weaving edge between the last node of the target method and node $p$
61:     end if
```

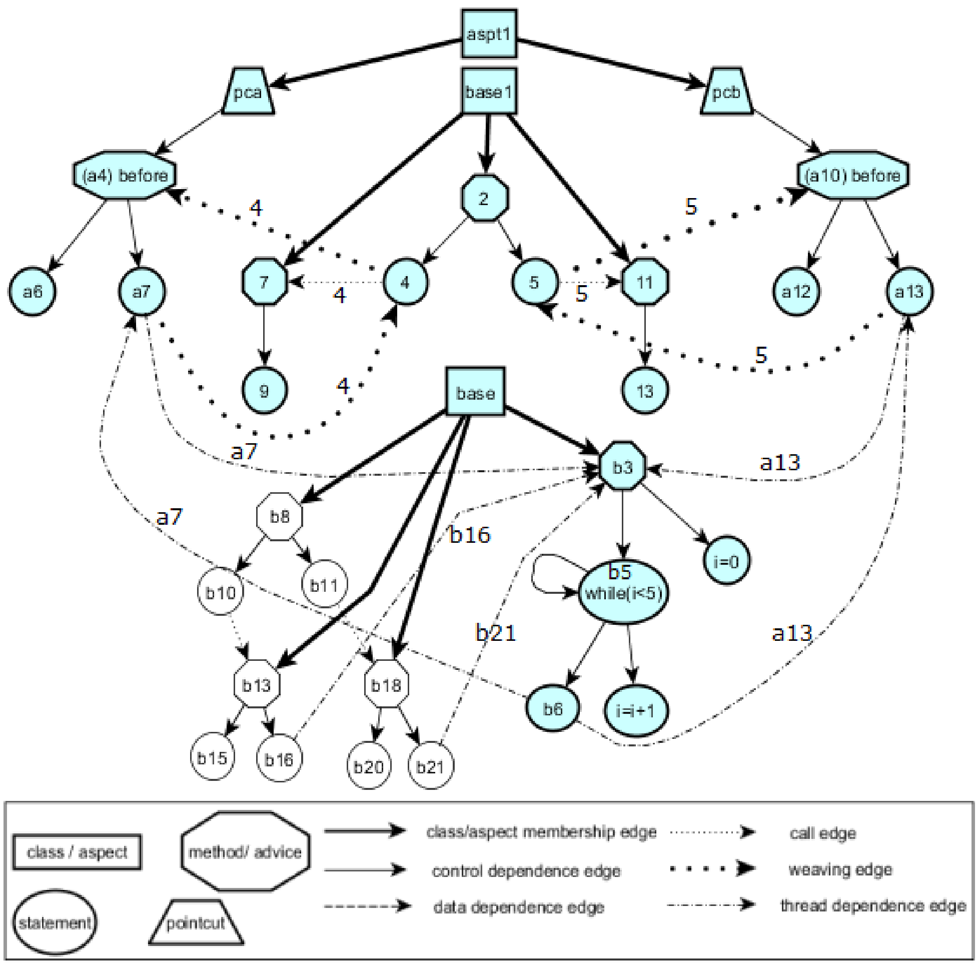


Figure 5.5: Partial MAODG for the example program given in Figure 5.4

Algorithm 11 Dynamic Data-dependence Algorithm (DDA)

```
1: INPUT:   $MAODG_P$- Partially constructed MAODG
2:          P- Input program
3:          I- Values for input program variables
4: OUTPUT: $MAODG$- Completely constructed MAODG
5: for each variable $v \in$ P do
6:     initialize $C(v) = \phi$ \\ C(v)= cache variable to store recent definition of $v$
7: end for
8: while (! P terminate) do
9:     for each statement $s \in$ P do
10:        execute statement $s$ of P associated with I
11:        for each variable $v \in s$ do
12:            if $v$ is defined at $s$ then
13:                $C(v) = s$
14:            end if
15:            if $v$ is used at $s$ AND $C(v) \neq \phi$ then
16:                add a data dependence edge $C(v) \rightarrow s$ to $MAODG_P$
17:            end if
18:        end for
19:    end for
20: end while
```

analysis of byte code. *In byte code, all for loops are converted into while loop*, hence our graph will contain only while loop notations.

Theorem 5.5. *Dynamic Data-dependence Algorithm (DDA) finds correct data dependence in a given program.*

Proof: A formal proof of the theorem and correctness of DDA can be constructed through mathematical induction by following an approach similar to that given in the proof of Theorem 4.5. □

5.2.3 Determining size of the MAODG

Slicing of CAOP is based on the MAODG. Hence, the size of the MAODG is very much important. The size of the SDG for OOPs is given by Larsen et al. [10]. We have extended their method [10] for calculating the size of our MAODG. In a concurrent program, when a thread starts execution, the system calls the *run()* method defined for the thread. While determining the size of MAODG, we consider the threads as methods. The aspect in an AOP is similar to the class in an OOP. Hence, we consider the class and aspects as the same, when determining the size of MAODG. Another feature of AOP is *introduction*, where we can declare new members into an existing class. We have considered the method introduction as simple methods in a class.

The MAODG is a dynamic intermediate graph that contains only the nodes created at the run-time. In Algorithm 3, we are creating the nodes for only executable statements in a given CAOP. But, it is very difficult to determine the exact size of the MAODG, because we do not have any idea regarding which statements of the given CAOP will execute for a particular execution. Hence, we want to give an approximation of the size of MAODG based on the number of nodes in the graph. We have listed the quantifiers that contribute to the size of MAODG, as shown in Table 5.1.

In CAOP, most of the statements belong to either a method, or a thread, or an advice. There are some statements which are not under any of the above, but belong to a class (i.e. class data members) or an aspect (i.e. through introduction). Those are not considered in the construction of MAODG. So, we can say that, if a statement is executing during the run-time of a program, it means that the statement must belong to a method, or a thread, or an advice. The upper bound on the number of parameter vertices in a module *mo*, where *mo* $\in \{method, advice\}$, is given as follows:

$$ParamVertices(mo) = Args + Globals + LocalVars \tag{5.1}$$

If *mo* is a thread then there will be no parameter passing, so $Args$=0. Then, the number of parameter vertices in a thread is given as follows:

$$ParamVertices(mo) = Globals + LocalVars \tag{5.2}$$

Using Equation 5.1, Equation 5.2 and the attributes given in Table 5.1, we compute the upper bound on the size of any module *mo* as follows:

$$\begin{aligned} Size(mo) = O(|Vertices| + |CallSites| + 2 * ParamVertices(mo) \\ *(1 + |DIP| * |CallSites|)) \end{aligned} \tag{5.3}$$

Another unpredictable entity in a program is the number of times a module (which includes method, thread and advice) executes in a particular run of the CAOP. Let us assume that a module is called k times during the execution of a program. For k times, we are not producing the nodes of a module. The nodes in a module are created only once in the MAODG. Whenever a module is called, we only create the call node and parameter nodes. Suppose, for each call to a module, we create 3 nodes (i.e. one call node, one parameter-in node and one parameter-out node). For k calls to a module, total additional nodes created will be $3k$. Now, the number of nodes in the MAODG can be given as O(number of nodes in all the modules + $3k$). The number of nodes in all modules is determined by multiplying the size of a module and the number of modules in a program.

Hence,

$$\begin{aligned} Size(MAODG) &= O(\text{number of nodes in modules + 3k}) \\ &= O(\text{size(mo) * number of modules + 3k}) \\ &= O(\text{size(mo) * module + 3k}) \end{aligned} \tag{5.4}$$

As, the number of nodes created for calling the modules is very little as compared to the size of module, we can ignore the $3k$ value from the size of the MAODG.

Now,

$$Size(MAODG) = O(\text{size(mo) * module}) \tag{5.5}$$

$Size(MAODG)$ is an approximate estimation based on the attributes that contribute to the size of MAODG. The actual size of MAODG may be less than the approximated value.

Table 5.1: Parameters which contribute to the size of MAODG

Sl. No.	Parameter	Description
1	Vertices	Maximum number of predicates or statements in a single method, thread or advice
2	Edges	Maximum number of edges in a single method, thread or advice
3	Args	Maximum number of formal parameters in a method or advice
4	Globals	Number of global variables in the program
5	LocalVars	Maximum number of local variables or objects in a class or aspect
6	CallSites	Maximum number of call sites in a method, thread or advice
7	DIP	Depth of inheritance
8	modules	Number of methods, threads and advices

5.3 Context Sensitive Dynamic Slicing of AOPs

There are several approaches available for slicing of AOPs [14, 29, 36, 70], but there is a scarcity of slicing techniques for CAOPs. In this chapter, we propose a context-sensitive dynamic slicing algorithm for CAOPs. The context-sensitivity increases the precision and correctness of the slice [16] and the dynamic slicing reduces the size of computed slice [8].

5.3.1 Context-sensitivity

During the literature survey, it was found that there are different types of techniques used for obtaining the slices by various researchers [5, 55]. All such slicing techniques can be broadly classified as either repeated backward data-flow analysis or traversal of a program dependence graph (PDG). We have used graph traversal technique to compute the slice. Let us consider the MAODG shown in Figure 5.6 for the example program given in Figure 5.4. Suppose we want to find the slice for node '9'. We apply a simple backward traversal technique and found that the following nodes are included in the slice: 9, 7, 4, 2, 1, a7, a6, a4, pca, aspt1, b6, while(i<5), i=0, i=i+1, b3, and also a13, a12, a10, pcb, 5. But it is clearly observed that a13, a12, a10, pcb, and 5 must not be present in the slice, as in no means these nodes affect node 9. This type of blind traversal is called *context-insensitive* slicing. *In rest of the paper, we will use context-insensitive backward slicing as CI slicing algorithm.*

In order to find more precise and accurate slices, we must have to include context-sensitivity. Context-sensitivity is a property of program slicing algorithm to preserve the call-site during entry and exit of a method. In our proposed slicing approach, in order

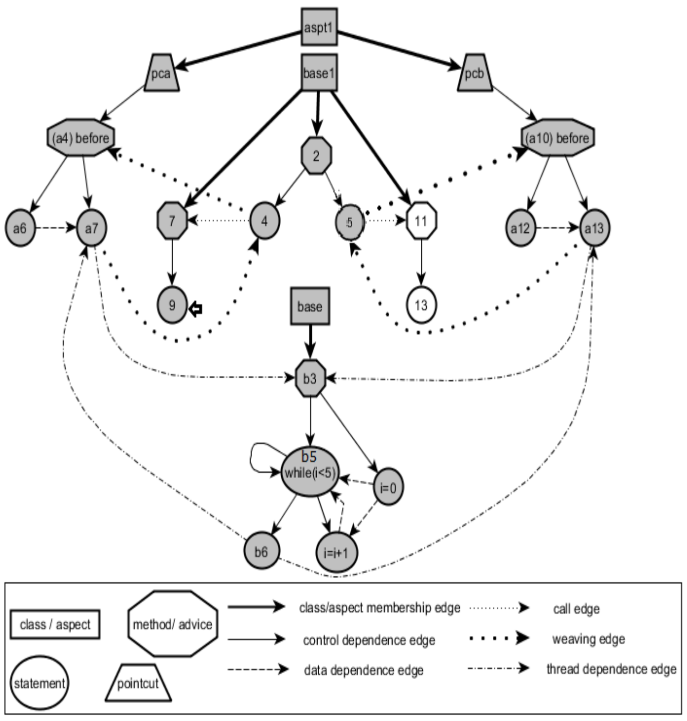


Figure 5.6: MAODG of the example program given in Figure 5.4, shaded nodes represent context-insensitive slice w.r.t. node 9

to incorporate the context-sensitivity, we have labelled call site of each call edge, thread edge, weaving edge and their corresponding return edges in the MAODG of a CAOP. The context-sensitivity in our approach is maintained by three stacks. These three stacks are: *Method stack*, *Advice stack* and *Thread stack*. Method stack keeps track of the call contexts of the methods, Advice stack is used for monitoring advice calls in an aspect and Thread stack keeps track of invocation of threads. Now for the same MAODG shown in Figure 5.6, we have modified the call, thread, weaving and their corresponding return edges by adding labels to them, and shown the modified MAODG in Figure 5.7.

Next, during the backward traversal, whenever we traverse any method return edge, we push the label of the edge into the Method stack and continue the traversal. When we reach the corresponding call edge, we pop from the Method stack and match the top of the stack with the current call edge label. If there is a match, we consider the current edge for further graph traversal; else we discard it. Similar process is followed for the advice and thread edges. The slice obtained using this approach is shown as shaded nodes in Figure 5.7. We can observe that now the nodes: a13, a12, a10, pcb, and 5, are absent in the slice.

5.3.2 Proposed Algorithm

The static slice is of large size than dynamic slice and also takes more computation time [16]. We propose a dynamic slicing algorithm that uses context sensitivity to compute the slices. We have named our algorithm Context-Sensitive Concurrent Aspect (CSCA) slicing

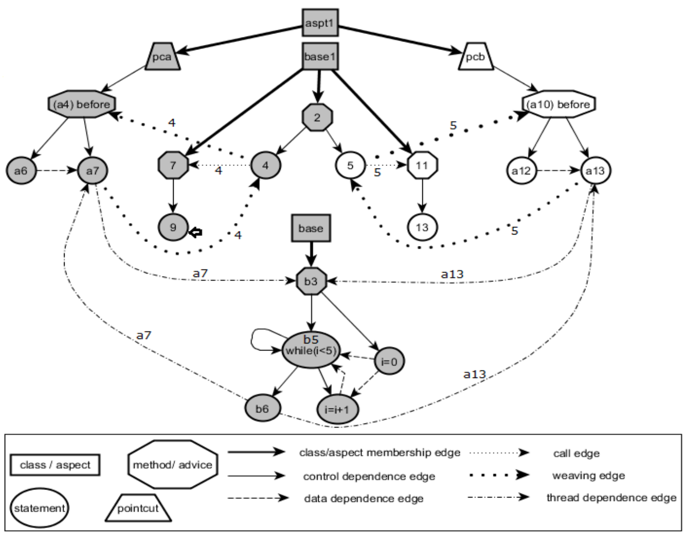


Figure 5.7: Modified MAODG of the example program given in Figure 5.4, shaded nodes represent context-sensitive slice w.r.t. node 9

algorithm. The pseudo code of the proposed CSCA slicing algorithm is given in Algorithm 12. Below, we briefly explain our proposed CSCA slicing technique for CAOPs.

Context-Sensitive Concurrent Aspect (CSCA) slicing Algorithm

The pseudo code for the proposed CSCA algorithm is given in Algorithm 12. The CSCA slicing algorithm is responsible for maintaining context-sensitivity during computation of slices. It takes the MAODG of a CAOP, as input and produces a list L which includes all the nodes included in the computed slice, as output. It maintains three stacks, i.e. SC, ST, and SA, for call-context, thread-context and advice-context, respectively. The algorithm is based on a worklist. First, the worklist W is initialized with the slicing criterion node ‘s’. The CSCA algorithm runs until the worklist W becomes empty. Repeatedly one node is removed from W and added into L and all the incoming edges to this node are examined.

Let CS be the label of one incoming edge into the slicing criterion node ‘s’. First the algorithm checks for method call related edges, such as call, parameter_in and parameter_out edges. If the current edge is a call or parameter_in edge, then the algorithm checks the status of the corresponding call-context stack SC. If SC is empty, then the source node of the current edge is inserted into worklist W. If SC is not empty, then the top element of SC is fetched and matched with the label of current edge. If both of them match, then only the source node of the current edge is inserted into W and the top element of SC is removed. If the current edge is found to be a parameter_out edge, then the source node of the edge is inserted into W and SC.

When any edge during the traversal of MAODG is not found to be a call related edge, then the algorithm checks it with thread related edges. To handle the thread related edges, we use thread stack ST. If the current edge is a *thread_start* edge, then the algorithm checks the status of the thread stack ST. If ST is empty, then the source node of the current edge is added into W. If ST is found to be non-empty, then the top element of ST is fetched and matched with the label of the $thread\_start$ edge. When both of them match, then only the source node of the $thread\_start$ edge is inserted into W and the top element is removed from ST. When any $thread\_return$ edge is found during traversal, then its source node is inserted into W and ST. If any inter-thread dependence is found, where a variable is defined in one thread and used by another thread, then an interference dependence edge is added from the variable definition node to the variable use node.

If an edge is still not processed by the algorithm, then it checks for the weaving edges. Weave stack SA is used to maintain the context-sensitivity in presence of weaving edges in MAODG. When a $weave\_start$ edge is encountered, the algorithm first checks the stack SA. If SA is empty, then the source node of $weave\_start$ edge is directly added into W. If SA is not empty, then the top element of SA is fetched and matched with the label of $weave\_start$ edge. The source node of the $weave\_start$ edge is inserted into W, when there is a match. When any $weave\_return$ edge is encountered, then its source node is added into W and also into SA.

If any edge does not match with any of these types of edges, then it must be a control or data dependence edge within one method. For these types of edges, the algorithm just adds the source node of these edges into W. This completes one iteration of the algorithm. Then, the same procedure is repeated by removing next element from W and adding it into L. Then all its incoming edges are examined. This process continues till the worklist W becomes empty. Finally, the nodes in slice are present in list L, which is the output of our algorithm.

Working of CSCA algorithm

Below we present the working of our CSCA slicing algorithm. Let us take node 9 as the slicing criterion, as shown in Figure 5.7. Initial values of $W = \{9\}, L = \{\}$ and $SC = ST = SA = \{\}$. Now, let us start traversing backward from node 9. The status of various data structures when processing different nodes during construction of the slice is given in Table 5.2. During traversal, first we reached at node 7. The connecting edge from node 9 to node 7 is a control dependence edge. As it is not a call, or thread or weaving edge, so no stack will be used, only we have to add node 7 into W. In the next step, node 7 is removed from W and added into L and its incoming edges are checked. One class membership edge (from node base1) and one call edge (from node 4) are found. Both the new nodes associated with these two edges must be added to W, as shown in the third row of Table 5.2. The label of the call edge (i.e. 4) is added into call stack SC.

In the next iteration, we remove node 4 from W and add it to L. The incoming edges to node 4 are traversed and the new nodes, i.e. node 2 and node ‘a7’, are added into W. During this traversal, we found presence of the weaving edge between node 4 and node ‘a7’. Hence, we push the label of the weaving edge (i.e. 4) into the Weave stack SA. Then, node 2 is

Algorithm 12 Context-Sensitive Concurrent Aspect (CSCA) Slicing Algorithm

```
1: INPUT:  $MAODG$ = (V,E)
2:        Slicing criterion s, s$\in$ V
3: OUTPUT:  L- the list of nodes in the computed slice for s
4: W = $\{s\}$ // initialize the worklist with s
5: L = $\{\}$  // the slice set
6: SC = $\{\}$ // the stack for maintaining method call-context
7: ST = $\{\}$ // the stack for maintaining thread start-context
8: SA = $\{\}$ // stack for maintaining advice call-context
9: Unmark all the edges in MAODG
10: repeat
11:    W= W $\setminus \{n\}$
12:    L= L $\bigcup \{n\}$
13:    for all $m \underset{e}{\rightarrow} n$ do // handle all incoming edges of $n$
14:       Let $CS_e$ is the call-site of edge $e$
      //for handling method call related edges
15:       if  $e \in \{parameter\_in, call\}$ && e has not been marked  then
16:          if $SC\,[TOP] == \phi$ then
17:             W= W $\bigcup \{m\}$
18:             mark $e$
19:          end if
20:          if $SC\,[TOP] == CS_e$ then
21:             W= W $\bigcup \{m\}$
22:             mark $e$
23:             POP(SC)
24:          end if
25:       else if  $e \in \{parameter\_out\}$ && e has not been marked  then
26:          W=W $\bigcup \{m\}$
27:          SC.PUSH(m)
28:          mark $e$
      //for handling thread related edges
29:       else if  $e \in \{thread\_start\}$ && e has not been marked  then
30:          if $ST\,[TOP] == \phi$ then
31:             W=W $\bigcup \{m\}$
32:             mark $e$
33:          end if
34:          if $ST\,[TOP] == CS_e$ then
35:             W=W $\bigcup \{m\}$
36:             mark $e$
37:             POP(ST)
38:          end if
39:       else if  $e \in \{thread\_return\}$ && e has not been marked  then
40:          W=W $\bigcup \{m\}$
41:          ST.PUSH(m)
42:          mark $e$
      //for handling aspect related edges
43:       else if  $e \in \{weave\_start\}$ && e has not been marked  then
44:          if $SA\,[TOP] == \phi$ then
```

```
45:                 W=W $\bigcup\{m\}$
46:                 mark $e$
47:             end if
48:             if $SA[TOP] == CS_e$ then
49:                 W=W $\bigcup\{m\}$
50:                 mark $e$
51:                 POP(SA)
52:             end if
53:         else if  $e \in \{weave\_return\}$ && e has not been marked  then
54:             W=W $\bigcup\{m\}$
55:             SA.PUSH(m)
56:             mark $e$
57:         else//intra-procedural edges
58:             if  e has not been marked  then
59:                 W=W $\bigcup\{m\}$
60:                 mark $e$
61:             end if
62:         end if
63:     end for
64: until W $== \phi$
```

processed, and it is added to L. The incoming edge into node 2 is from node ‘base1’, which is already inside W. So, it is not inserted into W. Next, we process node ‘a7’. First it is added to L and its incoming edges are examined. We found one data dependence edge from node ‘a6’ and another ‘thread_return’ edge from ‘b6’. Both the new nodes are added into W. The label of thread_return edge, which is ‘a7’, is pushed into thread stack ST. We repeat this process till worklist W becomes empty. At the end, the nodes included in the slice are listed into L, which is the output of our algorithm. The nodes contained in the list L in the last row and last column of Table 5.2, represent the resultant slice w.r.t. the slicing criterion 9.

5.3.3 Correctness of Context-Sensitive Concurrent Aspect (CSCA) slicing algorithm

In this section, we sketch the proof of correctness of our CSCA slicing algorithm.

Theorem 5.6. *Context-Sensitive Concurrent Aspect (CSCA) slicing algorithm always computes accurate and precise slices for concurrent AOPs.*

Proof: In this section, we sketch the correctness proof of CSCA slicing algorithm. The main properties of an algorithm are completeness, correctness and finiteness. Hence, the proof of our algorithm consists of three parts. First we prove that our algorithm is complete, i.e. it covers all the possible cases. Secondly, we prove that the algorithm is correct. Finally, we show that our algorithm terminates after finite number of iterations.

Table 5.2: Status of various data structures during computation of the slice w.r.t. slicing criterion 9

Processed Node	**W**	**SC**	**ST**	**SA**	**L**
9	{7}	{}	{}	{}	{9}
7	{4, base1}	{4}	{}	{}	{9,7}
4	{2,a7,base1}	{}	{}	{4}	{9,7,4}
2	{a7,base1}	{}	{}	{4}	{9,7,4,2}
a7	{a6,a4,b6,base1}	{}	{a7}	{4}	{9,7,4,2,a7}
a6	{a4,b6,base1}	{}	{a7}	{4}	{9,7,4,2,a7,a6}
a4	{pca,b6,base1}	{}	{a7}	{}	{9,7,4,2,a7,a6,a4}
pca	{aspt1,b6,base1}	{}	{a7}	{}	{9,7,4,2,a7,a6,a4,pca}
aspt1	{b6,base1}	{}	{a7}	{}	{9,7,4,2,a7,a6,a4,pca,aspt1}
b6	{b5,base1}	{}	{a7}	{}	{9,7,4,2,a7,a6,a4,pca,aspt1,b6}
b5	{b3,base1}	{}	{a7}	{}	{9,7,4,2,a7,a6,a4,pca,aspt1,b6,b5}
b3	{base,base1}	{}	{}	{}	{9,7,4,2,a7,a6,a4,pca,aspt1,b6,b5,b3}
base	{base1}	{}	{}	{}	{9,7,4,2,a7,a6,a4,pca,aspt1,b6,b5,b3,base}
base1	{}	{}	{}	{}	**{9,7,4,2,a7,a6,a4,pca,aspt1,b6,b5,b3,base,base1}**

Suppose Γ is the set of types of edges present in MAODG. In our algorithm, $\Gamma = \{control, data, call, thread, weaving\}$. Initially, the intended slice consists of only slicing criterion node s. There can be two possibilities, i.e. s may be a root node or may not be a root node. If s is a root node, then the slice will have only the slicing node. If s is not a root node, then it must be connected to some other node through an edge e. According to our algorithm $e \in \Gamma$, which is true, because all possible types of dependencies are covered in Γ and are handled by the algorithm. This shows that our algorithm is complete.

We prove the correctness of our algorithm using method of induction. We assume that the current computed partial slice $S_p = \{s_1, s_2, ..., s_{i-1}\}$ is correct. We have to show that after including the next node during the traversal, S_p retains its correctness. Suppose s_i is the next node in the traversal and e_i is the edge from $s_i \rightarrow s_{i-1}$. If

$$\begin{cases} \{s_{i-1}, s_i\} \in \text{same method} & \text{add } s_i \text{ to } S_p \\ s_{i-1} \in callee, s_i \in called & \text{pop label from SC and match} \\ s_{i-1} \in called, s_i \in callee & \text{add } s_i \text{ to } S_p\text{, push label into SC} \end{cases} \tag{5.6}$$

First, we concentrate on the method call section of our algorithm. There can be three possible locations of the two nodes, as shown in left hand side of Equation-5.6. According to our algorithm, and as shown in Equation-5.6, if both nodes s_i and s_{i-1} lie on the same method, then we directly insert node s_i into the slice. When node s_{i-1} belongs to the callee method and node s_i is in the called method, then we check the calling context by performing a pop operation on stack SC and matching the current edge label. If SC is empty, then also we have to add the current node s_i into the slice. In the other case, when node s_{i-1} belongs to the called method and node s_i is in the callee method, then we need to match the current edge label and the top of the stack SC, so that the calling context can be preserved.

When the edge e_i is not covered by the method call edge section, then the thread handling section of our algorithm will be invoked. In this section, all thread dependence edges are handled. Depending upon the position of nodes s_i and s_{i-1}, there can be three possible cases, as shown in left hand side of Equation-5.7. If both the nodes of an edge are present in the same thread, then it represents intra-thread dependence and the new node should be added into the slice. According to our algorithm, this type of case arises only for control or data dependence edges. In this case, the next node s_i is added into slice S_p. If node s_{i-1} is present in any method or advice part and new node s_i is present in a thread, then the edge e_i must be a $thread\_start$ edge. To preserve the context-sensitivity during MAODG traversal, we must match the label on the edge e_i and the top of thread stack ST. If both of them match, then node s_i will be added into the slice S_p and the top element of ST must be popped. In the reverse case, when the node s_{i-1} is present in a thread and new node s_i belongs to a method or advice, then the new context must be added into the thread stack ST. According to our algorithm, we add the new node to the slice and push the new node into ST. Hence, this situation is also handled correctly in the CSCA slicing algorithm.

$$\begin{cases} \{s_{i-1}, s_i\} \in thread & \text{add } s_i \text{ to } S_p \\ s_{i-1} \in method \text{ or } advice,\ s_i \in thread & \text{pop label from ST and match} \\ s_{i-1} \in thread, s_i \in method \text{ or } advice & \text{add } s_i \text{ to } S_p\text{, push label in ST} \end{cases} \tag{5.7}$$

If the current edge e_i is still unattended, then the last section of our algorithm, which deals with weaving edges, will handle it. In this case, only two situations are left, i.e. node s_{i-1} is in non-aspect part and the new node s_i is in aspect part, or vice versa. As shown in Equation-5.8, in the first case, the type of edge e_i must be $weave\_start$ edge. If during the traversal of MAODG any $weave\_start$ edge is found, then the top element of weave stack SA is checked. If the current edge label and the top of SA match, then the new node s_i will be added into the worklist W. When node s_{i-1} is present in any advice and the new node s_i is in non-aspect part, then the new node s_i will be added into W and also pushed into SA. This will preserve the context-sensitivity of weaving edges. In this way, our proposed algorithm generates correct slices for any given concurrent AOP whose intermediate representation is MAODG.

$$\begin{cases} s_{i-1} \in non-aspect\ part,\ s_i \in advice & \text{pop label from SA and match} \\ s_{i-1} \in advice, s_i \in non-aspect\ part & \text{add } s_i \text{ to } S_p\text{, push label in SA} \end{cases} \tag{5.8}$$

To prove the finiteness of our proposed algorithm, we assume that there is no cycle present in the MAODG of the program. The graph is having finite number of vertices $\{1, 2, 3, ..., n\}$ and finite number of edges $\{e_1, e_2, e_3, ..., e_k\}$, where n and k are positive integers. Initially the user enters the slicing criterion. Suppose the user has entered a slicing criterion node s and s $\in \{1, 2, 3, ..., n\}$. As there is no cycle in the graph, hence the traversal of the MAODG will have finite steps. From this, we can conclude that our algorithm terminates after executing finite number of steps. □

5.3.4 Complexity Analysis

In the following we discuss the space and time complexity of the CSCA slicing algorithm.
Space complexity: The CSCA slicing algorithm works on the input MAODG. In case of MAODG, apart from common dependence edges present in EAOSDG, we have added *thread dependence* edge. Suppose the number of statements in an input concurrent AOP is n, then there will be n nodes in MAODG. Let the number of edges in MAODG is e. Then to store MOADG, we need the space for storing information of n nodes and information of each edge $\in e$. Hence, the space complexity of storing the MAODG is of order $O(ne)$.
Time complexity: Context-Sensitive Concurrent Aspect (CSCA) slicing algorithm is based on insertion and deletion from a worklist W. For finding the worst case time complexity, suppose all the nodes of MAODG are connected, i.e. MAODG is a fully connected graph. Let the number of nodes in MAODG in n. While processing any node s, we have to traverse rest $(n - 1)$ nodes. As a result the outer loop of CSCA slicing algorithm runs $n * (n - 1)$ times. So, the worst case time complexity of the context-sensitive concurrent aspect slicing algorithm is $O(n^2)$.

5.4 Implementation and Results

In this section, first we discuss the implementation details of our slicing tool. We have named our slicing tool *Concurrent AspectJ slicer*. Then, we present the details of the case study projects that we have considered for our experiment. Finally, we compare our approach with another closely related work and present the outcomes of the experiment.

5.4.1 Experimental Setup

We have developed a partial tool called Concurrent AspectJ slicer. For developing our slicing tool and conducting the case studies, we have used, a personal computer having Intel Core *i5* processor, clock speed 2.40GHz, primary memory 4 GB and Windows 7 Home Basic (32 bit) operating system. The findings of our study may vary if some other system configuration is used to replicate the implementation.

5.4.2 Overview of Concurrent AspectJ slicer

The architecture of concurrent AspectJ slicer is given in Figure 5.8. It consists of four main parts: ASM framework, JSDG package, MAODG generator and slicer. The core of

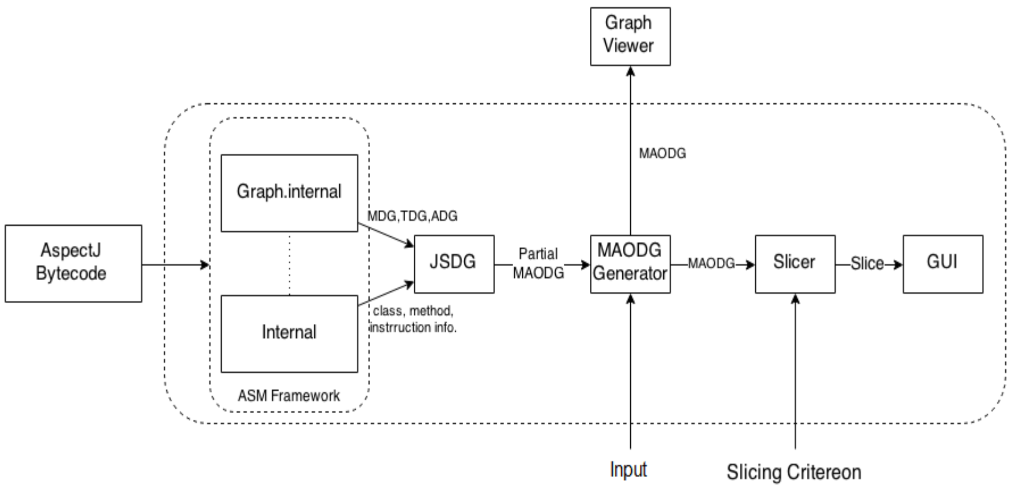


Figure 5.8: Architectural representation of Concurrent AspectJ slicer

the system is based on ASM [1] framework, which is a well known open source Java byte code manipulation and analysis framework. ASM is a collection of several packages for different analysis tasks. Two main packages of this collection are ‘‘internal” package and ‘‘graph.internal” package. The ‘‘internal” package analyses the Java bytecode and generates information about instructions, methods, classes, etc. The profiling is done by the *internal* package of the ASM framework and this package automatically produces the required dependence information. The ‘‘graph.internal” package uses the information fetched by ‘‘internal” package and generates the data-structure for method dependence graph (MDG) and thread dependence graph (TDG). We have developed a ‘‘JSDG” package which collects all the information from ASM framework and generates the partial MAODG for a given CAOP.

MAODG generator takes the input for the program’s input variables and uses our proposed DDA algorithm to find the dynamic dependences in the program. It then adds the dynamic dependences in the partial MAODG, which is provided by JSDG package, to create the complete MAODG. ‘‘Graph Viewer” is a GUI designed to display the created MAODG to the user. We have developed a package called ‘‘Slicer” that implements our context-sensitive dynamic slicing algorithm on the generated MAODG. The slicing criterion is also passed as input to the ‘‘slicer”. The ‘‘slicer” package computes the dynamic slices of the given CAOP and displays the resultant slice through a GUI.

5.4.3 Some related definitions

Before presenting the detailed case studies and discussions, we present below the definitions of some of the frequently used terms. In the case studies, we have used these terms to

[1]‘ASM Frameworks’, http://www4.comp.polyu.edu.hk/ cscllo/teaching/SDGAPI/.

compare our proposed slicing technique with some other existing slicing techniques.

Definition 5.7. Precise Dynamic Slice: A dynamic slice is said to be precise, if it is an executable slice and it contains *only* that statements that affect the value of the variable at a program point during execution.

Definition 5.8. Slice Size: Slice size is defined as the number of nodes of the dependency graph which are present in the resultant slice.

Definition 5.9. Slicing Time: Slicing time is defined as the time taken by the slicing algorithm to compute a slice. In our experiment we have considered the slice time in terms of milliseconds.

Definition 5.10. Slicing criterion: Slicing criterion is a point of the program, w.r.t. which we have to find the slice. For our experiment, we have taken the method return nodes of the dependency graph, as the slicing criteria. This point is justified as we are showcasing the context sensitivity while computing the slices and in a program the method return node is the most critical point where we can notice the effect of context sensitivity.

5.4.4 Case Studies

We have verified the efficiency and preciseness of our proposed slicing approach by comparing it with two existing approaches. For our first comparison, we have considered the context-insensitive (CI) slicing approach [42]. Then, we have compared our approach with a closely related approach proposed by Ray et al. [14]. We have implemented all these three approaches, i.e context-insensitive (CI) approach, the approach proposed by Ray et al., and our CSCA slicing approach, using our developed tool. All these three approaches are tested on five open source aspectJ projects, i.e. *Elevator-2 project*[2], *Red-Black Tree-1*[3], *online auction system* [4], *Tetris* game project [5] and *GoF Pattern-2* [6]. These projects are written in AspectJ and contain concurrency features such as *threads*. The details of the case study projects are given in Table 5.3. We have generated the MAODG for these projects using our tool and the values of the attributes of the MAODG are shown in Table 5.4.

First we have constructed the MAODG for each case study project. Next, all these three algorithms (CI, Ray et al. and CSCA) are applied on the above five projects considered for our experiment. The slicing process starts with determining suitable slicing criteria. It is known that there are five most popular slicing criteria used by several researchers, which are called *output-variables* [100]. Hence, we have chosen these output variables from the programs and computed the slices by taking these output variable nodes as slicing nodes. We have shown the concrete slices computed for the *Red-Black Tree* program, in Table 5.5. In this table, we have shown some selected slices only, as displaying all the thirty computed

[2]http://courses.cs.washington.edu/courses/cse142/03wi/projects/project2/StarterFiles/ElevatorController.java
[3]http://sir.unl.edu/php/showfiles.php
[4]http://lgl.epfl.ch/research/fondue/case-studies/auction/problem-description.html
[5]http://www.guzzzt.com/coding/aspecttetris.shtml
[6]https://www.cs.ubc.ca/labs/spl/projects/aodps.html

Table 5.3: Details of the attributes of the Case Study Projects

Sl. No.	Project	# Classes	# Aspects	LoC	Description
1	Elevator-2	5	3	352	Elevator control system. This version is a concurrent AspectJ program implementing Elevator system
2	Red-Black Tree-1	1	2	565	Implementation of Red-Black Tree data structure
3	Tetris	15	4	1027	Implementation of the popular game Tetris
4	OAS	21	9	1623	Online Auction System
5	GoF Patterns-2	30	14	3964	Implementation of GoF design patterns

Table 5.4: The values of the attributes of the MAODG of the Case Study Projects

Sl. No.	Project	No. of Nodes	No. of Edges	Time to generate MAODG
1	Elevator-2	540	997	302 ms
2	Red-Black Tree-1	785	1187	749 ms
3	Tetris	1667	2209	1432 ms
4	OAS	1783	2759	1140 ms
5	GoF Patterns-2	4205	3854	2568 ms

slices will take lots of space in the thesis. The second column of Table 5.5 shows, the node number of the MAODG for the given program, which is considered as slicing criterion node. The third column of Table 5.5 indicates the name of the slicing algorithm used, fourth column shows the list of nodes included in the slice, followed by the slice size and slice computation time in the last two columns. The comparative study of all the three slicing algorithms for each case study is done by taking the average slice size of all the computed slices for that case study. The average slicing time is also considered for comparing the effectiveness of the slicing algorithms. The average slice size and average slicing time obtained by using the three algorithms (CI, Ray et al., CSCA) on the above five case study projects are shown in Table 5.6 and Table 5.7 respectively.

Case Study-1: Elevator-2

This is an elevator simulator controller program. This controller generates a sequence of signals for the people arriving and changes the state of the system as directed by the ElevatorAlgorihm. The responsibilities of the controller include loading people on the elevator, unloading people, and moving the elevator to a specific floor. This is an open source Concurrent Java program and we have added three aspects into it to add some more functionalities. The detailed attributes of the Elevator controller program is given in Table

Table 5.5: Details of slices computed for Red-Black Tree-1 case study

Sl. No.	Slicing Criterion Node	Slicing Algorithm	Nodes in Generated Slice	Slice Size	Slicing Time (in ms)
1	37	CI	[1,32,33,35,36,37,86,87,89,90,110,111,112,113,114,115, 116,117,118,123,188,189,191,192]	24	5
		Ray et al.	[1,32,35,37,110,111,112,113,114,115,116,117,118,123]	14	5
		CSCA	[1,32,35,37,110,111,112,113,114,115,116,117,118,123]	14	5
2	91	CI	[1,32,33,35,36,86,89,90,91,110,111,112,113,114,115,116, 117,118,123,188,189,191,192]	24	8
		Ray et al.	[1,86,89,91,110,111,112,113,114,115,116,117,118,123]	14	5
		CSCA	[1,86,89,91,110,111,112,113,114,115,116,117,118,123]	14	4
3	108	CI	[1,94,98,102,106,108,124,126,129]	9	5
		Ray et al.	[1,102,106,108,124,126,129]	7	4
		CSCA	[1,102,106,108,124,126,129]	7	4
4	193	CI	[1,32,33,35,36,86,87,89,90,110,111,112,113,114,115,116, 117,118,123,188,189,191,192,193]	24	5
		Ray et al.	[1,110,111,112,113,114,115,116,117,118,123,188,191,193]	14	5
		CSCA	[1,110,111,112,113,114,115,116,117,118,123,188,191,193]	14	4
5	342	CI	[1,32,33,35,36,86,87,89, ... , 766,772,778,784]	228	15
		Ray et al.	[1,130,145,148,155,158, ... , 336,338,339,340,342]	39	7
		CSCA	[1,130,145,148,155,158, ... , 336,338,339,340,342]	39	5
.	.	.	.		.
.	.	.	.		.
.	.	.	.	.	.
.	.	.	.	.	.

5.3. The Elevator controller program was given as input to our tool and it generated the MAODG for the program. Table 5.4 contains the details about the different attributes of the MAODG for the program. We have applied all the three slicing algorithms, i.e. CI, Ray et al. [14] and CSCA, to compute the slices of the Elevator controller program. We have computed 15 slices for this Elevator controller program using each algorithm and found that the average slice sizes by using our CSCA and Ray et al.'s algorithm [14] are almost same. But, the average slice size obtained by using CI algorithm is 47% more than that of CSCA and the algorithm of Ray et al. [14]. While comparing the average slicing time, we found that, our CSCA algorithm runs 36.25% and 42.16% faster than that of Ray et al. and CI algorithm, respectively.

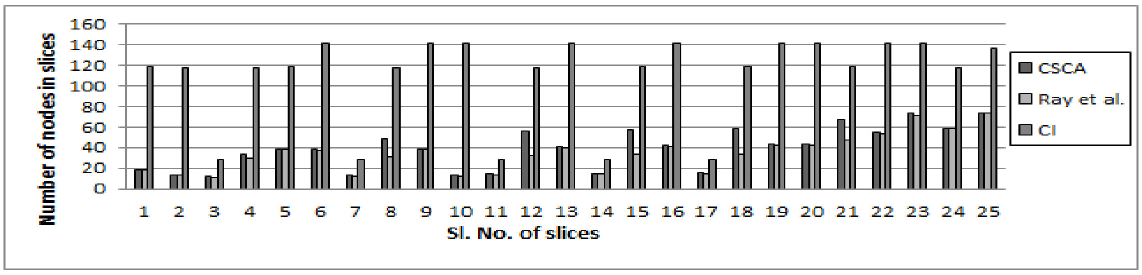
Case Study-2: Red-Black Tree-1

Red-Black tree is a balanced binary tree where each of the nodes are painted either red or black color. We have implemented the Red-Black tree using AspectJ. The detailed attributes of Red-Black tree program are given in Table 5.3. The MAODG generated for this project using our tool contains 785 nodes and 1187 edges. We have implemented three slicing algorithms, i.e. CI algorithm, Ray et al. and our CSCA algorithm on this case study. We have computed 30 slices by providing different slicing nodes to our tool. We found that the average slice sizes computed by our CSCA and Ray et al. algorithms are almost same. But the average slice size obtained by using CI algorithm, is 32.71.5% more than that of our CSCA and Ray et al.'s algorithm [14]. Also, we found that the average slicing time of our CSCA algorithm is 31.56% and 58.55% faster than that of the Ray et al.'s Algorithm [14] and CI algorithm, respectively.

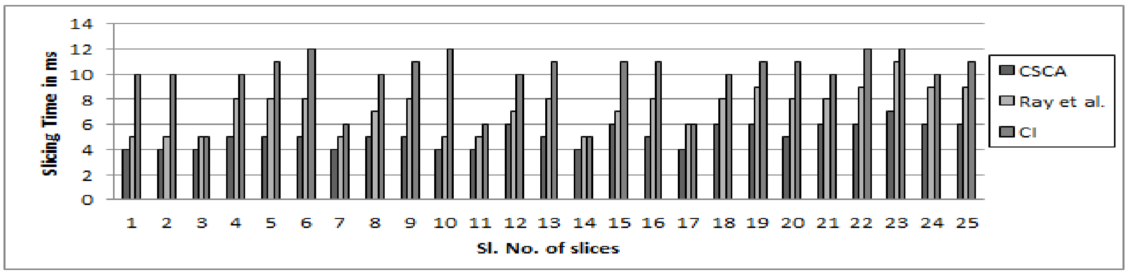
Case Study-3: Tertis

This is a very popular game. The goal is to pack the blocks so that they become lines. The lines are then deleted so that the user can add more blocks. The blocks are randomly generated at the top of the game board and are slowly dropped down until they reach the bottom. The game ends if the blocks reach to the top of the game board. This project is the re-implementation of existing concurrent Java project by introducing AOP concepts.

We have computed 46 slices by providing 46 different slicing criteria to our tool. We have implemented three algorithms, i.e. context-insensitive (CI), Ray et al.'s algorithm and our CSCA algorithm, on this case study. We have noted the slice size and slicing time for all the slices computed. It was observed that, out of 46 slices for each algorithm, 21 slices are found to be equivalent for each algorithm. Hence we are not considering these slices for our study. Rest 25 slices are compared w.r.t. their size and computation time for each algorithm. Figure 5.9a shows the comparison of slice size using the three algorithms. From Figure 5.9a, it can be observed that the average size of slices computed using CSCA is 13.32% less than the average slice size of Ray et al.'s algorithm and 68.88% less than the average slice size of CI algorithm. Figure 5.9b shows the comparison of slicing times of the three algorithms. From Figure 5.9b, it can be observed that the average slicing time of CSCA is 29.83% and 47.95% less than the Ray et al.'s algorithm [14] and CI algorithm, respectively.



(a) Comparison of slice size

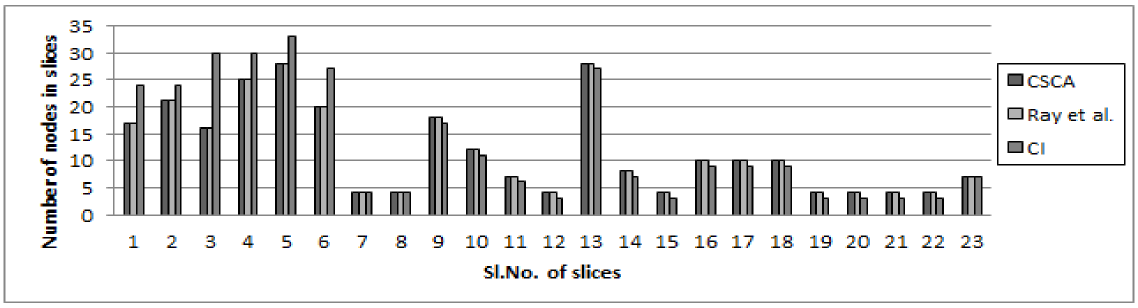


(b) Comparison of slicing time

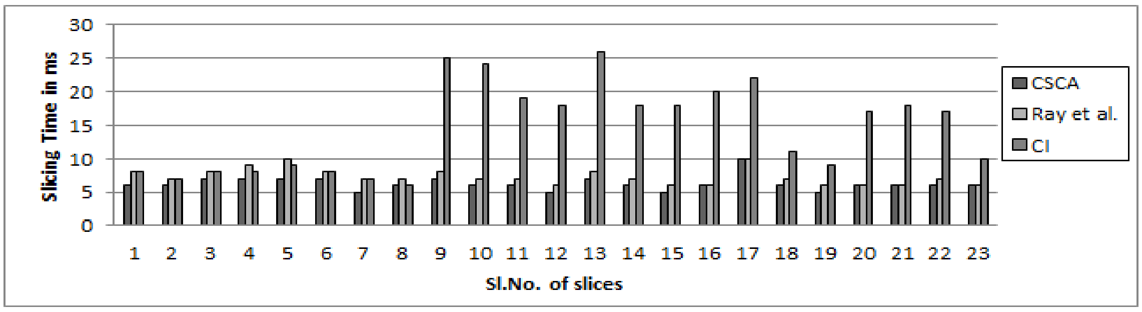
Figure 5.9: Findings of Case Study-3 (Tetris)

Case Study-4: Online Auction System (OAS)

OAS is an application software for buying and selling of goods through auction over internet. It allows the users of this system to negotiate over the buying and selling price of their goods. The OAS has 65 methods, and we have considered the *parameter_out* node of each method as our slicing criteria. Only 45 methods have *parameter_out* node in the MAODG and hence we have computed 45 slices by supplying different slicing criteria and input values to our tool. We have implemented all three algorithms as explained above and computed 45 slices by using each method. Carefully examining the outcome of the experiment, we observe that, in 22 out of 45 slices, we are getting the same slice size and slicing time for all the algorithms. Hence we have excluded them from our further analysis. The comparative study of rest 23 slices computed by each algorithm is given in Figure 5.10a, for slice size and in Figure 5.10b, for slicing time. From Figure 5.10a, it can be observed that the slices computed by using CSCA and Ray et al.'s algorithm are of same size. But, the average slice size of CSCA and Ray et al.'s algorithm is 9% less than the average slice size of CI algorithm. From Figure 5.10b, it can be observed that CSCA compute slices 13.24% faster than Ray et al.'s algorithm [14] and 55.57% faster than CI algorithm.



(a) Comparison of slice size



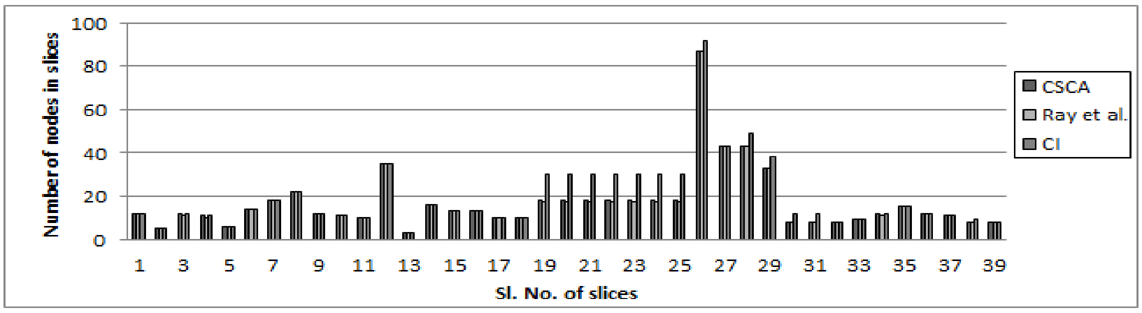
(b) Comparison of slicing time

Figure 5.10: Findings of Case Study-4 (OAS)

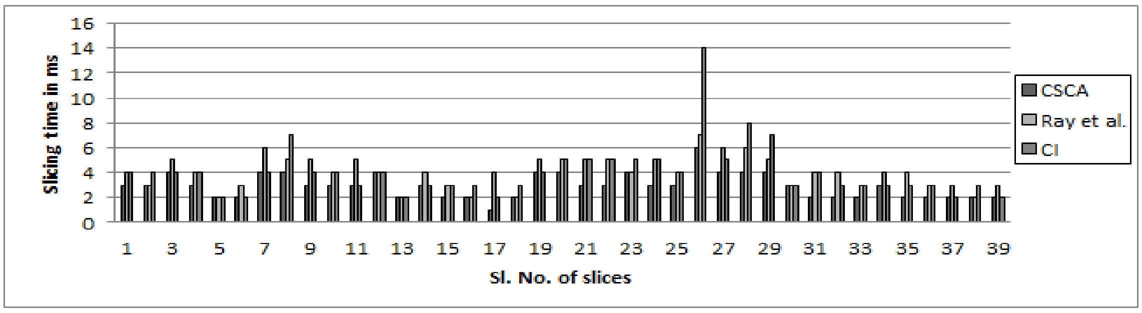
Case Study 5: GoF Patterns-2

This is the implementation of very famous GoF (Gang of Four) patterns by using AspectJ. 23 GoF design patterns are used in different programming languages, but a number of GoF patterns involve crosscutting structures in the relationship between roles in the pattern and classes in each instance of the pattern. These patterns are implemented in AspectJ to improve modularity. We have individually considered each pattern and performed slicing on them using CI, Ray et al. and our CSCA algorithms.

We have computed 39 slices by providing different slicing criteria to our tool. First we had applied CI algorithm, and computed 39 slices. After this we had used Ray et al. algorithm to compute 39 slices by providing the same slicing criteria. Lastly, our CSCA slicing algorithm was used to compute a set of 39 slices. Figure 5.11a shows the comparison of the average slice sizes using the three algorithms and it is visible that the slice sizes for most of the slices are remains same for all the three algorithms. It can be observed that, our CSCA slicing algorithm compute slices of same size as that of Ray et al.'s algorithm. But, the average slice size of CI is 15.28% more than CSCA and Ray et al.'s algorithms. Figure 5.11b displays the comparison of the slicing times taken by the three algorithms. The average slicing time of CSCA slicing algorithm is 27.9% and 27.36% less than the average slicing time of CI and Ray et al.'s algorithm, respectively.



(a) Comparison of slice size



(b) Comparison of slicing time

Figure 5.11: Findings of Case Study-5 (GoF Patterns-2)

5.4.5 Result Analysis

After completing the above experiments with five case studies, we have analysed the outcome to explore the hidden treasure. To produce a fair comparison of the performance of the three slicing algorithms, we have calculated the average slice size and the average slicing time for each case study.

Table 5.6: Comparison of Slice Size

Sl.No.	Project	Avg. Slice Size			Percentage Change	
		CI algo. [42]	Ray et al.'s algo. [14]	CSCA (proposed)	DCR$_1$	DCR$_2$
1	Elevator-2	102	54.73	54.06	47%	1.22%
2	Red-Black Tree-1	79.26	53.33	53.06	32.71%	0.5%
3	Tetris	109.52	39.32	34.08	68.88%	13.32%
4	OAS	13	11.83	11.83	9%	0%
5	GoF Patterns-2	19.82	17.02	16.79	15.28%	1.35%

In Table 5.6, we have used DCR$_1$ and DCR$_2$ whose definition is given below:

$$DCR_1 = (CI - CSCA)/CI * 100 \tag{5.9}$$

where, DCR$_1$ is *Percentage decrement of avg. slice size in CSCA algorithm as compared to CI algorithm.*

$$DCR_2 = (Rayetal. - CSCA)/Rayetal. * 100 \tag{5.10}$$

where, DCR$_2$ is *Percentage decrement of avg. slice size in CSCA algorithm as compared to Ray et al.'s algorithm.*

Table 5.6 shows the comparison of average slice size. In the first case study (i.e.Elevator-2 project), the CSCA algorithm and Ray et al.'s algorithm compute slices of same average size, but smaller than CI by 47%. In the second case study (i.e. Red-Black tree-1), we find the same result that our CSCA algorithm and Ray et al.'s algorithm computes similar results and average slice size is 33% less than CI slicing algorithm. For the third case study (i.e. Tetris project), the average slice size in our algorithm is 6% less than that of Ray et al.'s algorithm. The average slice size of CI is more than double than that of CSCA and Ray et al.'s approach. In the fourth project (i.e. OAS), our CSCA algorithm and the algorithm of Ray et al. give same average slice size and the average slice size obtained by CI algorithm is 9% more than these algorithms. Last case study is the GoF patterns, and the average slice size obtained by using of our approach is 1% less than that of Ray et al.'s approach. In this case study, the CI algorithm computed average slice size which is 18% more than that of our approach. From this, it implies that our CSCA algorithm and Ray et al.'s algorithm generate slices of almost same size and CI algorithm always gives very large size slices.

In Table 5.7, we have used DCR$_3$ and DCR$_4$ whose definition is given below:

$$DCR_3 = (CI - CSCA)/CI * 100 \tag{5.11}$$

where, DCR$_3$ is *Percentage decrement of avg. slicing time in CSCA algorithm as compared to CI algorithm.*

$$DCR_4 = (Rayetal. - CSCA)/Rayetal. * 100 \tag{5.12}$$

where, DCR_4 is *Percentage decrement of avg. slicing time in CSCA algorithm as compared to Ray et al.'s algorithm*.

Table 5.7: Comparison of Slicing Time

Sl.No.	Project	Avg. Slicing Time (in ms)			Percentage Change	
		CI algo. [42]	Ray et al.'s algo. [14]	CSCA (proposed)	DCR_3	DCR_4
1	Elevator-2	8.3	7.53	4.8	42.16%	36.25%
2	Red-Black Tree-1	14.96	9.06	6.2	58.55%	31.56%
3	Tetris	9.76	7.24	5.08	47.95%	29.83%
4	OAS	14.16	7.25	6.29	55.57%	13.24%
5	GoF Patterns-2	4.05	4.02	2.92	27.9%	27.36%

Now, when we compare the three algorithms considering the average slicing time, as shown in Table 5.7. We observe that, our CSCA algorithm computes slices much faster than Ray et al. algorithm and CI algorithm. In the Elevator-2 project, our CSCA algorithm computes slices faster than CI algorithm and Ray et al.'s algorithm by 42.16% and 36.25% respectively. For Red-Black Tree case study, the average slice computation time of our CSCA is 58.55% less than CI algorithm and 31.56% less than Ray et al.'s algorithm. In the Tetris case study, the average time taken to compute the slice using our CSCA is less than CI algorithm and Ray et al.'s algorithm by 47.95% and 29.83% respectively. Similarly in the case of OAS project, our slicing algorithm runs faster than that of CI algorithm and Ray et al.'s algorithm, by 55.57% and 13.24% respectively. Also, in the GoF pattern case study, the average slicing time of CSCA is 27.9% less than CI algorithm and 27.36% less than Ray et al.'s algorithm. Slice of smaller size makes the other side effects lesser. Most important is the slicing time, as the slices are used in other software development phases like debugging and testing. If the slicing time reduces, then the time and cost of debugging and testing will be ultimately reduced.

Form the above discussion, we can infer the followings:

- The proposed CSCA algorithm generates slices of almost same size or smaller size than that of Ray et al.'s algorithm. Hence it is more precise.
- CSCA algorithm produces sufficiently smaller slices in comparison to the CI algorithm.
- The time taken to generate the slices is always found to be minimum by using CSCA algorithm in comparison to Ray et al.'s algorithm and CI algorithm. Hence CSCA algorithm is faster than Ray et al. algorithm and CI algorithm.

5.4.6 Threats to validity

Below, we present some of the threats to the validity of our proposed approach.

1. As the slicing technique proposed in this chapter is only for AspectJ platform, it may not work satisfactorily for other aspect-oriented programming languages such as Aspect C++ and Aspect C#.

2. We have addressed some of the dynamic features of AspectJ like around and proceed. But, some other dynamic features like cflow, target/this, if etc., are not considered in this work.

3. Through the five case studies, we have tested our proposed slicing technique for computing precise and correct slices of projects upto 4000 LOCs. We believe, the other larger projects with similar structure in the same platform, may be handled through our slicing technique.

4. The proposed slicing technique is based on construction of intermediate representation (IR), and if the IR changes, then our slicing technique may not work properly.

5.5 Comparison with related work

There are many research work present in the field of concurrent object-oriented programs (OOPs), out of which the work of Zhao [69] is most efficient. Zhao proposed an intermediate representation called *System dependence Net* (SDN) to represent OOPs. But, SDN is no useful in representation of AOPs, because it cannot handle AOP features. So, in another work, Zhao [29] had extended SDN and proposed an intermediate representation for AOPs named *Aspect-oriented System Dependence Graph* (ASDG). It has special dependence edges for representing features of AOPs such as *aspect*, *advice* etc. But, Zhao had not developed any slicing algorithm for computing slices. Also, the proposed ASDG of Zhao is not capable of representing concurrent AOPs.

Ray et al. [14] presented a slicing approach for concurrent AOPs. Ray et al. [14] has developed intermediate representation *Concurrent Aspect-oriented System Dependence Graph (CASDG)* to represent concurrent AOPs. CASDG is constructed for a particular execution trace and next time if another execution trace is given then a new CASDG is constructed. This process is time consuming. In the work of Ray et al. [14], the concurrency model of AOP is not discussed. There are many concurrency models are present such a pointcut model, joinpoint model etc. In our proposed slicing approach, we have used the pointcut model given by Douence et al. [101]. Their slicing algorithm works on marking-unmarking of nodes during graph traversal. The marking-unmarking is very time consuming that makes the slice computation slow. In our proposed algorithm, we compute slice at the time of execution of input program, so not require to mark-unmark any node and it makes our algorithm faster then Ray et al. algorithm.

5.6 Summary

We proposed an intermediate graph called Multithreaded Aspect-Oriented Dependence Graph (MAODG) to represent CAOPs. MAODG represents the features of a CAOP. Based on this intermediate representation, we designed a context-sensitive dynamic slicing algorithm for CAOPs. To evaluate our proposed algorithm, we have designed a slicer. We presented five case studies of open source projects. We have implemented our approach, the context-insensitive (CI) slicing approach and the approach of Ray et al. [14]. Using the case studies, we have compared the efficiency of all the three approaches in terms of slice size and slicing time. We found that our CSCA slicing algorithm generated on an average 3% smaller slices than the approach of Ray et al. [14]. The CI approach is found to generate very large slices, i.e. more than double, in comparison to the other two techniques. We have also compared the slicing time of all the three algorithms and found that our CSCA slicing algorithm runs faster than the approach of Ray et al. and CI approach by 27% and 46% respectively. So, we conclude that our slicing technique computes more precise slices in little time. Hence, our proposed dynamic slicing technique for CAOPs is better than Ray et al.'s algorithm and CI algorithm.

The algorithm proposed in this chapter is not suitable to be applied to distributed AOPs running on several nodes connected through a network. In the next chapter, we are extending our framework to compute dynamic slices of distributed AOPs running on several nodes as is common in *Client-Server* applications.

Chapter 6

Dynamic Slicing of Distributed Aspect-Oriented Programs

The processing time taken for a single system to complete a task is larger than that of the processing time taken by more than one system performing the same task [102]. This is one of the advantages of distributed computing. Also, the reliability of the whole system increases by using distributed system, which makes the system more fault tolerant. In a distributed system, more than one computer is connected to a common network. The main task of an algorithm is divided into a finite number of small independent sub-tasks and assigned to different computers for execution of that small sub-tasks [103]. The most common example of a distributed system is the *client-server* computing. In this case, one computer acts as a server and rest of the computers act as clients. As the roles of the server and client are different, we have to write separate programs for the server and clients.

The programs written to perform some task in distributed manner are called distributed programs. In this chapter, we concentrate on Distributed Aspect-Oriented Programs (DAOPs). The basic program that performs all the intended computing for fulfilling the given task is written in a distributed manner. The basic distributed programs run on different computers connected to a network. The most expensive and tedious task in distributed programming is to handle the communication between the network nodes (i.e. computers). In the traditional object-oriented distributed programs, the distributed modules of the program communicate with each other by sending and receiving messages. But, the message can be passed to another computer at any point of time. Hence the codes for communication are scattered throughout the program. This type of code or section scattered throughout the program is called *crosscutting concern* [34]. AOP is a well-known programming paradigm, where all the crosscutting concerns are identified and bundled together in the form of an Aspect [34]. This Aspect module is responsible for the message passing between different modules running on different computers of the distributed system. This type of Aspect-Oriented Programs is called distributed AOPs. These distributed AOPs are very complex and difficult to understand. The rest of the activities in the Software Development Life Cycle (SDLC) apart from coding, rely on the complexity of the program. If a program is difficult to understand, then it will be very difficult to test it and debug the faults present in the program. Hence, there is a necessity of a program analysis tool that can reduce the complexity of a given distributed AOP. Program slicing is one of such techniques that work for the improvement of the understandability of a program by impact analysis. In our approach, we are interested in generating the executable slice, so that it can be used in program debugging and testing.

The rest of the chapter is structured as follows: Section 6.1 presents some basic concepts of distributed AOP. Section 6.2 contains the details of the proposed intermediate representation, called Distributed Aspect Dependence Graph (DADG) for distributed AOPs. Our proposed parallel context-sensitive dynamic slicing algorithm is also described in Section 6.3. We have implemented our slicing approach and developed a distributed AspectJ slicer. In Section 6.4, we present the implementation of our approach, the experimental study, the architecture of the slicer, and seven case studies. In Section 6.5, we present the comparison of our work with some related work. In Section 6.6, we summarize the chapter.

```
//MyClient.java
import java.net.*;
import java.io.*;
public class MyClient{}
public static void main(String [] args){
     BufferedReader input=new BufferedReader(new
 InputStreamReader(System.in));
     try {
        Socket client = new Socket("localhost", 9093);
        OutputStream outToServer = client.getOutputStream();
        DataOutputStream out = new DataOutputStream(outToServer);
        System.out.println("Enter a number");
        String N=input.readLine();
        out.writeUTF(N);
        InputStream inFromServer = client.getInputStream();
        DataInputStream in =
                new DataInputStream(inFromServer);
        System.out.println("Factorial= " + in.readUTF());
        client.close();
    }catch(IOException e)
     {
       e.printStackTrace();
     }}}
```

Figure 6.1: An example client program for calculation of factorial of a number

6.1 Basic Concepts

In this section, we introduce a few basic concepts and definitions that would be used in our proposed algorithm.

6.1.1 Distributed Aspect-Oriented Programs

Distributed computing is the concept of independent processors connected through communication links. After establishing the distributed architecture, the next step is to develop the programs that can run on a distributed environment [103]. These programs consist of several concurrently executable parts allocated to different processors for simultaneous execution. The distributed systems rely on message passing for communication between all the computers [78]. This message passing makes the distributed programs more complex. Here, the advantages of aspects can be availed to reduce the complexity of the distributed programs. The code that handles message passing between two computers can be identified and kept in a separate module as an aspect. These types of programs are called Distributed Aspect-Oriented Programs.

Among all the known architecture of distributed computing, the *client-server* architecture is the simplest and most commonly used architecture [102]. In Figure 6.1, we have considered an example distributed program for calculation of factorial of a given number. In Figure 6.1, $MyClient.java$ is a client program, which first establishes the connection with a server program by calling $Socket()$ method. Then it asks the user to enter a number through a keyboard and stores it in a variable. Then, using $getOutputStream()$ method the number is then sent to the server. At the end, the client program receives the result from the server program and prints the result.

In Figure 6.2, $MyServer.java$ is a server program, which is responsible for calculating the factorial of a given number. When it starts execution, it waits for a client program to send some data. Once the server program receives a number from the client program through $getInputStream()$ method, it then calculates the factorial of the input number. Then, it sends back the computed factorial of the number to the client that had sent the input number.

Now, we can observe that the server program is always running, once it starts execution. But, while observing the output, it is not understood when the server becomes ready for use. Similarly, when the work of the server is over, before closing the server program, all clients must be notified that the server is going to be closed so that no client should send new data. To incorporate these modifications, again we have to change the existing code. Another alternative way is to design a separate program that can handle all these additional requirements without changing the existing programs. We have created an aspect program using AspectJ [33] programming language to achieve this.

```
//MyServer.java
S1   import java.net.*;
S2   import java.io.*;
S3   public class MyServer extends Thread
S4   {  private ServerSocket serverSocket;
S5      public void run()
S6      { try
S7            { serverSocket = new ServerSocket(9093);
S8          }catch(IOException e)   {
S9             e.printStackTrace();}
S10           while(true)
S11          {try
S12             {System.out.println("waiting for the client");
S13              Socket server = serverSocket.accept();
S14              DataInputStream in =new
DataInputStream(server.getInputStream());
S15              String N=in.readUTF();
S16              System.out.println("data received from client "+N);
S17              int n=Integer.parseInt(N);
S18              int f = 1, i;
S19               if (n == 0)
S20               { f=1; }
else
S21                 { for (i = 1; i <= n; i++)
S22                   { f = f * i; }
}
S23                N=Integer.toString(f);
S24                System.out.println("Process Completed");
S25                DataOutputStream out =new
DataOutputStream(server.getOutputStream());
S26                out.writeUTF(N);
S27                server.close();
S28             }catch(IOException e)     {
S29                e.printStackTrace();
S30                break;
} } }
S31        public static void main(String [] args)
S32        {    Thread t = new MyServer();
S33             t.start();}}
```

Figure 6.2: An example server program

The AspectJ program is shown in Figure 6.3. We have developed an aspect called $MyServer\_Aspect.aj$ as shown in Figure 6.3, which has one *poincut* and two *advices*. The pointcut $PC()$ captures the execution of $run()$ method in $MyServer.java$ program shown in Figure 6.2. The $before()$ advice executes before the $run()$ method is executed and the $after()$ advice executes after the $run()$ method is executed. As a result, we can print the status of the server program, i.e. when it starts and when it completes.

```
//MyServer_Aspect.aj
A1   public aspect MyServer_Aspect {
A2   pointcut PC():execution(void MyServer.run());
A3   before():PC(){
A4      System.out.println("Server is starting...");}
A5   after():PC(){
A6      System.out.println("Server is closed!");} }
```

Figure 6.3: An example AOP for the server program

6.2 Intermediate Program Representation

Before computing the slice of a given DAOP, we have to represent it in the form of some dependence graph. The existing dependence graphs such as PDG, SDG, EAOSDG or MAODG, are not suitable to represent the features of DAOPs like message passing, synchronization, etc., because they do not have edges to represent these additional features of DAOPs. Hence, we have developed an intermediate representation named Distributed Aspect Dependence Graph (DADG) for DAOPs. In this section, we describe in brief about DADG.

6.2.1 Distributed Aspect Dependence Graph (DADG)

For developing an efficient program slicing technique, a suitable intermediate representation of the given program is required. Most of the researchers have used the dependence graph as intermediate representation and then proposed their slicing algorithms based on the dependence graph [43, 53]. We have proposed a dependence graph for representing the distributed AOPs accurately and precisely. We have considered various features of distributed AOPs, such as pointcuts, advices, inter-thread communications, etc. while constructing the intermediate graph representation. We have named this intermediate graph *Distributed Aspect Dependence Graph (DADG)*. DADG is a dynamic dependence graph whose different dependencies are added to the graph depending upon the present execution of the given program. DADG is a collection of Distributed Dependence Graphs (DDGs) for the non-aspect part and Aspect Dependence Graphs (ADGs) for the aspect parts. For each component of the distributed programs, we construct a DDG separately. We combine all the DDGs and ADGs to build the final DADG by using several special dependence edges. In DADG, we have developed one special node as defined below:

Definition 6.1. R-Node: Let P = $(P_1, P_2, ..., P_n)$ be a distributed program, where P_n is the n^{th} component program. We construct the DADG for the distributed program P by constructing a DADG for each component program represented by DADG=$(DADG_1, DADG_2, ..., DADG_n)$. If a statement 's' in a component program P_i is a message send statement and statement 'r' in P_j is the corresponding message receive statement, then we denote the node representing statement 'r' in the $DADG_j$ as an R-node.

The R-node is different from rest of the nodes of DADG because it stores the corresponding message send node (process-id and node number) in it.

For example, suppose that while generating the DADG for a component program P, we found that its process-id is $P1$, and it has a node named $node10$ which receives some message from another program Q which is running on another computer. Now, let the process-id of program Q is $P2$ and it has a node named $node12$ which is the corresponding message send node for $node10$ in P. Now, we have to add an R-node in the $DADG_P$ in-place-of $node10$ and store the pair $(P2, 12)$ in the R-node.

Algorithm 13 Dynamic DADG Construction Algorithm

```
INPUT:
P- Distributed A-O program
P = $(P_1, P_2, ..., P_n)$, where $P_i$ is a component program
I = $(I_1, I_2, ..., I_n)$, where I is the input set for P
OUTPUT: The DADG for P

1: Invoke Distributed Control-dependence Algorithm (DCA) for construction of the partial
   DADG ($DADG_P$) having all the nodes and control dependence edges.
2: Invoke Distributed Data-dependence Algorithm (DDA) for adding data dependence
   edges into $DADG_P$.
```

6.2.2 DADG Construction Algorithm

In our proposed DADG, we have categorized the different types of edges into two categories. Broadly, the control, call, thread and weaving edges are categorized as control dependence edges. Data dependence edges are kept under separate category from control dependence edges. In our approach, we construct all control dependence edges statically, and data dependence edges dynamically. Hence, our proposed DADG construction algorithm has two parts. For the construction of the DADG of a given distributed AOP, our algorithm first calls the Distributed Control-dependence Algorithm (DCA) for the construction of the partial DADG. Then, it invokes the Distributed Data-dependence Algorithm (DDA), which adds dynamic data dependencies into the partial DADG.

Distributed Control-dependence Algorithm (DCA)

The pseudo code for Distributed Control-dependence Algorithm (DCA) is presented in Algorithm 14. The DCA algorithm creates a partial DADG ($DADG_P$) of the given program. It first creates a node for each statement or predicate present in the program. Some additional nodes are generated to satisfy the syntax of SDG, like actual parameter-in/out, formal parameter-in/out nodes, etc. Next, control dependence edges are inserted between the nodes if one node is controlling the execution of another node. Then, step-wise call dependence, thread dependence, and weaving edges are added to the graph.

Algorithm 14 Distributed Control-dependence Algorithm (DCA)

```
1: INPUT: P- Distributed A-O program
           P = $(P_1, P_2, ..., P_n)$, where $P_i$ is a component program of P
2: OUTPUT: $DADG_P$- Partially constructed DADG
3: Let $P_e = \{P_i \mid P_i$ is a component program that executes in present scenario$\}$
4: for each $P_i \in P_e$ do
5:     for each executable statement or predicate $\in P_i$ do
6:         Create a node n in the DADG
7:         if n is a call node then
8:             Create separate actual-in and actual-out nodes for each call site
9:         else if n is a method entry node then
10:            Create formal-in and formal-out nodes for each method entry node
11:        else if n is a receive node then
12:            Rename the node as R-node
13:            Create a dashed circle node to represent R-node
14:        end if
15:    end for
       /* Add control dependence edges */
16:    if node $j$ controls the execution of other node $k$ then
17:        add control dependence edge $j \rightarrow k$
18:    end if
       /* Add call dependence edges */
19:    if node $n$ is a call node and $m$ is the entry node for the method called at $n$ then
20:        add call dependence edge $n \rightarrow m$
21:        insert a label $n$ on the call edge, i.e. $n \xrightarrow{n} m$
22:        add parameter-in edges between actual-in and formal -in nodes
23:        add parameter-out edge between formal-out and actual-out nodes
24:        insert label $n$ on each parameter edge
25:    end if
       /* Add thread dependence edges */
26:    if node $t$ is a thread start node and $r$ is the entry node for the run method called at $t$
   then
27:        add thread dependence edge $t \rightarrow r$
28:        insert a label $t$ on the call edge, i.e. $t \xrightarrow{t} r$
29:    end if
30:    if node $y$ is the last node of the run method for the run method called at $t$ then
31:        add thread dependence edge $y \rightarrow t$
32:        insert a label $t$ on the thread edge, i.e. $y \xrightarrow{t} t$
33:    end if
       /* Add weaving edges */
34:    if $v$ is the entry node of an advice target method and $w$ is the starting node of before
   advice then
35:        add weaving edge $v \rightarrow w$
36:        insert a label $v$ on the weaving edge, i.e. $v \xrightarrow{v} w$
37:    end if
```

```
38:     **if** node $x$ is the last node of before advice **then**
39:         add weaving edge $x \rightarrow v$
40:         insert a label $v$ on the weaving edge, i.e. $x \xrightarrow{v} v$
41:     **end if**
42:     **if** node $c$ is the last node of an advice target method and $d$ is the starting node of
    after advice **then**
43:         add weaving edge $c \rightarrow d$
44:         insert a label $c$ on the weaving edge, i.e. $c \xrightarrow{c} d$
45:     **end if**
46:     **if** node $g$ is the last node of after advice **then**
47:         add weaving edge $g \rightarrow c$
48:         insert a label $c$ on the weaving edge, i.e. $c \xrightarrow{c} c$
49:     **end if**
50: **end for**
```

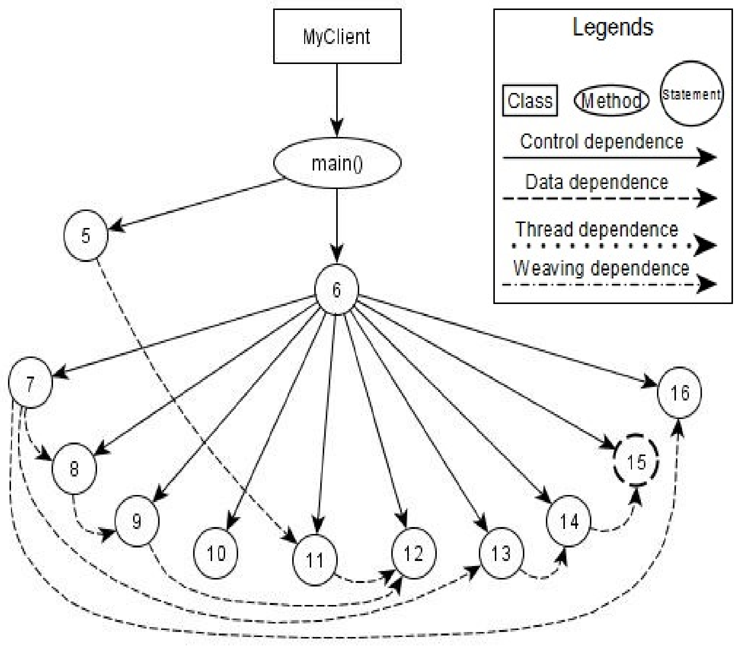


Figure 6.4: DADG for the example program given in Figure 6.1

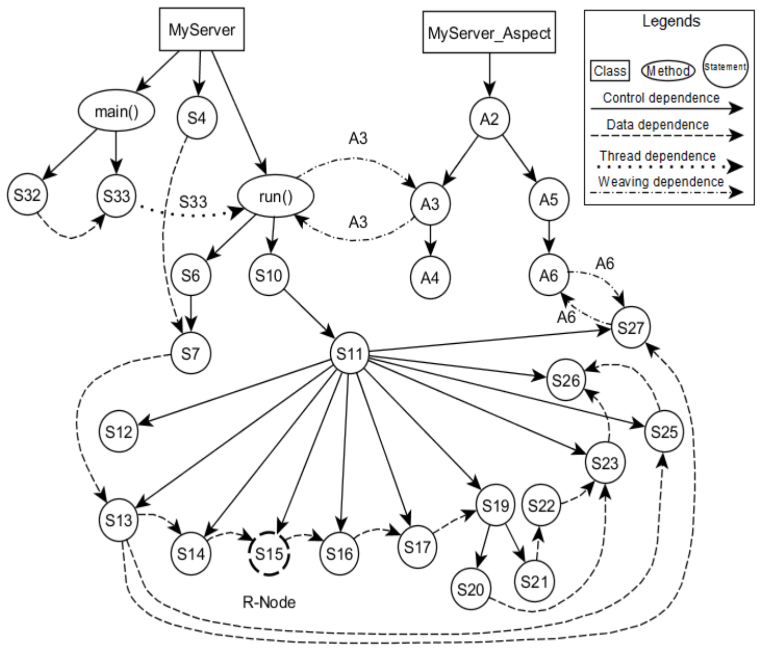


Figure 6.5: DADG of the server example program given in Figure 6.2 and Figure 6.3

Distributed Data-dependence Algorithm (DDA)

The pseudo code for Distributed Data-dependence Algorithm (DDA) is given in Algorithm 15. After the construction of partial DADG ($DADG_P$) for the given program, we execute the input program to find its execution trace. For example, the execution trace for the example client program given in Figure 6.1 is found as: [3, 4,5,6,7,8,9,10,11,12,13,14,15,16]. Similarly, we found the execution trace of the server program shown in Figure 6.2 as [S31,S32,S33,S5,S6,S7,S10,S11,S12,S13,S14,S15,S16,S17,S18,S19,S21,S22,S23,S24,S25, S26,S27]. The execution trace of the aspect code given in Figure 6.3 is [A2,A3,A4,A2,A5,A6]. The DCA has added all control dependence edges into the DADG. *Distributed Data-dependence Algorithm (DDA)* uses dynamic analysis to determine the actual communication and data dependencies during runtime. We add data dependency edges (if any) between the nodes of DADG by using DDA. The actual communication dependence between all the component programs of a distributed AOP is determined at run-time. In each of the DADG, we search for the R-nodes and store the pair (P_s, s), where P_s is the process-id of sender program and 's' is the sender node in the R-nodes. This helps in handling the communication dependence. *We do not add any communication edge explicitly between the receiver node and sender node, which reduces the space complexity and time complexity of the graph generation process*. Also, it helps us to find the dynamic slice of individual DADG parallely by applying our proposed slicing algorithm. The complete DADG for the example program of Figure 6.1 is shown in Figure 6.4. The DADG for the server program and aspect program of Figure 6.2 and 6.3 is shown in Figure 6.5.

Algorithm 15 Distributed Data-dependence Algorithm (DDA)

```
1: INPUT:   $DADG_P$- Partially constructed DADG
2:          P- Distributed A-O program
3:          I = $(I_1, I_2, ..., I_n)$, where I is the input set for P
4: OUTPUT: $DADG$ for the given P
5: for each $P_i \in P_e$ do
6:     for each variable $v \in P_i$ do
         /∗ Let C(v) present the statement of the current definition of the variable ∗/
7:         initialize $C(v) = null$
8:     end for
9:     while (! $P_i$ terminate ) do
10:        for each statement $s \in P_i$ do
11:            execute statement $s$ of $P_i$ associated with $I_i$
12:            if $s$ represents a $R - node$ in the $DADG_i$ then
13:                store $(P_x, x)$
         /∗ where $P_x$ is the process-id of sender and ‘x’ is the sender node ∗/
14:            end if
15:            for each variable $v \in s$ do
16:                if $v$ is defined at $s$ then
17:                    $C(v) = s$
18:                end if
19:                if $v$ is used at $s$ and $C(v) \neq null$ then
20:                    add a data dependence edge $C(v) \rightarrow s$ to $DADG_P$
21:                end if
22:            end for
23:        end for
24:    end while
25: end for
```

Theorem 6.2. *Dynamic Data-dependence Algorithm (DDA) finds correct data dependence in a given distributed program.*

Proof: A formal proof of the theorem and correctness of DDA can be constructed through mathematical induction by following an approach similar to that given in the proof of Theorem 4.5. □

6.2.3 Determining size of the DADG

The size of DADG can be calculated in similar manner as we have calculated the size of MAODG in Section 5.2.3 of Chapter 5.

6.3 Proposed Algorithm: Parallel Context Sensitive Dynamic Slicing

There are several approaches available for slicing of AOPs [11, 36, 104], but there is a scarcity of slicing techniques for distributed AOPs. Here, we propose a parallel context-sensitive dynamic slicing algorithm for distributed AOPs. The context-sensitivity increases the preciseness and correctness of the slice [16] and the dynamic slicing reduces the size of computed slice [105]. Finally, the parallelism increases the slice computation speed.

6.3.1 Context-sensitivity

There are different types of techniques used to obtain the slices by various researchers [16, 29, 82] All such slicing techniques can be broadly classified as either repeated backward data-flow analysis or traversal of a program dependence graph (PDG). We have used graph traversal technique to compute the slices. To find precise and accurate slices, we have considered context-sensitivity during slicing [16]. Context-sensitivity imparts that the call-site must be preserved during entry and exit of a method. To incorporate the context-sensitivity in our proposed slicing algorithm, we have labelled each call , thread, weaving edges and their corresponding return edges in the DADG of a distributed AOP. Then, we use three stacks to preserve the context-sensitivity in our approach. These three stacks are: *Method stack*, *Advice stack* and *Thread stack*. Method stack keeps track of the call contexts of the methods, Advice stack is used for monitoring advice calls in an aspect and Thread stack keeps track of invocation of threads.

6.3.2 Parallelism

An algorithm can be divided into some independent tasks (component tasks) that can be concurrently executed, and then these individual tasks can be assigned to different computers connected to a network, for achieving faster processing time. Now, the component tasks can be carried out on separate computers; as a result the execution time of the overall algorithm reduces. In our proposed slicing algorithm, we ensure accuracy and preciseness of the resultant slices by introducing context-sensitivity. Further, the slice computation time can be reduced by introducing parallelism into the slicing algorithm. The slicing algorithm is dependent on the intermediate graph used for representing the given input program. We have developed our intermediate graph i.e. DADG in such a way that it can support a parallel slicing algorithm. As shown in Figure 6.4 and Figure 6.5, we generate separate DADGs for all component programs of a distributed AOP. Finally, to perform parallel slicing, we need different slicing criteria for individual component programs. In our proposed slicing algorithm, we generate a new concurrent sub-task, when we get any new slicing criterion and a separate DADG.

6.3.3 Proposed Algorithm

The context-sensitive slicing algorithm always generates more accurate and precise slices as compared to the context-insensitive slicing algorithms [16]. Moreover, the introduction of parallelism in the slicing algorithm makes the slice computation faster than the sequential slicing algorithms [103]. We propose a dynamic slicing algorithm to compute slices for a distributed AOP, which is a context-sensitive as well as parallel algorithm. We have named our proposed algorithm *Parallel Context-Sensitive Dynamic Slicing* (PCDS) algorithm. We present the pseudo code of our proposed slicing technique for distributed AOPs in Algorithm 16. Below, we explain our proposed PCDS algorithm.

Parallel Context-Sensitive Dynamic Slicing (PCDS) Algorithm

The proposed PCDS algorithm is a parallel slicing algorithm. In PCDS algorithm, we traverse all the nodes of DADG starting from the slicing criterion node. A worklist W is used for storing initial slicing criterion node. In each step, one node from W is deleted and processed by adding it in a list L. After processing of each node, it's neighbor nodes are inserted into W. During the traversal, if we encounter any R-node (i.e. Receive node) then, the PCDS algorithm automatically generates a new thread and starts computing the slice for the new component program with a new slicing criterion using the same PCDS algorithm. Here, the new slicing criterion is the respective send node of the R-node that we found during the traversal of the initial program. Also, the proposed algorithm uses three stacks that are responsible for maintaining context-sensitivity during the computation of slices. It takes the DADG for a component of a distributed AOP, as input. It maintains three stacks, i.e. S1, S2, and S3, for call-context, thread-context, and advice-context, respectively. The push operation is performed on stack S1 when we encounter any parameter-out edge during the backward graph traversal. When we find any parameter-in or call edge during the traversal, we pop from stack S1 and match the top of S1 with the current edge context. We consider the current edge to be included in the slice only if there is a match occurred. A similar approach is adopted for handling thread and weaving edges.

Algorithm 16 PCDS($DADG_i, P_i, n$)

```
1: INPUT:  $DADG_i$ = (V,E)
2:        Slicing criterion $< P_i, n >$
3: OUTPUT:  $L_i$ is the list of nodes in the computed slice, final slice
   L will be union of all $L_i$
4: W = $\{n\}$, $L_i = \{\}$  // initialize the worklist and slice set
5: Declare S1, S2 and S3 // the stacks for tracking method, thread and advice call-sites
6: Unmark all the edges in $DADG_i$
7: while W $! =$ null do
8:     Remove n from W and add to $L_i$
        /* Process communication dependence and generating parallel process */
9:     if n is a receive node (R-node) then
10:        Retrieve communication pair $< P_s, n_s >$ from n
11:        spawn $PCDS(DADG_s, P_s, n_s)$
12:    end if
13:    for all edge $e$ between m to n do // handle all incoming edges of $n$
14:        if e has not been marked then
15:            mark e
16:            Let $CS_e$ is the call-site of edge $e$
17:            if $e \in \{control, data\}$ then
18:                W.add(m) // insert the node m in W
        /* Process method call related edges */
19:            else if e $\in$ { parameter-in or call edge } then
20:                if (S1.top==$CS_e$)||(S1.top==null) then
21:                    W.add(m) // insert the new node m in W
22:                    S1.pop() // delete the node in top of S1
23:                end if
24:            else if $e \in \{parameter-out\}$ then
25:                W=W $\bigcup \{m\}$
26:                S1.push(n)
        /* Process thread related edges */
27:            else if $e \in \{thread\_start\}$ then
28:                if (S2.top==$CS_e$)||(S2.top==null) then
29:                    W=W $\bigcup \{m\}$
30:                    S2.pop()
31:                end if
32:            else if $e \in \{thread\_return\}$ then
33:                W=W $\bigcup \{m\}$
34:                S2.push(n) /* Process aspect related edges */
35:            else if $e \in \{weave\_start\}$ then
36:                if (S3.top==$CS_e$)||(S3.top==null) then
37:                    W=W $\bigcup \{m\}$
```

```
38:                 S3.pop()
39:             end if
40:         else if  $e \in \{weave\_return\}$  then
41:             W=W $\bigcup \{m\}$
42:             S3.push(n)
      /* Process intra-procedural edges */
43:         else
44:             W=W $\bigcup \{m\}$
45:         end if
46:     end if
47:   end for
48: end while
49: sync
50: Compute final slice $L = L_1 \bigcup L_2 \bigcup ... \bigcup L_e$
```

Working of PCDS algorithm

Let's consider node $S23$ of the DADG given in Figure 6.5, as the slicing criterion. This is the first part of the slice, so we take $i = 1$. Initial values of $W = \{S23\}$, $L_1 = \{\}$ and $S_1 = S_2 = S_3 = \{\}$. Now, start traversing backward from node $S23$. The status of data structures when processing different nodes is given in Table 6.1. In Table 6.1, we can observe that Stack S1 always remains null, because in our example no call edge is involved. During slice computation of the server program, when we found that S15 is a R-node, and it is included in the slice, at this point, our PCDS algorithm generates another thread and starts computing another parallel slice of the client program with slicing criterion $node12$, as this is the corresponding send node in Figure 6.4. After this point, according to our algorithm, two slices are computed parallely. The resultant slice for *MyClient* program is shown in Table 6.2. The final slice of the distributed AOPs given in Figure 6.1, Figure 6.2 and Figure 6.3, with respect to slicing criterion S23 is the union of the resultant slices shown in Table 6.1 and Table 6.2, i.e. Slice (S23) = {S23,S21,S11,S19,S10,S17,S5,S16,S33,A4,S15,S32,S31,A3,S14,S3,A2,S13,A1,S7,S6,S4, 12,11,6,9,5,4,8,3,7}. The nodes included in the resultant dynamic slice are also shown in shaded nodes in Figure 6.6 and Figure 6.7. The nodes included in computed slice, are shaded with gray color, and the nodes not included in slice are left unshaded. Figure 6.6 shows the slice of server program given in Figure 6.2 and Figure 6.3, w.r.t. slicing criterion $S23$. Similarly, Figure 6.7 shows the slice of client program given in Figure 6.1, w.r.t. slicing criterion $node12$.

Table 6.1: Status of different data structures while finding the slice of MyServer program w.r.t. ‘‘*node S23*”

Processed Node	W	S1	S2	S3	L_1
S23	{S21,S11}	{}	{}	{}	{S23}
S21	{S11,S19}	{}	{}	{}	{S23,S21}
S11	{S19,S10}	{}	{}	{}	{S23,S21,S11}
S19	{S10,S17}	{}	{}	{}	{S23,S21,S11,S19}
S10	{S17,S5}	{}	{}	{}	{S23,S21,S11,S19,S10}
S17	{S5,S16}	{}	{}	{}	{S23,S21,S11,S19,S10,S17}
S5	{S16,S33,A4}	{}	{S33}	{A4}	{S23,S21,S11,S19,S10,S17,S5}
S16	{S33,A4,S15}	{}	{S33}	{A4}	{S23,S21,S11,S19,S10,S17,S5, S16}
S33	{A4,S15,S32,S31}	{}	{}	{A4}	{S23,S21,S11,S19,S10,S17,S5, S16,S33}
A4	{S15,S32,S31,A3}	{}	{}	{A4}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4}
S15 (R-node)	{S32,S31,A3,S14}	{}	{}	{A4}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15}
S32	{S31,A3,S14}	{}	{}	{A4}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32}
S31	{A3,S14,S3}	{}	{}	{A4}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31}
A3	{S14,S3,A2}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3}
S14	{S3,A2,S13}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14}
S3	{A2,S13}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14, S3}
A2	{S13,A1}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14, S3,A2}
S13	{A1,S7}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14, S3,A2,S13}
A1	{S7}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14, S3,A2,S13,A1}
S7	{S6,S4}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14, S3,A2,S13,A1,S7}
S6	{S4}	{}	{}	{}	{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14, S3,A2,S13,A1,S7,S6}
S4	{}	{}	{}	{}	**{S23,S21,S11,S19,S10,S17,S5, S16,S33,A4,S15,S32,S31,A3,S14, S3,A2,S13,A1,S7,S6,S4}**

Table 6.2: The resultant slice of MyClient program w.r.t. slicing criterion "*node 12*"

Processed Node	**W**	L_2
	{12}	{}
12	{11,6,9}	{12}
11	{6,9,8}	{12,11}
6	{9,5,4}	{12,11,6}
9	{5,4,8}	{12,11,6,9}
5	{4,8}	{12,11,6,9,5}
4	{8,3}	{12,11,6,9,5,4}
8	{3,7}	{12,11,6,9,5,4,8}
3	{7}	{12,11,6,9,5,4,8,3}
7	{}	**{12,11,6,9,5,4,8,3,7}**

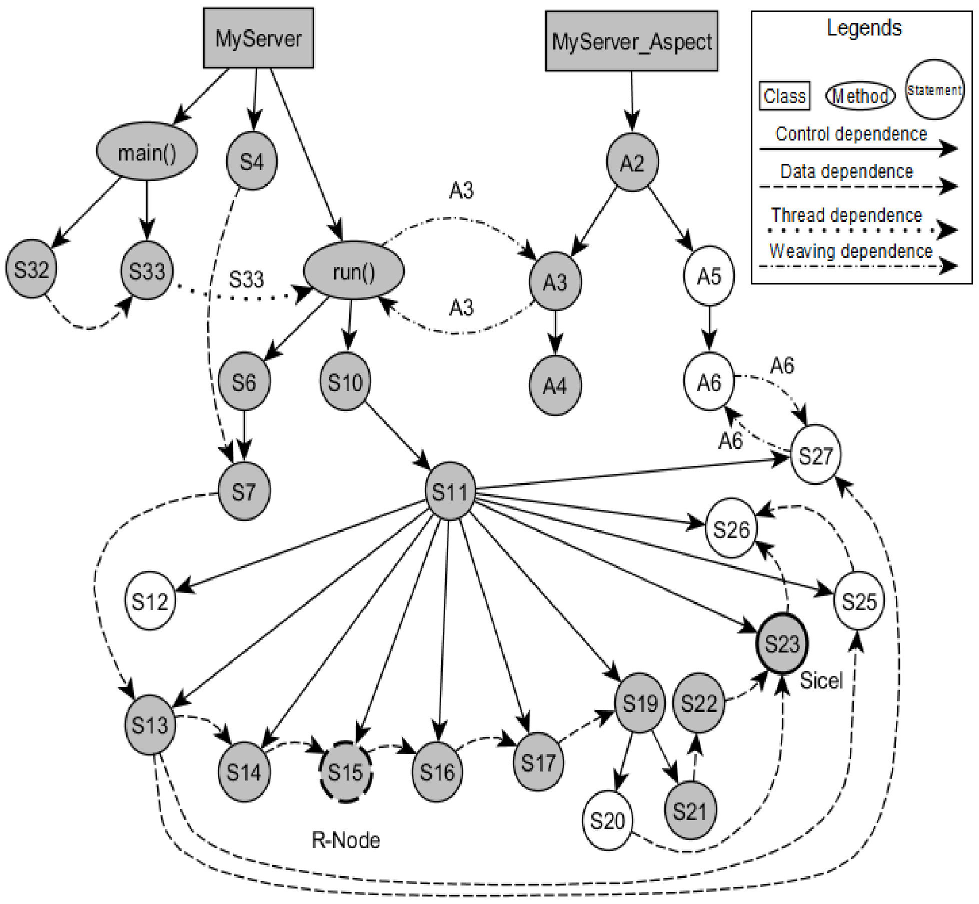


Figure 6.6: Updated DADG of the example programs given in Figure 6.2 and Figure 6.3, shaded nodes represent context-sensitive slice w.r.t. node S23

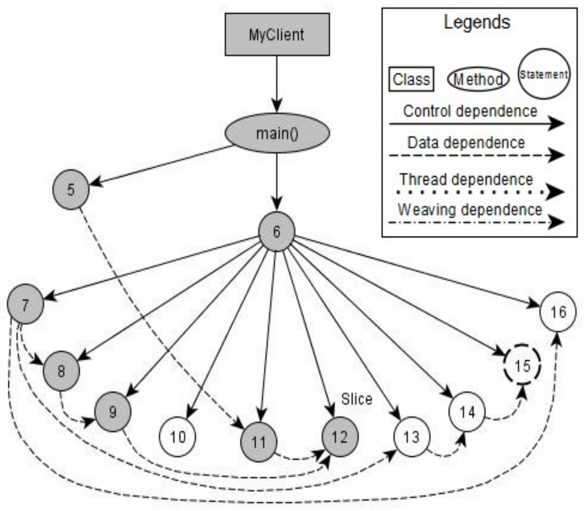


Figure 6.7: Updated DADG of the example program given in Figure 6.1, shaded nodes represent context-sensitive slice w.r.t. node 12

6.3.4 Correctness of Parallel Context-Sensitive Dynamic Slicing (PCDS) algorithm

In this section, we sketch the proof of correctness of our PCDS algorithm.

Theorem 6.3. *Parallel Context-Sensitive Dynamic Slicing (PCDS) algorithm always computes correct slices for a given distributed AOP.*

Proof: In the proof of the correctness of any algorithm, we must follow the steps of completeness, correctness, and finiteness. Hence, the proof of our algorithm consists of three parts. First, we prove that our algorithm is complete, i.e. it covers all the possible cases. Secondly, we prove that the algorithm is correct. Finally, we show that our algorithm terminates after a finite number of iterations.

For the proof of completeness of our algorithm let's consider Γ is the set of types of edges in the DADG. In our algorithm, $\Gamma = \{control, data, call, thread, weaving\}$. Initially, the intended slice consists of only the slicing criterion node s. There can be two possibilities, i.e. s may be a root node or may not be a root node. If s is a root node, then the slice will contain only the node representing the slicing criterion. If s is not a root node, then it must be connected to some other node through an edge e. According to the representation of DADG, the edge e must be of any type $\in \Gamma$. This is also true in our algorithm, because all possible types of dependencies $\in \Gamma$, are covered in the algorithm. To handle the communication dependence during the slice computation, our algorithm uses R-node. When any R-node is found during slicing, the algorithm first extracts it's corresponding sender process-id (P_s) and sender node number (n_s), which are stored at R-node during the construction of DADG. Next, our algorithm starts a new thread and begins slice computation taking the corresponding

DADG for the process-id P_s and slicing criterion as n_s. Hence, our algorithm can handle all possible types of dependencies that may be present in a distributed AOP.

We prove the correctness of our algorithm by method of mathematical induction. We assume that the current computed partial slice $S_p = \{s_1, s_2, ..., s_{i-1}\}$ is correct. We have to show that after including the next node during the traversal, S_p retained its correctness. Suppose s_i is the next node in the traversal, which is connected from s_{i-1} with an edge e_i. First, we concentrate in the method call section of our algorithm. There can be three possible locations of the two nodes (i.e. s_i and s_{i-1}), as shown below-

$$\begin{cases} \{s_{i-1}, s_i\} \in \text{same method} & \text{add } s_i \text{ to } S_p \\ s_{i-1} \in callee, s_i \in called & \text{add } s_i \text{ to } S_p\text{, push label} \\ s_{i-1} \in called, s_i \in callee & \text{pop label and match} \end{cases} \tag{6.1}$$

According to our algorithm, and as shown in Equation 6.1, if both nodes s_i and s_{i-1} lie on the same method, then we directly insert node s_i into the slice. When s_{i-1} node belongs to callee method and s_i node lies in the called method, then our algorithm checks the calling context by performing a pop operation on the stack and matching the current edge label. If the stack is empty, then also we have to add the current node s_i into the slice. In the other case, when s_{i-1} node belongs to the called method and s_i node is in the callee method, then our algorithm matches the current edge label and the top of the stack so that the calling context can be preserved. Similarly, we can show that our algorithm covers all the possible cases and finds the correct slice.

We assume that there is no cycle present in the DADG of the program. The graph is having finite number of vertices $\{1, 2, 3, ..., n\}$ and finite number of edges $\{e_1, e_2, e_3, ..., e_k\}$, where n and k are positive integers. Initially, the user enters the slicing criterion. Suppose the user has entered a slicing criterion node s and s $\in \{1, 2, 3, ..., n\}$. As there is no cycle in the graph, hence the traversal of the DADG will have finite steps. From this, we can conclude that our algorithm terminates after executing a finite number of steps. This completes the proof. □

6.3.5 Complexity Analysis

In the following we discuss the space and time complexity of the Parallel Context-Sensitive Dynamic Slicing (PCDS) algorithm.

Space complexity: The PCDS algorithm works on the input DADG. DADG is the collection of individual DADGs made for each component program for distributed AOP. Suppose there are p number of component programs exist in an input distributed AOP. Let the number of statements in each component program is n, then there will be n nodes in the DADG for each component program. Let the number of edges in each DADG is e. Then to store one DADG, we need the space for storing information of n nodes and information of each edge $\in e$. The space for storing DADG of one component program is of order $O(ne)$. But, we have

p number of component programs in a given distributed AOP, then the space complexity of total distributed AOP is $p * O(ne)$.

Time complexity: PCDS algorithm is a parallel algorithm which works on insertion and deletion from a worklist W. For finding the worst case time complexity, suppose the given DADG is a fully connected graph. As, PCDS algorithm traverses the given DADG, the worst case time complexity of PCDS algorithm should be $O(n^2)$. Suppose, there are m number of parallel slicing task on different DADGs are executing simultaneously. Then, the time complexity of whole slicing process will reduce to $O(n^2)/m$.

6.4 Implementation and Results

In this section, first we present our implementation details of the developed slicing tool named D-AspectJ slicer. Then, we discuss the details of the case studies that we have considered for our experiment. At last, we compare our proposed approach with some closely related work and present the outcomes of the experiment.

6.4.1 Setup

We have developed a partial slicing tool called D-AspectJ slicer. To develop our slicing tool and conduct the case studies, we have used, three personal computers having Intel Core *i5* processor, clock speed 2.40GHz, primary memory 4 GB and Windows 7 Home Basic (32 bit) operating system. The two computers are connected through ethernet. The findings of our study may vary if some other system configuration is used to replicate the implementation.

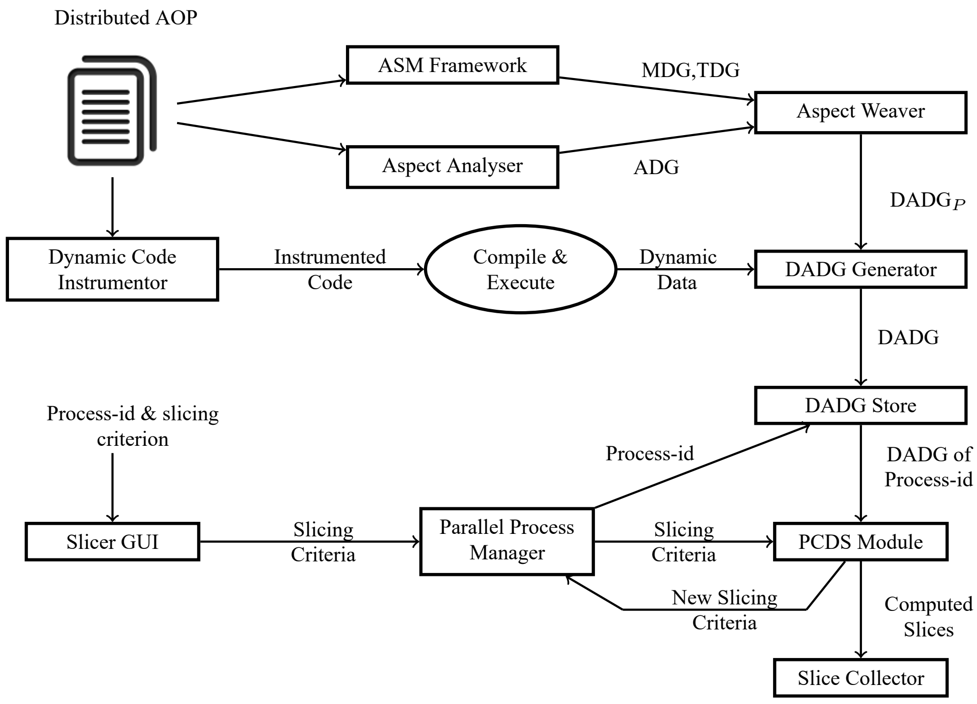


Figure 6.8: Architectural overview of D-AspectJ Slicer

6.4.2 Overview of D-AspectJ slicer

The architecture of our tool D-AspectJ slicer is given in Figure 6.8. A distributed AOP consists of three parts - a non-aspect distributed part, an aspect part and a set of dynamic communication dependencies. The input to D-AspectJ slicer is a distributed AOP. The same input distributed AOP is given to three main components of D-Aspect slicer, that are- ASM Framework, Aspect Analyzer and Dynamic Code Instrumentor. The ASM Framework is an open source JAVA code analyzer. ASM Framework is a collection of many predefined packages that perform different analysis tasks. Among them, the ‘‘internal” and ‘‘graph” packages are the most important packages. The ‘‘internal” package analyzes the basic JAVA code and gathers information about instructions, methods, classes, etc. The ‘‘graph” package uses the information provided by ‘‘internal” package and generates the intermediate representation for the whole JAVA program.

We have designed a package called ‘‘Aspect Analyser” that fetches the Aspect part of the given distributed AOP and generates the Aspect Dependence Graph (ADG). Then the ‘‘Aspect weaver” combines the non-aspect part and ADG to form a partial DADG ($DADG_P$). The $DADG_P$ is incomplete because it does not contain any data or communication dependence information. To get the dynamic data and communication dependence information, the distributed AOP is supplied to the Dynamic Code Instrumentor. It is a program that adds appropriate JAVA code into the original program such that the data and communication dependence between the statements can be captured at run-time. Then the instrumented program is compiled and executed to find the dynamic data and communication dependence information. Finally, a DADG generator accepts the $DADG_P$ and the dynamic dependence information to generate the complete DADG for the given distributed AOP. This DADG is then stored in a DADG file for future use.

When the user wants to compute a slice of the given distributed AOP, he gives the process-id and slicing criterion at the Slicer GUI. Then this information are forwarded to the Parallel Process Manager package. This package then searches the corresponding DADG from the DADG file using the given process-id. It then provides the DADG of the program and slicing criterion to the PCDS module. This package is responsible to implement our proposed slicing algorithm. The PCDS module starts computing the required slice. In the meanwhile during the slice computation, if the PCDS module finds any receive node (R-node) then it extracts the process-id of the sender node and *new slicing criterion* from the R-node and sends these to the Parallel Process Manager. The Parallel Process Manager then generates another concurrent or parallel process and searches the corresponding DADG from the DADG file. The similar process of slice computation starts again, but concurrently with the original slicing process. When all the component programs of the distributed AOP that are involved in the present slice computation are processed, the Slice Collector combines all the sub-slices of the component programs and generates the final slice of the given distributed

AOP.

Table 6.3: Details of the attributes of the Case Study Projects

Sl. No.	**Project**	**Description**	**No. of Classes**	**No. of Aspects**	**LoC**
1	Sliding-Window	Implementation of Sliding-Window protocol of data transfer	2	2	90
2	Server-Client-2	An example of server-client communication	6	1	103
3	FTP	Program for File Transfer within a network	3	2	161
4	Chess Game	Network Chess game	7	2	388
5	Online Chatting	Online chatting software	13	5	631
6	Tic-Tac-Toe	Implementation of a very popular game Tic-Tac-Toe	12	4	1107
7	Article	Implementation of a very popular messenger	12	2	1334

Table 6.4: The values of the attributes of the DADG of the Case Study Projects

Sl. No.	**Project**	**No. of Nodes**	**No. of Edges**	**No. of R-Nodes**	**Time to generate DADG (in ms)**
1	Sliding-Window	114	186	2	119
2	Server-Client-2	160	207	1	263
3	FTP	232	319	3	557
4	Chess Game	566	728	2	622
5	Online Chatting	934	1184	17	1726
6	Tic-Tac-Toe	1124	1580	4	1140
7	Article	1765	2287	4	2933

6.4.3 Case Studies

The credibility of a proposed slicing algorithm is established when it is applied and evaluated using standard case studies. As the distributed AOP is a new trend in software development, it is very difficult to find open source programs. However, we have used seven standard open source distributed JAVA projects. After analyzing all projects, we have modified and added some new Aspects into these projects, depending upon the complexity of communication among their component programs. The open source JAVA projects considered for our experiment are: *Sliding-Window Protocol Implementation*[1], *Server-Client-2*[2], *FTP Program*

[1]http://code-worm.blogspot.in/2012/10/82-java-program-fro-sliding-window.html

[2]This program is presented in [106]

[3], *Chess Game*[4], *Online Chatting*[5], *Tic-Tac-Toe* game project [6] and *Article* project [7]. The working of D-AspectJ Slicer and our slicing technique are then tested on the modified open source distributed AOP projects. The details of the case study projects are given in Table 6.3. All these case studies are open source projects and can be downloaded from the sources provided as foot notes. We have generated the DADG for these projects using our tool, and the values of the attributes of the DADG are shown in Table 6.4. It includes the number of nodes and edges in the DADG, the number of R-nodes present in each program and time taken to generate the DADG by our D-AspectJ Slicer tool.

We have computed the slices by proving different slicing criteria to our developed tool. We have implemented three algorithms, i.e. PADS algorithm proposed by Ray et al. [13], Contradictory Graph Coloring Algorithm (CGCA) proposed by Barpanda et al. [82] and our PCDS algorithm. In our experiment, we have used each algorithm to find 5-10 slices of the individual programs. We have noted the slice size and slicing time to compute the slice, for all the computed slices. Then, we have computed the average slice size and slicing time for each case study using the three different algorithms.

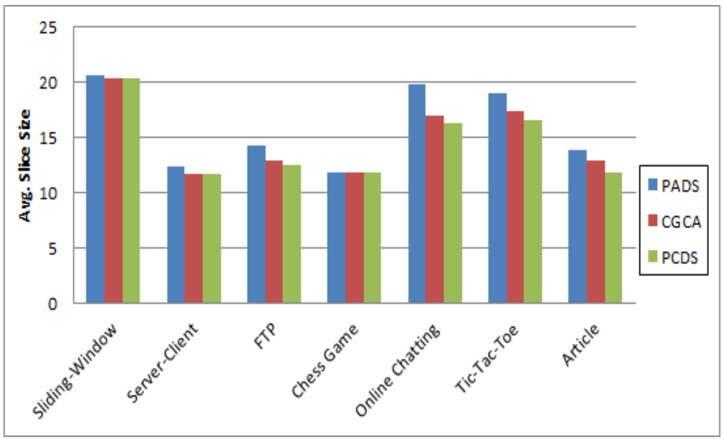


Figure 6.9: Comparison of average slice size

6.4.4 Implications

After completing the above experiments and case studies, we have analyzed the outcome to explore the hidden treasure. To produce a fare comparison of the performance of the three slicing algorithms, we have calculated the average slice size and the average slice computation time for each case study. First, we have computed number of slices using PADS algorithm [13] by taking different slicing criteria. Then, we have calculated the average slice size and average slicing time. Similarly, by taking the same slicing criteria as above, we

[3]http://www.sourcecodesworld.com/source/show.asp?ScriptID=708
[4]http://www.szic.pk/cs/chess/chess.php
[5]http://pirate.shu.edu/wachsmut/Teaching/CSAS2214/Virtual/Lectures/chat-client-server.html
[6]http://freesourcecode.net/javaprojects
[7]http://www.codeproject.com/Articles/524120/A-Java-Chat-Application

have computed slices using CGCA [82] and our proposed PCDS algorithm. Then, average slice size and average slicing time is also calculated for the two algorithms. Figure 6.9 shows the comparison of average slice size for above three algorithms. From Figure 6.9, we observe that the average slice size computed by PCDS algorithm is always smaller than PADS algorithm [13] and CGCA [82].

Table 6.5: Comparison of Slice Size

Sl.No.	Project	Avg. Slice Size			Percentage Change	
		PADS [13]	CGCA [82]	PCDS (proposed)	DCR_1	DCR_2
1	Sliding-Window	20.6	20.04	20.04	2.71%	0%
2	Server-Client-2	12.34	11.71	11.71	5.1%	0%
3	FTP	14.22	12.9	12.45	12.44%	3.48%
4	Chess Game	11.83	11.83	11.83	0%	0%
5	Online Chatting	19.82	17.02	16.29	17.81%	4.28%
6	Tic-Tac-Toe	18.94	17.33	16.58	12.46%	4.32%
7	Article	13.89	12.9	11.76	15.33%	8.83%

In Table 6.5, we have used DCR_1 and DCR_2 whose definition is given below:

$$DCR_1 = (PADS - PCDS)/PADS * 100 \tag{6.2}$$

where, DCR_1 is *Percentage decrement of avg. slice size in PCDS as compared to PADS*.

$$DCR_2 = (CGCA - PCDS)/CGCS * 100 \tag{6.3}$$

where, DCR_2 is *Percentage decrement of avg. slice size in PCDS as compared to CGCA*.

Table 6.5 shows the comparison of average slice size. It shows that, average slice size of slices computed by our PCDS algorithm is 0% to 17.81% smaller than that of PADS algorithm [13]. Also, we observed that the average slice size of slices computed by our PCDS algorithm is 0% to 8.83% smaller than that of CDCA [82]. Based on the average slice size given in Table 6.5, we compare the efficiency of all three algorithms. We found that, our proposed PCDS algorithm generates on an average 9.4% smaller slices than PADS algorithm [13]. Also, PCDS algorithm generates 2.98% smaller slices than CGCA [82].

Figure 6.10 shows, comparison of average slicing time for PADS algorithm [13], CDCA [82] and our PCDS algorithm. We found that our PDPS algorithm computes slices much faster than PADS algorithm and CGCA algorithm, as shown in Figure 6.10.

We compare the three algorithms with respect to the average slice computation time. Table 6.6 shows the comparison of average slice computation time. We find the percentage decrease in average slicing time of proposed PCDS algorithm in comparison with the average slicing time of PADS algorithm [13] and CGCA [82]. From Table 6.6, we find that PDPS algorithm generates slices 7.85% to 24.54% faster than PADS algorithm [13]. Also, we find

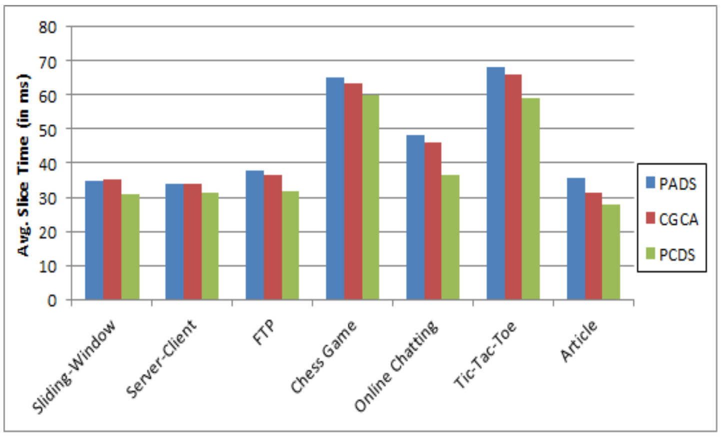


Figure 6.10: Comparison of average slice time in milliseconds

that the proposed PDPS algorithm compute slices 5.33% to 20.82% faster than CGCA [82]. From Table 6.6, we find that on an average our PCDS algorithm compute slices 14.68% faster than PADS algorithm and 11.51% faster than CGCA.

Table 6.6: Comparison of Slicing Time

Sl.No.	Project	Avg. Slicing Time (in ms)			Percentage Change	
		PADS [13]	CGCA [82]	PCDS (proposed)	DCR$_3$	DCR$_4$
1	Sliding-Window	34.6	35.2	31	10.4%	11.93%
2	Server-Client-2	34	33.83	31.33	7.85%	7.38%
3	FTP	37.72	36.54	31.54	16.38%	13.68%
4	Chess Game	65.08	63.21	59.84	8.05%	5.33%
5	Online Chatting	48.16	45.9	36.34	24.54%	20.82%
6	Tic-Tac-Toe	68.33	65.85	59.17	13.4%	10.14%
7	Article	35.55	31.19	27.66	22.19%	11.31%

In Table 6.6, we have used DCR$_3$ and DCR$_4$ whose definition is given below:

$$DCR_3 = (PADS - PCDS)/PADS * 100 \tag{6.4}$$

where, DCR$_3$ is *Percentage decrement of avg. slice size in PCDS as compared to PADS*.

$$DCR_4 = (CGCA - PCDS)/CGCS * 100 \tag{6.5}$$

where, DCR$_4$ is *Percentage decrement of avg. slice size in PCDS as compared to CGCA*. From the above discussion we, can infer the following:

- The proposed PCDS algorithm generates slices of smaller sizes or almost same sizes, than the slices produced by PADS algorithm and CGCA algorithm. Hence PCDS algorithm computes more precise slices.

- The time taken to generate slices is always found to be minimum while using PCDS algorithm. Hence it is faster than PADS and CGCA slicing algorithms.
- We observed that when a distributed AOP project have more number of $R{-}nodes$ in its DADG, then our PCDS algorithm works more efficiently than the rest two algorithms.

6.4.5 Threats to validity

In this section, we present some of the threats to the validity of our proposed approach.

1. As the slicing technique proposed in this chapter is only for AspectJ platform; it may not work for other aspect-oriented programming languages such as Aspect C++ and Aspect C#.
2. Through our experimental study, we have tested our proposed slicing technique for computing precise and correct slices of projects upto 1400 LOCs. We believe that, the other larger projects with similar structure in the same language, may be handled with our slicing technique.
3. The proposed slicing technique is based on the construction of intermediate representation (IR), and if the IR changes, then our slicing technique may not work properly.

6.5 Comparison with related work

In-order to establish the efficiency of our proposed slicing algorithm, we have implemented and compared our proposed slicing approach with two most closely related slicing techniques. First, we compare our work with the Parallel Aspect-Oriented Dynamic Slicing (PADS) algorithm proposed by Ray et al. [13]. Their algorithm is based on marking and un-marking of the edges of a Distributed Aspect-oriented Program Dependence Graph (DAPDG). Our proposed algorithm is based on a dynamic graph DADG, which is smaller in size than the static dependence graph proposed by Ray et al. [13]. Also they created a new logical node called C-node to represent the communication dependence between the component programs of a distributed program. There are two disadvantages of this approach. First, some new extra C-nodes are created, and second they are using extra communication edges. Whereas, in our DADG, we neither use any extra node nor create any communication edge. Also, our algorithm is a parallel and context-sensitive algorithm which can compute precise slices in little time.

Next, we have compared our proposed work with a slicing approach proposed by Barpanda et al. [82] called Contradictory Graph Coloring Algorithm (CGCA). Their algorithm is a sequential algorithm. Hence much time consuming. Also, they have not considered aspect-oriented features and context-sensitivity in their proposed algorithm. We have extended CGCA to handle AOP features, so that we can compare our proposed technique with it. We have implemented all the three approaches; i.e, our slicing approach, Ray et al. approach [13] and the approach proposed by Barpanda et al. [82], using our developed tool.

Ray et al. [13] have proposed a dynamic slicing algorithm for distributed AOPs. They have used an intermediate graph called Distributed Aspect-oriented Program Dependence Graph (DAPDG) to represent the aspect-oriented features. DAPDG is created for each current execution trace, starting from scratch. Next time for a different execution trace, another new DAPDG is formed. In our approach, we use the previously available information of the DADG. They [13] have proposed a Parallel Aspect-oriented Dynamic Slicing (PADS) algorithm that extends the existing Node Marking Dynamic Slicing (NMDS) algorithm, proposed by Mohapatra et al. [106]. But, the PADS algorithm does not represent the non-determinism behavior of the distributed AOPs. Also, the size of DAPDG is large because it stores the relative local slice on each node of the graph. The work of Ray et al. is most closely related to our proposed slicing technique. Hence, we have implemented the approach of Ray et al. and compared it with our proposed approach in Section-6.4.

6.6 Summary

In this chapter, we proposed an intermediate graph called Distributed Aspect Dependence Graph (DADG) to represent distributed AOPs. DADG represents the features of a DAOP. Based on this intermediate representation, we designed a Parallel Context-sensitive Dynamic Slicing (PCDS) algorithm for distributed AOPs. To evaluate our proposed algorithm, we have designed a D-AspectJ slicer. We presented seven case studies of open source projects. We have implemented our approach, PADS algorithm proposed by Ray et al. [13] and CGCA proposed by Barpanda et al. [82]. Using the case studies, we have compared the preciseness of all the three approaches in terms of slice size and slicing time. We found that our PCDS algorithm generated on an average 9.4% smaller slices than the PADS algorithm and 2.98% smaller than the CGCA slicing algorithm.

We have also compared the slice computation time of all the three algorithms and we found that our PCDS slicing algorithm runs faster than the approaches of Ray et al. [13] and Barpanda et al. [82] by 14.68% and 11.51% respectively. So, we conclude that our slicing technique computes more precise slices in little time for distributed AOPs.

Program slicing has many applications such as debugging, testing, software maintenance, software refactoring etc. In the next chapter, we present an approach for software refactoring using program slicing.

Chapter 7

Software Refactoring using Program Slicing

Any software in the real-world emerges by acquiring new requirements, new techniques, and newly changed scenarios. In the early days, the need for change in software arised at long intervals of time. But in the present days, the requirements change at small intervals of time. In order to incorporate all these changes, the code of the software undergoes many modifications and additions. As a result, the code of the software becomes more and more complex and drifts away from its original design [31]. This drifting of code from design affects all the descendant activities, like testing, maintenance, etc., to be followed correctly.

In order to solve the above-said problem, we need a technique that will incorporate all the evolving requirements and changes, and simultaneously ensure the quality of the software [86]. One such technique is software refactoring. Software refactoring as defined in [31] is *‘‘the process of changing an object-oriented software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure”*. In this process, we redistribute the classes, variables and methods, to make the overall software low complex and of better quality.

But, refactoring of the full software is a tedious task in terms of time and cost involved in it [31]. There exists many research papers dealing with the refactoring techniques [86, 87, 107]. However, these papers do not reveal the techniques used to select the target methods for refactoring. We observe that instead of refactoring all the modules of the software, we should refactor only that modules that need refactoring. Also, we know that the software quality can be evaluated using software metrics [108]. Hence, we use *slice-based cohesion metrics* [42] to identify that modules that need refactoring. We compute the cohesion of each module in the software and then check their cohesion metric values. If some modules has less cohesion metric value than the acceptable threshold, then that module is restructured. In this chapter, we address the modules fit for restructuring as target modules.

Now the problem is to develop a technique to refactor the target modules such that it reduces their complexity. Here, *program slicing* and Aspect-Oriented Programming (AOP) come into the picture. Each target module is sliced into a number of slices by taking the output variables [109] as slicing criteria. Then, among the resultant slices, the most similar slices are combined to form a new module. The new modules that are obtained are defined as *advices* of an *aspect*. As a result of this technique, the target module will produce the same output at a reduced complexity.

The primary objective of this paper is to develop a technique for code refactoring. We need to identify that methods, which need refactoring. We show that the slice-based cohesion metrics are useful in this task. The division of one method into two partitions is a difficult task. In this chapter, we show that program slicing is useful for partitioning a method and how AspectJ programs are used to restructure the method code into different methods.

This chapter is structured as follows: In Section 7.1, we discuss the basic concepts to understand our work. Section 7.2 introduces our proposed refactoring approach. Also in this section, we propose the refactoring algorithms and present the working of our proposed refactoring algorithm along with an example. In Section 7.3, we discuss the experimental evaluation of our proposed refactoring approach by taking some JAVA projects. In Section 7.4, we present the comparison of our proposed approach with some related works. Section 7.5 present the summary of work done in this chapter.

7.1 Basic Concepts

In this section, we present the basic concepts that are required to understand our proposed approach. Below, we present the concept of slice-based cohesion metrics.

7.1.1 Slice-Based Cohesion Metrics

Software metrics are used to quantify the complexity of software [32]. Slice-based cohesion metrics were proposed by Weiser [42]. He informally presented five slice-based metrics: *Tightness, Coverage, Overlap, Parallelism,* and *Clustering*. Out of these five, *Parallelism* and *Clustering* are highly co-related with tightness, coverage, and overlap. Hence, we can drop these two metrics [32]. Two more metrics: *MinCoverage and MaxCoverage* were proposed by Ott and Thuss [110]. The formalization of the five metrics are shown below [32]:

$$Tightness(M) = \frac{|SL_{int}|}{lenght(M)} \tag{7.1}$$

$$MinCoverage(M) = \frac{1}{lenght(M)} min_i\,|SL_i| \tag{7.2}$$

$$Coverage(M) = \frac{1}{|V_o|}\sum_{i=1}^{|V_o|}\frac{|SL_i|}{lenght(M)} \tag{7.3}$$

$$MaxCoverage(M) = \frac{1}{length(M)} max_i\,|SL_i| \tag{7.4}$$

$$Overlap(M) = \frac{1}{|V_o|} \sum_{i=1}^{|V_o|} \frac{|SL_{int}|}{|SL_i|} \tag{7.5}$$

where M is the module under consideration, V_m is the set of variables in M, V_o is the set of output variables, SL_i is the slice obtained for $v_i \in V_o$, and SL_{int} is the intersection of SL_i over all $v_i \in V_o$.

After analyzing 63 programs, Meyers et al. [32] stated that there is a strong correlation between *tightness* and *minCoverage*, between *minCoverage* and *overlap* and between *tightness* and *overlap*. Hence, it is not necessary to compute all the metrics. Depending upon this analysis, they [32] gave benchmark values for overlap that lies between 0.6908 and 1.0, and the benchmark value for tightness lies between 0.2973 and 0.3039. We use these values in Section 7.2 for explaining our refactoring approach.

7.2 Our Proposed Approach

In this section, we present our proposed refactoring approach. First, we describe the proposed refactoring approach with the help of a block diagram. We explain the function of each component of the block diagram. Then, we present the proposed algorithm for refactoring of a given program. Our proposed algorithm is a collection of three algorithms. The first algorithm is the main algorithm that in turn calls the other two algorithms. The second algorithm calculates the cohesion metrics for a given method and the third algorithm splits the target method into two parts- an *advice* and a *method*.

7.2.1 Block Diagram of our proposed approach

The block diagram of our proposed approach is given in Figure 7.1. This is a collection of seven basic blocks of our approach, and it shows the stepwise flow of activities that must be carried out to perform code refactoring.

As shown in Figure 7.1, the class file of the JAVA program i.e. the byte code of the program, is given as input. The SDG Constructor (Block-1) produces the SDG for the whole JAVA class. The SDG consists of representations for all the methods in the class. But, we need SDG for each method individually. So, we give the SDG of the whole class to the *Modularizer*.

Modularizer component (Block-2) takes the SDG of JAVA class file as input and produces the *Procedure Dependence Graph* (PDG) for individual methods in the class. Then, we need to compute the slice-based cohesion metrics for each method. Using Equations 7.1-7.5 given in Section 7.1 [32], the *METRICS CALCULATOR* computes the values of the different slice-based cohesion metrics (Block-3). According to the research carried out in [32], among the five slice-based cohesion metrics, tightness and overlap are most important and if these two are computed, then rest three can be covered. Hence, in our approach, we

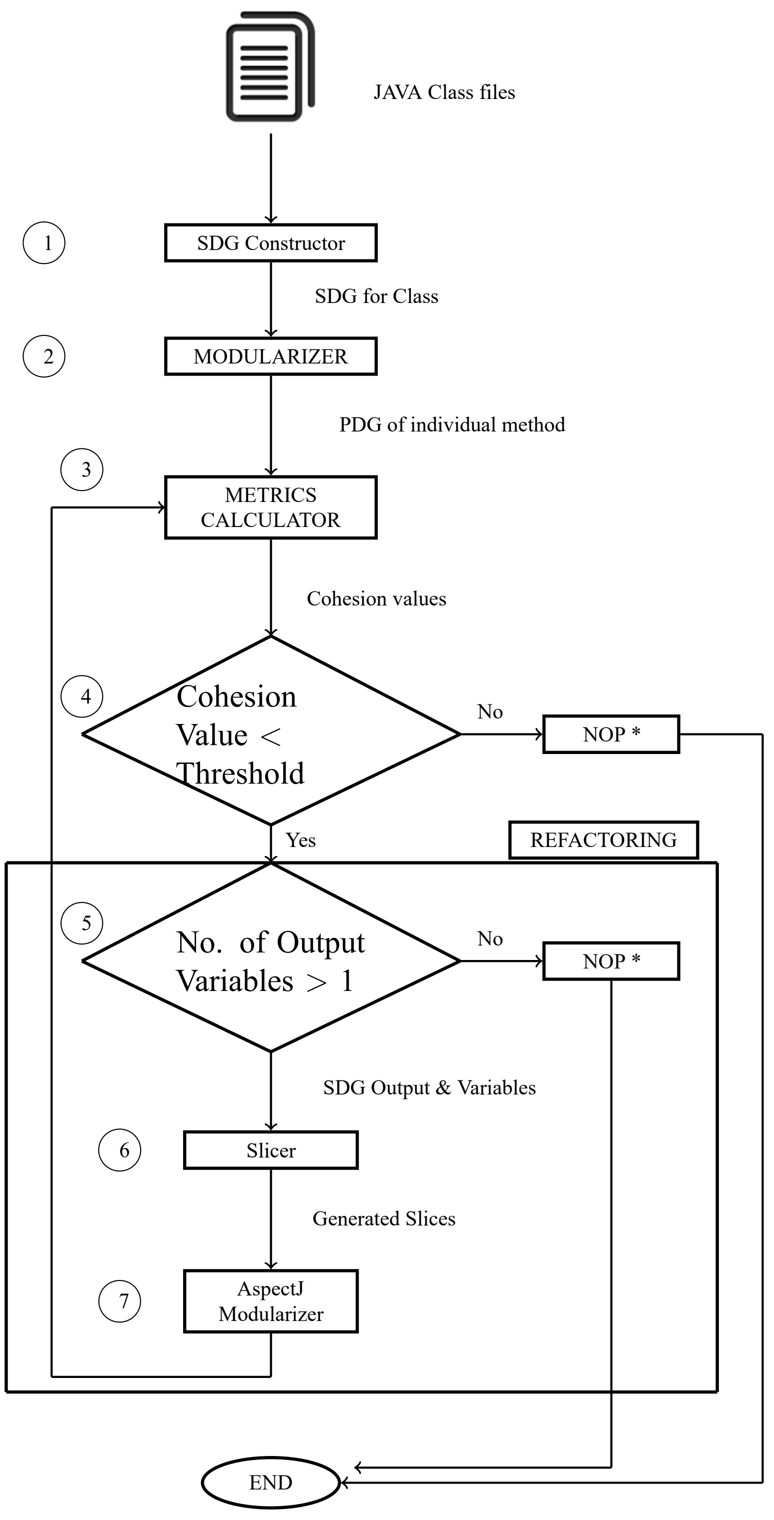


Figure 7.1: Block Diagram of our approach, where NOP*: No Operation

consider tightness and overlap to simplify the technique and decrease the computation time.

Now, the values of the computed cohesion metrics are compared with the threshold benchmark values (Block-4), to find the methods that need refactoring. According to the research conducted by [32], the tightness of any method should not be less than 0.3039. Similarly, the value of overlap should not be less than 0.6908. Hence in our approach, we have taken those values (i.e. 0.3039 for tightness and 0.6908 for overlap) as threshold values to identify the methods that require refactoring.

In refactoring, we consider each method identified during the above process. We first fetch the number of output variables present in the method. For simplification, we have considered only the printed variables as output variables. If the number of output variables is greater than 1 (Block-5), then only refactoring is possible. Then, the output variables and SDGs of the methods are supplied to the Slicer component (Block-6). The Slicer component computes the slices of the input method w.r.t. the output variables.

Now, the computed slices are given as input to the AspectJ Modularizer component (Block-7). The main task of AspectJ Modularizer is to increase the cohesion by creating an Aspect and fitting the computed slices into the Aspect. Here, we create an aspect using AspectJ program. Aspects can contain advices, which are similar to the methods in Java. So, we accommodate the computed slices into the aspects as their advices. After creation of the aspects, we remove the codes in the slice from the original method. After removing the codes from the original method, we get the modified method.

After completion of the above mentioned steps, we are at the end of one iteration of the refactoring process. Again, we have to compute the values of the cohesion metrics for all the methods using Equation 7.1-7.5 and check whether these values agree with the threshold values or not. We repeat the above steps again till the cohesion metrics of all the methods satisfy the threshold benchmark values. If all the methods satisfy the prescribed threshold benchmark values, then the process of refactoring is stopped.

7.2.2 Proposed Algorithm for Software Refactoring

In this section, we discuss our methodology for refactoring of object-oriented software. We propose a *Refactoring* algorithm (i.e. Algorithm 17), that takes the SDG of the target program (that we want to refactor) and the threshold values for slice-based cohesion metrics, as input. It then forms the PDG by deleting the call and parameter passing edges from the given SDG. Then the algorithm calls *Compute_Cohesion* algorithm (i.e. Algorithm 18), which takes the PDG, n, and output as the input and calculates the slice-based cohesion metrics for each PDG. These computed cohesion metrics values are returned to *Refactoring* algorithm (Algorithm 17).

After checking the cohesion metrics values (tightness and coverage) for each method, against the corresponding threshold values, the *Refactoring* algorithm decides which methods must be refactored. It then calls another algorithm called *Binary_Refactoring*

Algorithm 17 Refactoring (SDG, threshold)

```
Require: SDG$< Node, Edge >$
Require: Threshold for $< tightness, coverage >$

  -generate pecedure dependence graph (PDG) from SDG
  repeat
    -identify output variable nodes $n$
    -call Compute_Cohesion (PDG,n)

    if Threshold values $>$ $(tightness, coverage)$ then
      $-callBinary\_Refactoring(PDG, n)$
    end if
  until all methods are processed
```

Algorithm 18 Compute_Cohesion (PDG,n,output)

```
Require: PDG$< Node, Edge >$
Require: Integer n
Require: List of output variables

  for i=1 to n do
    -compute the slice taking the output $[i]$ as slicing criterion
    - store each node of the slice in List $[i]$
  end for
  -compute:
        $SL_i$ = number of nodes in the List$[i]$
        $SL_{int}$ = number of common nodes in all the List$[]$
        $lenght(M)$= number of nodes in the PDG
  -compute cohesion metrics using following formulae:
        $Tightness(M) = |SL_{int}| / lenght(M)$
        $Coverage(M) = \frac{1}{n} \sum_{i=1}^{n} \frac{|SL_i|}{lenght(M)}$

  return (tightness,coverage)
```

Algorithm 19 Binary_Refactoring (PDG,n)

```
Require: PDG< $Node, Edge$ >
Require: Integer n
  for i=1 to n do
    -compute the slice taking the output [$i$] as slicing criterion
    - store each node of the slice in List [$i$]
  end for
  -compute m=$\lfloor n/2 \rfloor$
  -find $List_{aspect} = \bigcup_{i=1}^{m} List[i]$ most common nodes
  -declare an Aspect with pointcut (call current method)
  -declare an after() advice into the Aspect
  -move codes represented by $List_{aspect}$
  from current method to $after()$ advice
```

(Algorithm 19). The *Binary_Refactoring* algorithm then splits the identified methods into two parts, one of which is a new *advice* in an aspect program, and the other is the given module itself having low complex code. This process is recursively carried out till the cohesion metrics values (tightness and coverage) of the resultant modules are greater than the corresponding threshold values.

7.2.3 Working of the Algorithm

We have considered an example program shown in Figure 7.2, that calculates the values of five cohesion metrics, i.e. tightness, minCoverage, coverage, maxCoverage, overlap, and also displays their values [109]. The byte-code of the class $Metrics$ is given as input to the SDG Generator Tool (Block-1). Our tool analyses the byte-code of the class file and produces the SDG for the given program, as shown in Figure 7.3.

Now the user has to enter the threshold values for the cohesion metrics. The benchmark values for only tightness and overlap are maintained in [32]. In our approach, we consider the threshold values for tightness as 0.3039 and for overlap as 0.6908.

Working of Refactoring Algorithm

We supply the generated SDG and the threshold values for tightness and coverage, as input to the Refactoring algorithm.

The algorithm first searches for the *class dependence edges*, *call edges* and *parameter edges* in the given SDG. It deletes all Class and Parameter edges and separately generates Procedure Dependence Graphs (PDG) for each method present in the program, as shown in Figure 7.4.

Then the algorithm processes all PDGs one-by-one. It first identifies all the output nodes

```
class Metrics{

public static void  main(String agrs[]){
        int []sizes = {12,6,8,8,10,6};
        int module = 20;
        int slint = 4;
        int vo = 6;
        calcMetrics(sizes, module, slint, vo);
        }

static void calcMetrics(int []sizes, int module, int slint, int vo)
{
        double min = 99999.9;
        double max = 0.0;
        int sumslice = 0;
        double tightness, coverage, overlap;
        int i;
        for (i = 0; i <vo; i++)
        {
                if (sizes[i] <min) min = sizes[i];
                if (sizes[i] >max) max = sizes[i];
                 sumslice = sumslice + sizes[i];
        }
        tightness = (double) slint / module;
        coverage = (double) sumslice / (vo * module);
        overlap = calcOverlap (sizes, slint, vo);
        min = min / module;
        max = max / module;
        System.out.println("tightness ="+ tightness);
        System.out.println("coverage = "+ coverage);
        System.out.println("min-coverage ="+ min);
        System.out.println("max-coverage ="+ max);
        System.out.println("overlap ="+overlap);
}

static double calcOverlap (int sizes[], int slint, int vo)
{
        double total = 0.0;
        int j;
        for (j = 0; j <vo; j++)
                total = total + (float) slint/sizes[j];
        return total/vo;
}
}
```

Figure 7.2: An example program for calculating the values of cohesion metrics

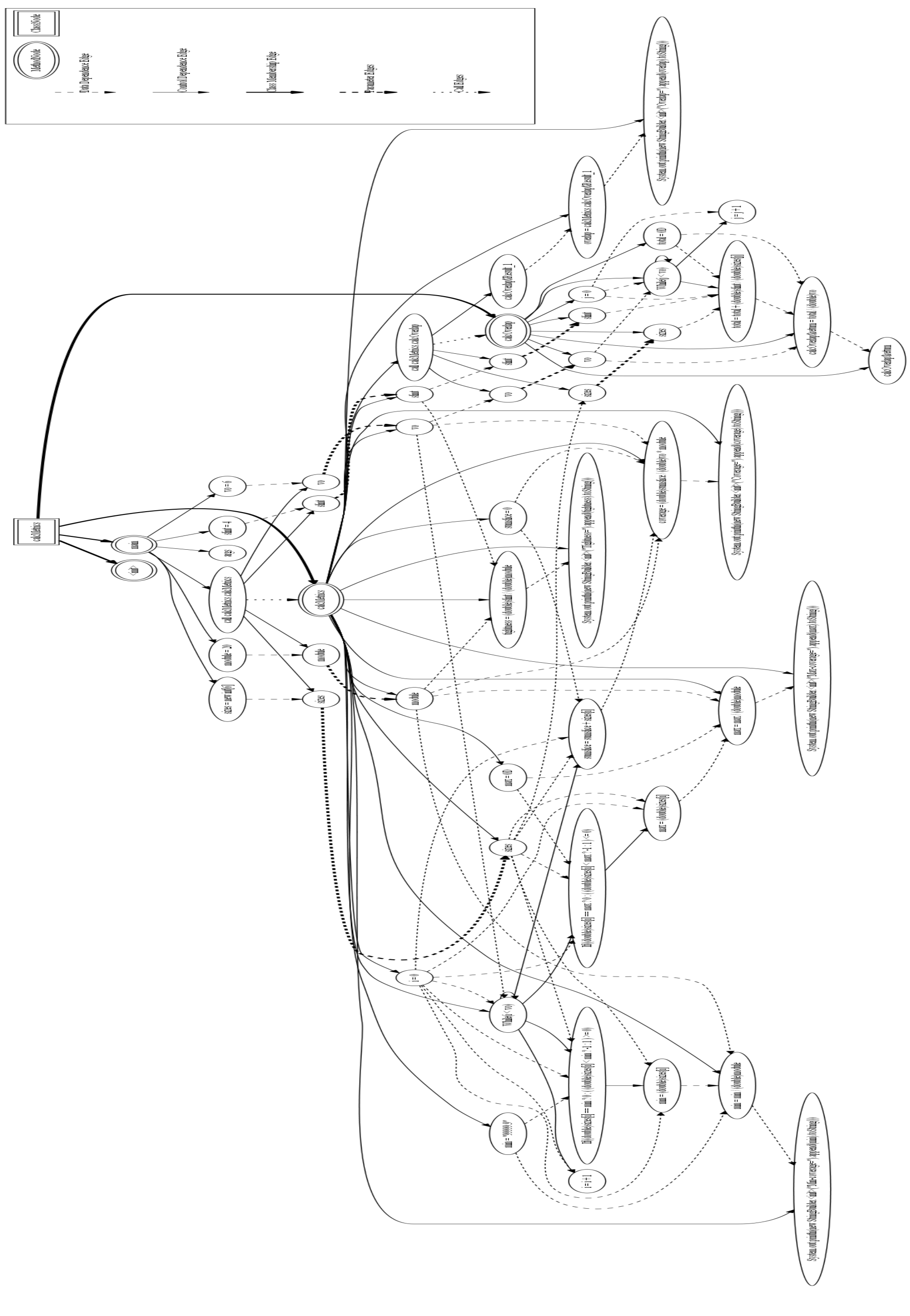
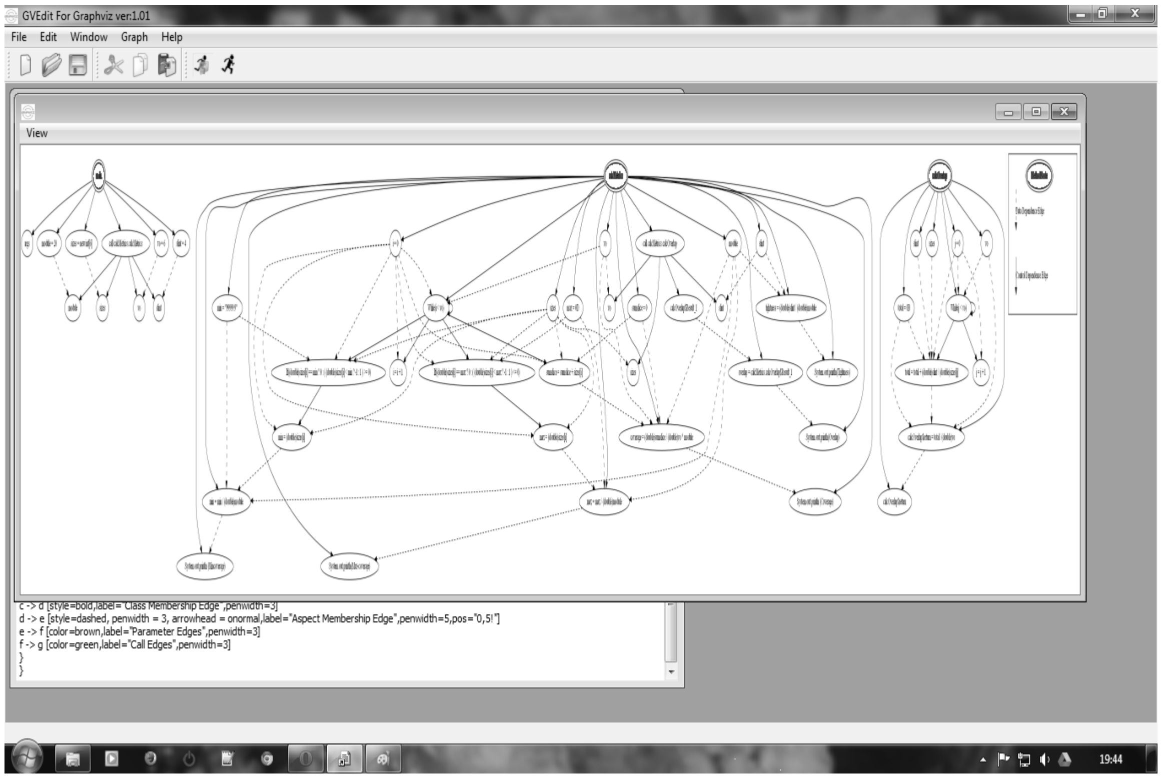


Figure 7.3: SDG of the example program given in Figure 7.2



```
c -> d [style=bold,label="Class Membership Edge",penwidth=3]
d -> e [style=dashed, penwidth = 3, arrowhead = onormal,label="Aspect Membership Edge",penwidth=5,pos="0,5!"]
e -> f [color=brown,label="Parameter Edges",penwidth=3]
f -> g [color=green,label="Call Edges",penwidth=3]
}
}
```

Figure 7.4: PDGs of the example program given in Figure 7.2

from the PDG of the current method. For simplification of explanation, we consider the output nodes as that nodes which have out-degree=0. The algorithm counts such nodes and stores them in a variable *n*.

Then, Refactoring algorithm calls Algorithm 18 (*Compute_Cohesion* algorithm) to compute the tightness and coverage of each method. Once it gets that values, then it compares them with the threshold values provided by the user. If the values of the metrics for a method, are less than the corresponding threshold values, then it refactors that method by calling *Binary_Refactoring()* algorithm.

Working of Compute_Cohesion Algorithm

The cohesion metrics for a method are computed by calling *Compute_Cohesion()* algorithm. The Compute_Cohesion algorithm takes the following as input: the PDG of the current method, *n* and the output nodes. It then computes the slices from the PDG by taking each output node as the slicing criterion. Hence, we get n number of slices which are stored in an array called List[]. The detailed analysis for all the three methods in the example program is shown in Table 7.1. Then, the algorithm computes the tightness and overlap values according to the equations given in Section 7.1. Table 7.2 shows the computed metrics values for the example program. These values are returned to the callee algorithm, i.e. Refactoring().

Table 7.1: Analysis of node-based intraprocedural slices

Method name	**length(no. of nodes in PDG)**	**Output node**	**Nodes in the slice List** [ ]	SL_i	**Intersection nodes**	SL_{int}
calcMetrics	31	40	$\{14, 30, 16, 17\}$	4		
		41	$\{14, 31, 16, 18, 21, 28, 15, 22, 23\}$	9		
		42	$\{14, 38, 16, 19, 25, 15, 22, 24, 23, 18\}$	10		
		43	$\{14, 39, 16, 20, 27, 15, 22, 26, 23, 18\}$	10		
		44	$\{14, 37, 36, 32, 34, 17, 33, 15, 35, 18\}$	10	$\{14\}$	1
calcOverlap	11	49	$\{45, 55, 48, 50, 53, 46, 47, 51, 52\}$	10	All	10
main	11	None	0	0	0	0

Table 7.2: Original Slice-based Metrics calculated for the methods of the example program given in Figure 7.2

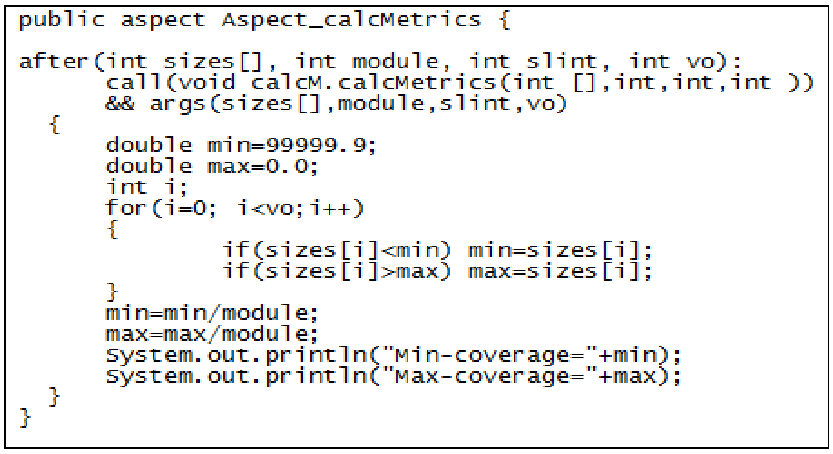
Metrics	**main()**	**calcMetrics()**	**calcOverlap()**
Tightness	1	0.032	1
Coverage	1	0.213	1
Min-Coverage	1	0.129	1
Max-Coverage	1	0.322	1
Overlap	1	0.132	1

Working of Binary_Refactoring Algorithm

Now, after checking the values of cohesion metrics of all methods, we observed that only calcMetrics() fails to satisfy the threshold values. Hence, it must be refactored. Now, the Refactoring algorithm calls *Binary_Refactoring()* algorithm for the method calcMetrics. Binary_Refactoring algorithm takes the PDG of the method calcMetrics and the number of output variable nodes (*n*) as the input and computes the slices of the PDG for the method calcMetrics.

As the name suggests, this algorithm divides the given method into two parts. According to this algorithm, we compute the value of *m*, as the lower bound of *n/2*. Now, we have to choose *m* number of slices, that are having most common nodes. Then, we find the union of these *m* slices and store the result in an array $List_{Aspect}$. Now, we have to create an aspect using any aspect-oriented programming language. We have used AspectJ for creating the aspects. Then, within the aspect, we declare an advice *after()*, that handles the call of the target method. Initially, advice *after()* does not contain anything.

Next, we have to identify the codes, in the target method, that are represented by nodes of $List_{Aspect}$ in the PDG. Move all the codes from the body of the method, into the advice *after()*. After the execution of *Binary_Refactoring()* algorithm, the newly created *aspect* will look like the code as shown in Figure 7.5. The code for the modified *calcMetrics* is shown in Figure 7.6. We have calculated the slice-based cohesion metrics for the newly created advice *after()* and the modified calcMetrics(), as shown in Table 7.3 and found that now all cohesion metrics values are within the threshold. It completes one iteration of refactoring



```
public aspect Aspect_calcMetrics {

after(int sizes[], int module, int slint, int vo):
      call(void calcM.calcMetrics(int [],int,int,int ))
      && args(sizes[],module,slint,vo)
  {
      double min=99999.9;
      double max=0.0;
      int i;
      for(i=0; i<vo;i++)
      {
              if(sizes[i]<min) min=sizes[i];
              if(sizes[i]>max) max=sizes[i];
      }
      min=min/module;
      max=max/module;
      System.out.println("Min-coverage="+min);
      System.out.println("Max-coverage="+max);
  }
}
```

Figure 7.5: The newly created aspect for the method calcMetrics of the example program given in Figure 7.2, after refactoring

and it repeats again.

Table 7.3: Updated Slice-based Metrics for calcMetrics() and after() after one iteration of refactoring

Metrics	**calcMetrics()**	**after()**
Tightness	0.04	0.333
Coverage	0.306	0.588
Min-Coverage	0.16	0.588
Max-Coverage	0.4	0.588
Overlap	0.153	0.571

Theorem 7.1. *Refactoring algorithm proposed in this thesis is correct and terminates after finite time.*

Proof: The main controlling algorithm in our approach is *Refactoring()* algorithm (Algorithm 17). In the proof of the correctness of any algorithm, we must follow the steps of completeness, finiteness, and correctness. Hence, the proof of our algorithm consists of three parts. First, we prove that our algorithm is complete, i.e. it covers all the possible cases. Secondly, we show that our algorithm terminates after a finite number of iterations. Finally, we prove that the algorithm is correct.

For the proof of completeness of our algorithm, let's assume that the SDG for a given program is generated and readily available to us. *Refactoring()* algorithm takes this SDG as input. All the call edges of SDG are removed, in order to get the PDGs. Suppose, we get p number of PDGs out of the given SDG after removing all call edges. Each PDG represents one method in the input program. It means that there are p number of methods in the given input program. *Refactoring* algorithm repeats the intended process for p times. It shows that all the methods present in a given program are handled by our algorithm. Hence, our proposed refactoring algorithm is complete.

```
public class calcM {
       public static void main(String args[])
       {
               int sizes[]= {12,6,8,8,10,6};
               int module=20;
               int slint=4;
               int vo=6;
               calcMetrics(sizes,module,slint,vo);
       }
public static void calcMetrics( int sizes[], int module, int slint, int vo)

 {
       int sumslice=0;
       double tightness, coverage,overlap;
       int i;
       for(i=0; i<vo;i++)
       {
               sumslice=sumslice+sizes[i];
       }
       tightness=(double)slint/module;
       coverage=(double)sumslice/(vo*module);
       overlap=calcOverlap(sizes,slint,vo);


       System.out.println("Tightness="+tightness);
       System.out.println("Coverage="+coverage);
       System.out.println("Overlap="+overlap);
       }

 static double calcOverlap( int sizes[],int slint, int vo)
       {
               double total=0.0;
               for( int j=0; j<vo;j++)
               {
               total=total+(double)slint/sizes[j];
               }
       return total/vo;
       }
       }
```

Figure 7.6: Code for the modified method calcMetrics after refactoring

Refactoring algorithm itself terminates after *p* iterations, as shown above. *Refactoring* algorithm calls two more algorithms, i.e. *Compute_Cohesion()* and *Binary_Refactoring()*. *Compute_Cohesion()* algorithm does only the calculation of different slice-based cohesion metrics values, and it terminates after the computation. The other algorithm is *Binary_Refactoring()*, which computes the slices of the given PDG and groups them into two modules. As there are finite number of slices computed by this algorithm, we can assure that after finite time the *Binary_Refactoring()* algorithm terminates. As a result the whole refactoring process terminates after finite execution time.

We prove the correctness of our algorithm by *Proof by Cases*. In the *Refactoring()* algorithm, first we compute the cohesion metrics of a method and then check whether the method needs refactoring or not. Depending upon the computed cohesion metrics values of a given method and the input threshold values provided by the user, there can be three possible cases. Our proposed algorithm handles these three cases as explained below:

Case 1: Threshold $>$ computed cohesion metrics values

According to the main aim of refactoring, the cohesion metrics values for a method should be less than the standard threshold metrics values. When the given threshold values are greater than the method's cohesion metrics values, then the method needs refactoring. In our *Refactoring()* algorithm, it calls the *Binary_Refactoring()* algorithm that refactors the given method. So, this case can be handled correctly by our algorithm.

Case 2: Threshold $<$ computed cohesion metrics values

When the given threshold cohesion metrics values are less than the method's computed cohesion metrics values, the method remains unchanged. In the proposed *Refactoring()* algorithm, the same thing happens. In the algorithm, when it finds that the given threshold values are less than the method's cohesion metrics values, it does nothing and proceeds for next iteration.

Case 3: Threshold $==$ computed cohesion metrics values

Any method's cohesion metrics values should be greater than or equal to the threshold values provided by the user. So, when the given threshold values are equal to the method's coheson metrics values, then the method needs no refactoring. In the proposed *Refactoring()* algorithm, when this situation arises, the algorithm does nothing and the given method remains as it is.

Form the above cases, it can be deduced that our proposed refactoring algorithm works correctly in all the possible conditions and hence, the algorithm is correct. This proves the Theorem. □

7.3 Implementation and Results

In this section, we explain the implementation of our proposed algorithm. For implementation of our technique, we have developed a tool named *JSlicer* for the construction of SDG of Java programs and to compute the slices. The basic part of the tool is based on a Java SDG generation API, that is an open source API. This API generates SDG according to the representation proposed by [111]. Our tool analyses the byte-code of the class file and produces the SDG for the given program.

We have applied our proposed refactoring technique on some benchmark open-source Java projects to evaluate the effectiveness of our approach. For our study, we have taken 11 open source Java projects [1], whose size ranges from 100 to 1000 lines of code. In the study, the values of cohesion metrics for 651 methods are calculated. A total of 5000 lines of code are analysed. The details of the case study projects are given in Table 7.4. Table 7.4 shows the project name, Lines of Code (LOC), number of classes and methods present in a particular project, number of slicing criteria considered for refactoring, average size of the slices computed, and average slice computation time.

Table 7.4: Details of the case study projects

Sl. No.	**Project Name**	**LOC**	**No. of Classes**	**No. of Methods**	**No. of Slicing Criteria considered**	**Avg. Slice Size**	**Avg. Slice Computation Time (in ms)**
1	Alarm Clock	125	6	20	12	7.44	3.08
2	Binary Search Tree	130	4	23	6	12.37	5.25
3	Cruise Control	261	4	32	17	9.10	2.61
4	Groovy	361	2	34	14	5.91	2.79
5	Daisy	883	22	106	28	17.33	4.05
6	Deos	838	24	133	15	10.07	3.18
7	Double Linked List	277	1	32	6	15.14	2.85
8	Elevator-3	934	12	97	11	7.29	2.10
9	Lang	990	4	101	21	24.51	6.39
10	Vector	254	1	49	7	15.45	4.98
11	Red Black Tree-2	334	1	24	8	6.02	3.27

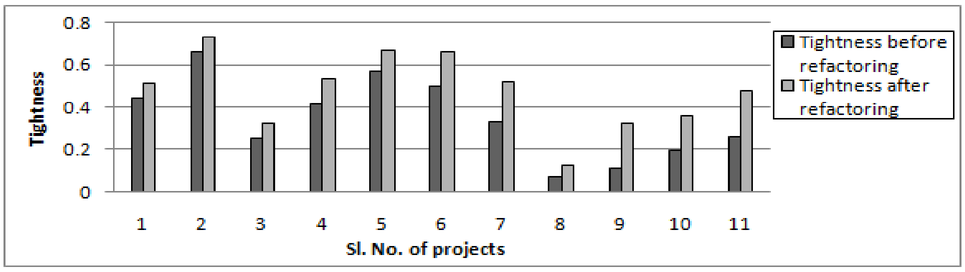
Results

We have calculated the values of cohesion metrics of all 651 methods. *It is very difficult to display all 651 method's cohesion metrics values, hence we have shown only the average*

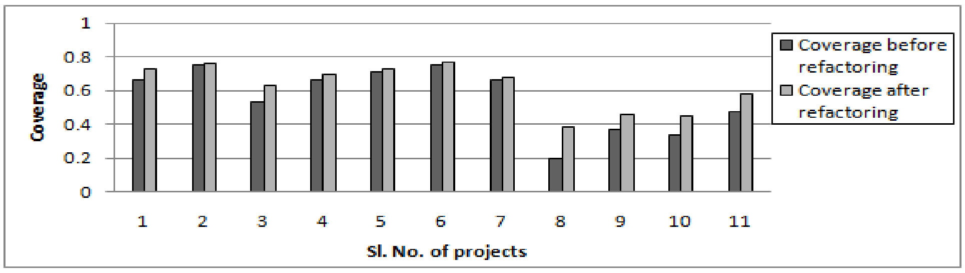
[1] http://sir.unl.edu/portal/index.php

Table 7.5: Details of change in cohesion metrics due to refactoring

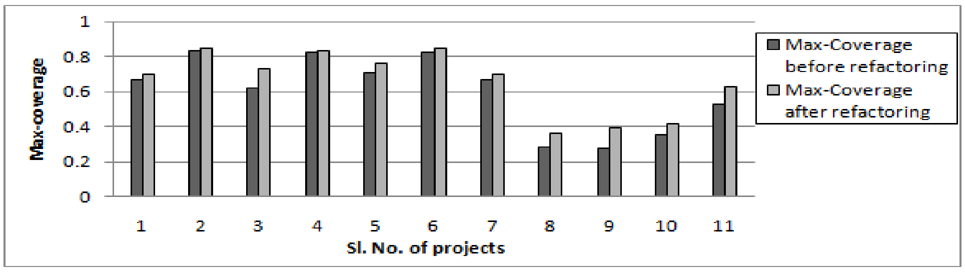
Sl. No.	Project Name	Tightness		Coverage		Max-coverage		Min-coverage		Overlap	
		before	after	before	after	before	after	before	after	before	after
1	Alarm Clock	0.44	0.51	0.66	0.73	0.66	0.69	0.55	0.61	0.61	0.69
2	BST	0.66	0.73	0.75	0.76	0.83	0.85	0.66	0.71	0.75	0.78
3	Cruise Control	0.25	0.32	0.53	0.63	0.625	0.73	0.44	0.53	0.531	0.68
4	Groovy	0.42	0.53	0.66	0.69	0.83	0.83	0.5	0.63	0.66	0.71
5	Daisy	0.57	0.66	0.71	0.73	0.71	0.76	0.71	0.76	0.71	0.76
6	Deos	0.5	0.66	0.75	0.77	0.83	0.85	0.66	0.7	0.74	0.78
7	Double Linked List	0.333	0.52	0.66	0.68	0.666	0.69	0.66	0.73	0.666	0.69
8	Elevator-3	0.07	0.12	0.2	0.38	0.28	0.36	0.1	0.38	0.21	0.46
9	Lang	0.11	0.32	0.37	0.46	0.27	0.39	0.44	0.52	0.37	0.67
10	Vector	0.19	0.35	0.34	0.44	0.35	0.42	0.32	0.47	0.34	0.58
11	Red Black Tree-2	0.26	0.47	0.47	0.58	0.52	0.63	0.42	0.57	0.47	0.684



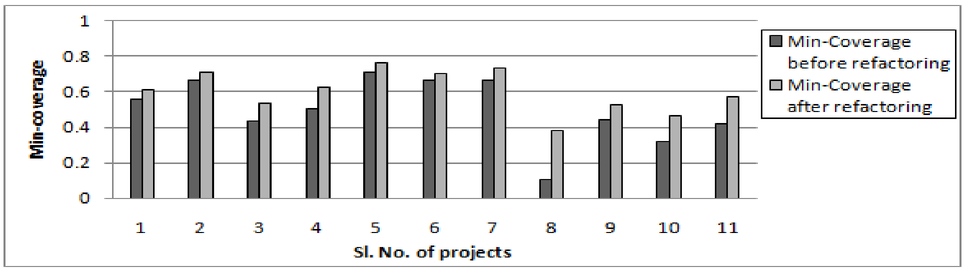
(a) Comparison of Tightness values



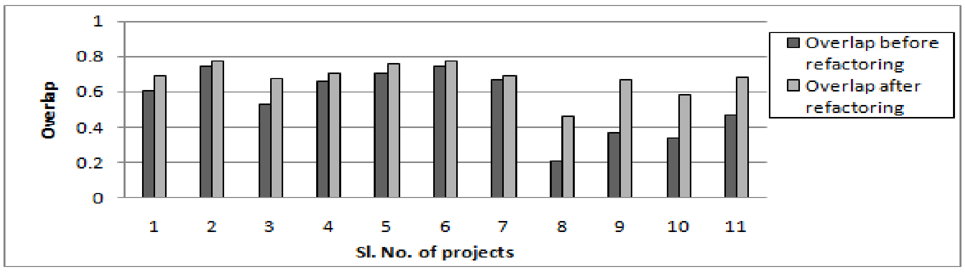
(b) Comparison of Coverage values



(c) Comparison of Max-Coverage values



(d) Comparison of Min-Coverage values



(e) Comparison of slice Overlap values

Figure 7.7: Effect of refactoring on Slice-based Metrics

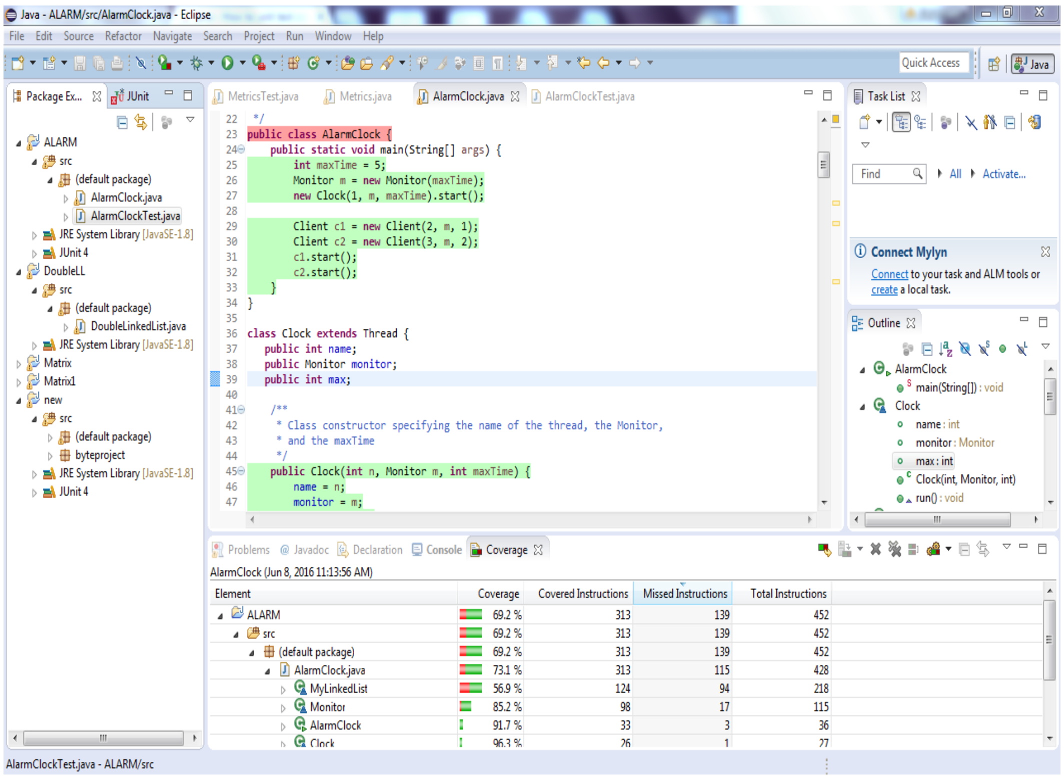
metrics value of the overall project, in Table 7.5. In our study, we considered only that methods for refactoring whose tightness and overlap values are less than the threshold values. On that selected methods, we have applied our proposed refactoring algorithm. After refactoring, we got the modified methods and we have again calculated their cohesion metrics. We found that the values of the cohesion metrics of all the methods have increased, as shown in Table 7.5. The effect of refactoring on tightness, coverage, max-coverage, min-coverage and overlap are also shown in Figure 7.7 (a)-(e). From Figure 7.7(a), it can be observed that there is 56.39% (approx.) increase in Tightness after refactoring takes place. Similarly, Figures 7.7(b)-7.7(e) show that there is 13.47% increase in Coverage, 19.48% increase in Max-Coverage, 0.6% increase in Min-Coverage, and 35.3% increase in Overlap after refactoring. From these plots, we clearly observe that, we achieve significant improvement in the values of cohesion metrics by applying our proposed refactoring approach.

The results from this study from the chapter's three main contributions:

- First, the result of our study indicates that refactoring of methods improves overall quality of the project.
- Second, slice-based cohesion metrics are very effective in quantifying the cohesiveness of the methods, for code refactoring.
- Finally, this study shows that cohesion of a method increases by refactoring it.

7.3.1 Refactoring Impact Analysis

We have conducted experiments with *EclEmma* plug-in for Eclipse to check the effect of our proposed refactoring technique on the behavior of the case study projects. *EclEmma* is an open source code coverage analysis tool that comes as an Eclipse plug-in. For any Java application executed in *Eclipse*, *EclEmma* collects coverage data and automatically calculates code coverage percentage as soon as the application terminates. *EclEmma* analyses each class and method of a project during its execution. We have used *EclEmma* plug-in for eclipse to show the effect of our proposed refactoring technique on execution of overall project. We have first designed some JUnit test suit for each case study project. Before applying the proposed refactoring technique, we have noted the code coverage of each case study project. Then, we applied our proposed refactoring technique on each of the project and again calculated the code coverage percentage of each project. The findings of our testing is provided in Table 7.6. Table 7.6 contains the name of projects, *EclEmma* code coverage percentage before and after refactoring, and finally the percentage of change in code coverage. We found that, there is negligible change in the code coverage percentage in each of the case study projects ranging from 0.02% to 3.43%. It shows that our refactoring technique only changes the structure of programs, and does not affect their original functionality. The increase in code coverage percentage is due to the fact that, during refactoring we are dividing one method into two methods and some statements are added as the header of newly created method. When the test cases are applied on the module after refactoring, then more number of program statements will be executed. Hence, there is an increase in code coverage percentage after refactoring of a software.



```
 */
public class AlarmClock {
    public static void main(String[] args) {
        int maxTime = 5;
        Monitor m = new Monitor(maxTime);
        new Clock(1, m, maxTime).start();

        Client c1 = new Client(2, m, 1);
        Client c2 = new Client(3, m, 2);
        c1.start();
        c2.start();
    }
}

class Clock extends Thread {
  public int name;
  public Monitor monitor;
  public int max;

   /**
    * Class constructor specifying the name of the thread, the Monitor,
    * and the maxTime
    */
    public Clock(int n, Monitor m, int maxTime) {
        name = n;
        monitor = m;
```

AlarmClock (Jun 8, 2016 11:13:56 AM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
ALARM	69.2 %	313	139	452
src	69.2 %	313	139	452
(default package)	69.2 %	313	139	452
AlarmClock.java	73.1 %	313	115	428
MyLinkedList	56.9 %	124	94	218
Monitor	85.2 %	98	17	115
AlarmClock	91.7 %	33	3	36
Clock	96.3 %	26	1	27

Figure 7.8: Screen shot of EMMA coverage analyzer

7.3.2 Threats to validity

In this section, we present some of the threats to the validity of our proposed approach.

1. We have considered method return and print statements as output variables. We have not considered all types of *Output variables* during the slicing criterion selection of our experiment. We believe that our refactoring approach will produce similar results even after altering the type of output variable.

2. Our proposed technique is developed by keeping in view Java and AspectJ programming languages. We have not tested our technique with other languages. However, we belive that our algorithm may work fine for C++ and AspectC++ after making suitable changes in the SDG construction, considering the relevant features of C++ and AspcetC++.

Table 7.6: Impact of refactoring on code coverage

Sl. No.	Project Name	Percentage of code coverage before refactoring	Percentage of code coverage after refactoring	Percentage of change in code coverage
1	Alarm Clock	69.2	71.5	3.32
2	Binary Search Tree	87.5	88.7	1.37
3	Cruise Control	66.6	66.9	0.45
4	Groovy	46.5	47.2	1.50
5	Daisy	40.7	42.1	3.43
6	Deos	24.3	24.8	2.05
7	Double Linked List	74.6	74.9	0.40
8	Elevator-3	21.4	22.0	2.80
9	Lang	30.1	30.8	2.32
10	Vector	67.8	69.0	1.76
11	Red Black Tree-2	76.7	76.9	0.02

7.4 Comparison with related work

Many work is done in the field of Software refactoring [31]. Wang et al. [112] have developed a tool called SEGMENT to insert blank line in program to increase readability of that program. But, the internal structure and complexity of the program is not changing. Bavota et al. [86] have proposed a refactoring approach for complex class. They have used semantic analysis to identify the relationship between method and class. But, they have not developed any fully automated tool, that can automatically perform refactoring based on some predefined formula. We have used the slice based cohesion metrics to quantify the formula for fully automate our proposed refactoring approach.

Monteiro et al. [88] have developed an approach to convert OOPs into AOPs. They have used 17 existing refactoring techniques to find the crosscutting concern from a given object-oriented program and then move the crosscutting concern to an *aspect*. But, they have not proposed any new type of refactoring technique. Sward et al. [89] have proposed that some software metrics such as cohesion, coupling, cyclomatic complexity etc. can be used in refactoring to improve the quality of software. They have considered some fixed example and perform refactoring to show that it reduces complexity. They have not implemented their proposed approach. We have developed a tool that perform refactoring of a given program.

7.5 Summary

In this chapter, we presented a technique for code refactoring using slice-based cohesion metrics and AOP. We clearly mentioned the cohesion metrics benchmark values, that should be satisfied for refactoring. We presented a detailed process for refactoring a given program. To explain our proposed technique, we considered an example Java program. In order to verify the working of our refactoring approach, we considered 11 open-source Java projects. We have observed that the increase in Tightness is 56.39% and increase in Coverage is 13.47% after refactoring.

Chapter 8
Conclusions

The primary aim of our work was to develop efficient dynamic slicing algorithms for aspect-oriented programs. In the following, we summarize the important contributions of our work. Finally, some suggestions for future work are given.

8.1 Contributions

In this section, we summarize the important contributions of our work. There are four important contributions, *Slicing of Aspect-Oriented Programs*, *Context-Sensitive Slicing of Concurrent Aspect-Oriented Programs*, *Parallel Context-Sensitive Dynamic Slicing of Distributed Aspect-Oriented Programs*, and *Software Refactoring using Program Slicing*.

8.1.1 Slicing of Aspect-Oriented Programs

We first developed an intermediate representation for representing simple Aspect-Oriented Programs . We have named this intermediate representation *Extended Aspect-Oriented System Dependence Graph* (EAOSDG). We statically construct EAOSDG only once before the program starts execution. Then, we present three dynamic slicing algorithms for aspect-oriented programs. The first one is the *Pointcut- Table* (PT) slicing algorithm, second one is Extended Two-Phase slicing algorithm and the third one is the *Context-Sensitive* (CS) slicing algorithm. We have developed a prototype tool for automatic generation and computation of slices for given AOP. We want to compare our three proposed slicing algorithms and therefore we have preform seven case studies. From the experiment we found that, CS slicing algorithm computes more precise slices in comparison to rest two algorithms. Also, CS slicing algorithm takes significant little computation time to generate slices in contrast with other two slicing algorithms.

8.1.2 Context-Sensitive Slicing of Concurrent Aspect-Oriented Programs

Due to the presence of inter-thread synchronization and communication, some control and data flows occurring in the threads of a Java program, are interdependent. To be able to capture this aspect, we have proposed an intermediate graph named as *Multithreaded*

Aspect-Oriented Dependence Graph (MAODG). Having introduced MAODG as an intermediate representation for concurrent aspect-oriented programs, we presented a dynamic slicing algorithm for concurrent aspect-oriented programs. We named this algorithm *Context-Sensitive Concurrent Aspect* (CSCA) slicing algorithm for concurrent aspect-oriented programs. The CSCA slicing algorithm uses MAODG as the intermediate representation and is based on preserving call-site using three stacks maintained for context-sensitivity. We have shown that CSCA slicing algorithm computes *correct* dynamic slices with respect to any slicing criterion. We have developed a slicing tool to verify the *correctness* and *preciseness* of our MBDS algorithm. Using five open source case study project, we have compared our proposed slicing algorithm with Context-Insensitive slicing algorithm and Ray et al. [14] slicing algorithm. The results obtained from this tool shows that our CSCA slicing algorithm performs much batter than rest two algorithms.

8.1.3 Parallel Context-Sensitive Dynamic Slicing of Distributed Aspect-Oriented Programs

In distributed aspect-oriented programs, each component program might communicate with another, by using *sockets*. Our MAODG representation can not model this type of communication. In order to represent this type of communication dependency, we have modified our MAODG. We introduce the notion of *R-node* to represent communication among component programs by using sockets. We call the modified intermediate representation *Distributed Aspect Dependence Graph* (DADG). We extended our CSCA slicing algorithm for dynamic slicing of *distributed* aspect-oriented programs. We named our algorithm *Parallel Context-Sensitive Dynamic Slicing* (PCDS) algorithm for distributed aspect-oriented programs. To achieve fast response time, our PCDS algorithm can run in a parallel manner, rather than running sequentially as in CSCA slicing algorithm. We have shown that PCDS algorithm computes *correct* dynamic slices with respect to any slicing criterion. We presented seven case studies of open source projects. We have implemented our approach, the PADS algorithm proposed by Ray et al. [13] and the CGCA proposed by Barpanda et al. [82]. The advantage of our algorithms is that when a request for a slice is made, it is running as parallel process. This results in substantial reduction in the response time for slice extraction.

8.1.4 Software Refactoring using Program Slicing

Software restructuring is essential for maintaining software quality. It is a usual practice that we first design the software and then go for coding. After coding, if there is any change in the requirement or if the output is incorrect, then we have to modify the code again. For each small code modification, it is not feasible to alter the design. These minor changes made to the code causes decay in the software design. Software refactoring is used to restructure the code to improve the design and quality of the software. We have proposed an approach for performing code refactoring. We have used slice-based cohesion metrics to identify the target methods that require refactoring. After identifying the target methods, we used program slicing to divide the target method into two parts. Finally, we have used the concept of *aspects* to alter the code structure in a manner that does not change the external behavior of the original module.

8.2 Future Work

In brief, we outline the following possible extensions to our work.

- Though we have developed our approach for AspectJ language, it can easily be extended for computing dynamic slices of any other aspect-oriented languages such as AspectWerkz, JMangler, Hyper/J, MixJuice, PROSE, ArchJava and JAC because these frameworks support AOP with Java.
- In our work, we have not considered the synchronized inter-process communication methods such as wait(), notify() and notifyAll() while computing dynamic slices of concurrent aspect-oriented programs. Our work can be extended to handle the synchronized methods by developing a suitable framework.
- In our work we have not considered composite data types such as arrays while computing dynamic slices of Concurrent aspect-oriented programs. Our work can be extended to handle composite data types by developing a suitable framework.
- In our work, we have showcased only one application of program slicing, i.e. Software Refactoring. There are several other applications of program slicing such as Test Case generation, Unit Testing, Reverse Engineering, automatic bug detection etc. There require more research works in such fields.

References

[1] M. Weiser, ‘‘Program slicing,’’ in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[2] X. Zhang, R. Gupta, and Y. Zhang, ‘‘Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams,’’ in *International Conference on Software Engineering*, 2004.

[3] B. Korel and J. Laski, ‘‘Dynamic program slicing,’’ *Information Processing Letters*, vol. 29, no. 3, pp. 155–163, 1988.

[4] R. J. Hall, ‘‘Automatic extraction of executable program subsets by simultaneous dynamic program slicing,’’ *Automated Software Engineering*, vol. 2, no. 1, pp. 33–53, 1995.

[5] D. W. Binkley and K. B. Gallagher, ‘‘Program slicing,’’ *Advances in Computers*, vol. 43, pp. 1–50, 1996.

[6] S. Horwitz, T. Reps, and D. Binkley, ‘‘Interprocedural slicing using dependence graphs,’’ *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–61, 1990.

[7] G. Canfora, A. Cimitile, and A. De Lucia, ‘‘Conditioned program slicing,’’ *Information and Software Technology*, vol. 40, no. 11, pp. 595–607, 1998.

[8] J. T. Lallchandani and R. Mall, ‘‘Computation of dynamic slices for object-oriented concurrent programs,’’ in *Software Engineering Conference, 2005. APSEC’05. 12th Asia-Pacific*. IEEE, 2005, pp. 8–pp.

[9] D. Liang and M. J. Harrold, ‘‘Slicing objects using system dependence graphs,’’ in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 358–367.

[10] L. Larsen and M. J. Harrold, ‘‘Slicing object-oriented software,’’ in *Software Engineering, 1996., Proceedings of the 18th International Conference on*. IEEE, 1996, pp. 495–505.

[11] J. Zhao, ‘‘Slicing aspect-oriented software,’’ in *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002, pp. 251–260.

[12] B. Korel and J. Laski, ‘‘Dynamic slicing of computer programs,’’ *Journal of Systems and Software*, vol. 13, no. 3, pp. 187–195, 1990.

[13] A. Ray, C. Kumar Niraj, S. Mishra, and D. P. Mohapatra, ‘‘Dynamic slicing of distributed aspect-oriented programs,’’ *Pensee Journal*, vol. 76, no. 5, pp. 69–86, 2014.

[14] A. Ray, S. Mishra, and D. P. Mohapatra, ‘‘An approach for computing dynamic slice of concurrent aspect-oriented programs,’’ *International Journal of Software Engineering and Its Applications*, vol. 7, no. 1, pp. 13–32, 2013.

[15] B. Korel and J. Rilling, ‘‘Dynamic program slicing methods,’’ *Information and Software Technology*, vol. 40, pp. 647–659, 1998.

[16] J. Krinke, ‘‘Context-sensitive slicing of concurrent programs,’’ *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 178–187, 2003.

[17] D. Binkley, ''Computing amorphous program slices using dependence graphs and a data flow model,'' in *Proceedings of the ACM Symposium on Applied Computing, ACM Press*, 1999.

[18] M. Harman and S. Danicic, ''Amorphous program slicing,'' in *Program Comprehension, 1997. IWPC'97. Proceedings., Fifth Iternational Workshop on*. IEEE, 1997, pp. 70–79.

[19] M. Harman, D. Binkley, and S. Danicic, ''Amorphous program slicing,'' *Journal of Systems and Software*, vol. 68, pp. 45–64, 2003.

[20] D. Binkley, N. Gold, M. Harman, J. Krinke, and S. Yoo, ''Observation-based slicing,'' *RN*, vol. 13, no. 13, p. 13, 2013.

[21] D. Jackson and E. J. Rollins, ''Chopping: A generalization of slicing,'' DTIC Document, Tech. Rep., 1994.

[22] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, ''A brief survey of program slicing,'' *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 2, pp. 1–36, 2005.

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

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

[25] G. A. Venkatesh, ''The semantic approach to program slicing,'' in *ACM SIGPLAN Notices*, vol. 26, no. 6. ACM, 1991, pp. 107–119.

[26] B. Hofer and F. Wotawa, ''Combining slicing and constraint solving for better debugging: The conbas approach,'' *Adv. Soft. Eng.*, vol. 2012, pp. 13:13–13:13, Jan. 2012.

[27] A. R. S.R.Raheman and S. Pradhan, ''Dynamic slicing of aspect oriented programs using aodg,'' *International Journal of Computer Science and Information Security*, vol. Vol-9(4), pp. 123–126, 2011.

[28] M. Sahu and D. P. Mohapatra, ''A node-marking technique for dynamic slicing of aspect-oriented programs,'' in *Information Technology,(ICIT 2007). 10th International Conference on*. IEEE, 2007, pp. 155–160.

[29] J. Zhao, ''Slicing aspect-oriented software,'' in *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. IEEE, 2002, pp. 251–260.

[30] K. M. Chandy and C. Kesselman, *Compositional C++: Compositional parallel programming*. Springer, 1993.

[31] T. Mens and T. Tourwé, ''A survey of software refactoring,'' *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.

[32] T. M. Meyers and D. Binkley, ''Slice-based cohesion metrics and software intervention,'' in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*. IEEE, 2004, pp. 256–265.

[33] R. Laddad, *AspectJ in action: practical aspect-oriented programming*. Dreamtech Press, 2003.

[34] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.

[35] R. Miles, *AspectJ cookbook*. " O'Reilly Media, Inc.", 2005.

[36] J. Singh and D. P. Mohapatra, ''A unique aspect-oriented program slicing technique,'' in *Advances in Computing, Communications and Informatics (ICACCI), 2013 International Conference on*. IEEE, 2013, pp. 159–164.

[37] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza *et al.*, ''On the impact of aspectual decompositions on design stability: An empirical study,'' in *ECOOP 2007–Object-Oriented Programming*. Springer, 2007, pp. 176–200.

[38] M. Lippert and C. V. Lopes, ‘‘A study on exception detection and handling using aspect-oriented programming,’’ in *Proceedings of the 22nd international conference on Software engineering*. ACM, 2000, pp. 418–427.

[39] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira, ‘‘Exceptions and aspects: the devil is in the details,’’ in *FSE*, vol. 14, 2006, pp. 152–156.

[40] J. Viega and J. Vuas, ‘‘Can aspect-oriented programming lead to more reliable software?’’ *Software, IEEE*, vol. 17, no. 6, pp. 19–21, 2000.

[41] H. Yin, C. Bockisch, and M. Aksit, ‘‘A fine-grained debugger for aspect-oriented programming,’’ in *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 2012, pp. 59–70.

[42] M. Weiser, ‘‘Program slicing,’’ in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[43] F. Tip, ‘‘A survey of program slicing techniques,’’ *Journal of programming languages*, vol. 3, no. 3, pp. 121–189, 1995.

[44] K. J. Ottenstein and L. M. Ottenstein, ‘‘The program dependence graph in a software development environment,’’ in *ACM Sigplan Notices*, vol. 19, no. 5. ACM, 1984, pp. 177–184.

[45] J. R. Lyle and M. D. Weiser, ‘‘Automatic program bug location by program slicing,’’ in *Proceedings of the second International Conference on Computers and Applications, Peking, China*, 1987, pp. 877–882.

[46] T. Reps and G. Rosay, ‘‘Precise interprocedural chopping,’’ in *Proceedings of Third ACM Symposium on the Foundations of Software Engineering*, October 1995, pp. 41–52.

[47] K. Gallagher and J. Lyle, ‘‘Using program slicing in software maintenance,’’ *IEEE Transactions on Software Engineering*, vol. SE-17, no. 8, pp. 751–761, 1991.

[48] D. Binkley, ‘‘The application of program slicing to regression testing,’’ *Information and Software Technology, Special Issue on Program Slicing*, vol. 40, no. 11-12, pp. 583–594, 1998.

[49] L. Xu, B. Xu, Z. Chen, J. Jiang, H. Chen, and H. Yang, ‘‘Regression testing for web applications based on slicing,’’ in *Proceedings of 28th IEEE Annual International Computer Software and Applications Conference, IEEE CS Press*, 2003, pp. 652–656.

[50] D. Binkley, ‘‘Using semantic differencing to reduce the cost of regression testing,’’ in *Proceedings of the IEEE Conference on Software Maintenance*, November 1992, pp. 41–50.

[51] R. Gupta, M. J. Harrold, and M. L. Soffa, ‘‘Program slicing-based regression testing techniques,’’ *Journal of Software Testing, Verification and Reliability*, vol. 6, 1996.

[52] J. Beck and D. Eichmann, ‘‘Program and interface slicing for reverse engineering,’’ in *Proceedings of the 15th international conference on Software Engineering*. IEEE Computer Society Press, 1993, pp. 509–518.

[53] D. W. Binkley and K. B. Gallagher, ‘‘Program slicing,’’ *Advances in Computers*, vol. 43, pp. 1–50, 1996.

[54] A. De Lucia, ‘‘Program slicing: Methods and applications,’’ in *scam*. IEEE, 2001, p. 0144.

[55] J. Silva, ‘‘A vocabulary of program slicing-based techniques,’’ *ACM Computing Surveys (CSUR)*, vol. 44, no. 3, p. 12, 2012.

[56] R. Mall, *Fundamentals of Software Engineering*. Prentice Hall, India, 2nd Edition, 2003.

[57] M. Harman and S. Danicic, ‘‘Using program slicing to simplify testing,’’ *Journal of Software Testing, Verification and Reliability*, vol. 5, 1995.

[58] I. Forgacs and A. Bertolino, ‘‘Feasible test path selection by principal slicing,’’ in *Proceedings of 6th Euoropean Software Engineering Conference*, September 1997.

[59] S. Horwitz, J. Prins, and T. Reps, ‘‘Integrating noninterfering versions of programs,’’ *ACM Transactions on Programming Languages and Systems*, vol. 11, pp. 345–387, July 1989.

[60] D. Binkley, S. Horwitz, and T. Reps, ‘‘Program integration for languages with procedure calls,’’ *ACM Transactions on Software Engineering and Methodology*, vol. 4, no. 1, pp. 3–31, 1995.

[61] Z. Chen, Y. Zhou, B. Xu, J. Zhao, and H. Yang, ‘‘A novel approach for measuring class cohesion based on dependence analysis,’’ in *Proceedings of International Conference on Software Maintenance, IEEE Press*, 2002, pp. 377–384.

[62] J. M. Bieman and L. M. Ott, ‘‘Measuring functional cohesion,’’ *IEEE Transactions on Software Engineering*, vol. 20, no. 8, pp. 644–657, August 1994.

[63] M. Weiser, ‘‘Reconstructing sequential behavior from parallel behavior projections,’’ *Information Processing Letters*, vol. 17, no. 10, pp. 129–135, 1983.

[64] M. Fowler, *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[65] F. Castor, N. Cacho, E. Figueiredo, A. Garcia, C. M. Rubira, J. S. de Amorim, and H. O. da Silva, ‘‘On the modularization and reuse of exception handling with aspects,’’ *Software: Practice and Experience*, vol. 39, no. 17, pp. 1377–1417, 2009.

[66] M. P. Monteiro and J. M. Fernandes, ‘‘An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms,’’ *Software: Practice and Experience*, vol. 38, no. 4, pp. 361–396, 2008.

[67] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant’Anna, S. Soares, P. Borba, U. Kulesza *et al.*, ‘‘On the impact of aspectual decompositions on design stability: An empirical study,’’ in *ECOOP 2007–Object-Oriented Programming*. Springer, 2007, pp. 176–200.

[68] D. P. Mohapatra, R. Mall, and R. Kumar, ‘‘An overview of slicing techniques for object-oriented programs,’’ *Informatica*, vol. 30, no. 2, 2006.

[69] J. Zhao, ‘‘A dependence-based representation for concurrent object-oriented software maintenance,’’ in *Software Maintenance and Reengineering, 1998. Proceedings of the Second Euromicro Conference on*. IEEE, 1998, pp. 60–66.

[70] T. ter Braak, ‘‘Extending program slicing in aspect-oriented programming with intertype declarations,’’ in *5th Twente Student Conference on IT*, 2006.

[71] D. P. Mohapatra, M. Sahu, R. Kumar, and R. Mall, ‘‘Dynamic slicing of aspect-oriented programs,’’ *Informatica*, vol. 32, no. 3, 2008.

[72] J. Zhao, ‘‘Slicing concurrent java programs,’’ in *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*. IEEE, 1999, pp. 126–133.

[73] Z. Guangquan and R. Mei, ‘‘An approach of concurrent object-oriented program slicing based on ltl property,’’ in *Computer Science and Software Engineering, 2008 International Conference on*, vol. 2. IEEE, 2008, pp. 650–653.

[74] M. G. Nanda and S. Ramesh, ‘‘Slicing concurrent programs,’’ *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 5, pp. 180–190, 2000.

[75] D. Goswami and R. Mall, ‘‘Dynamic slicing of concurrent programs,’’ in *High Performance Computing—HiPC 2000*. Springer, 2000, pp. 15–26.

[76] J. Cheng, ‘‘Slicing concurrent programs,’’ in *Automated and Algorithmic Debugging*. Springer, 1993, pp. 223–240.

[77] Z. Chen and B. Xu, ‘‘Slicing concurrent java programs,’’ *ACM Sigplan Notices*, vol. 36, no. 4, pp. 41–47, 2001.

[78] E. Duesterwald, R. Gupta, and M. Soffa, ‘‘Distributed slicing and partial re-execution for distributed programs,’’ in *Languages and Compilers for Parallel Computing*. Springer, 1993, pp. 497–511.

[79] M. Kamkar and P. Krajina, ‘‘Dynamic slicing of distributed programs,’’ in *Software Maintenance, 1995. Proceedings., International Conference on*. IEEE, 1995, pp. 222–229.

[80] H. F. Li, J. Rilling, and D. Goswami, ‘‘Granularity-driven dynamic predicate slicing algorithms for message passing systems,’’ *Automated Software Engineering*, vol. 11, no. 1, pp. 63–89, 2004.

[81] D. P. Mohapatra, R. Mall, and R. Kumar, ‘‘A novel approach for dynamic slicing of distributed object-oriented programs,’’ in *Distributed Computing and Internet Technology*. Springer, 2005, pp. 304–309.

[82] S. S. Barpanda and D. P. Mohapatra, ‘‘Dynamic slicing of distributed object-oriented programs,’’ *IET software*, vol. 5, no. 5, pp. 425–433, 2011.

[83] C. L. Liu, *Elements of Discrete Mathematics: A Computer Oriented Approach*. Tata McGraw-Hill Education, 2013.

[84] H. Agrawal and J. R. Horgan, ‘‘Dynamic program slicing,’’ in *ACM SIGPLAN Notices*, vol. 25, no. 6. ACM, 1990, pp. 246–256.

[85] X. Wang, L. Pollock, and K. Vijay-Shanker, ‘‘Automatic segmentation of method code into meaningful blocks: Design and evaluation,’’ *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 27–49, 2014.

[86] G. Bavota, A. De Lucia, and R. Oliveto, ‘‘Identifying extract class refactoring opportunities using structural and semantic cohesion measures,’’ *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, 2011.

[87] S. Mohsin and Z. Kaleem, ‘‘Program slicing based software metrics towards code restructuring,’’ in *Computer Research and Development, 2010 Second International Conference on*. IEEE, 2010, pp. 738–741.

[88] M. P. Monteiro and J. M. Fernandes, ‘‘Towards a catalog of aspect-oriented refactorings,’’ in *Proceedings of the 4th international conference on Aspect-oriented software development*. ACM, 2005, pp. 111–122.

[89] R. E. Sward, A. Chamillard, and D. A. Cook, ‘‘Using software metrics and program slicing for refactoring,’’ DTIC Document, Tech. Rep., 2004.

[90] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon, ‘‘Automated scheduling for clone-based refactoring using a competent ga,’’ *Software: Practice and Experience*, vol. 41, no. 5, pp. 521–550, 2011.

[91] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-oriented programming*. Springer, 1997.

[92] T. ter Braak, ‘‘Extending program slicing in aspectoriented programming with intertype declarations,’’ in *5th Twente Student Conference on IT*, 2006.

[93] J. Gait, ‘‘A probe effect in concurrent programs,’’ *Software: Practice and Experience*, vol. 16, no. 3, pp. 225–233, 1986.

[94] D. Giffhorn and C. Hammer, ‘‘Precise slicing of concurrent programs,’’ *Automated Software Engineering*, vol. 16, no. 2, pp. 197–234, 2009.

[95] J. Krinke, ‘‘Evaluating context-sensitive slicing and chopping,’’ in *Software Maintenance, 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 22–31.

[96] J. H. Andrews, ‘‘Process-algebraic foundations of aspect-oriented programming,’’ in *International Conference on Metalevel Architectures and Reflection*. Springer, 2001, pp. 187–209.

[97] C. Allan, P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. De Moor, D. Sereni, G. Sittampalam, and J. Tibble, ‘‘Adding trace matching with free variables to aspectj,’’ in *ACM SIGPLAN Notices*, vol. 40, no. 10. ACM, 2005, pp. 345–364.

[98] R. Douence, D. Le Botlan, J. Noyé, and M. Südholt, ‘‘Concurrent aspects,’’ in *Proceedings of the 5th international conference on Generative programming and component engineering*. ACM, 2006, pp. 79–88.

[99] J. R. Larus, ‘‘Abstract execution: A technique for efficiently tracing programs,’’ *Software: Practice and Experience*, vol. 20, no. 12, pp. 1241–1258, 1990.

[100] P. Green, P. C. Lane, A. Rainer, and S. Scholz, ‘‘An introduction to slice-based cohesion and coupling metrics,’’ 2009.

[101] R. Douence, P. Fradet, and M. Südholt, ‘‘Composition, reuse and interaction analysis of stateful aspects,’’ in *Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM, 2004, pp. 141–150.

[102] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004, vol. 19.

[103] W. Fokkink, *Distributed Algorithms: An Intuitive Approach*. MIT Press, 2013.

[104] D. Balzarotti, A. Castaldo, L. C. D'Ursi, and M. Monga, ‘‘Slicing aspectj woven code,’’ in *FOAL 2005 Proceedings*, 2005, p. 27.

[105] D. Binkley, S. Danicic, T. Gyimóthy, M. Harman, Á. Kiss, and B. Korel, ‘‘A formalisation of the relationship between forms of program slicing,’’ *Science of Computer Programming*, vol. 62, no. 3, pp. 228–252, 2006.

[106] D. P. Mohapatra, R. Kumar, R. Mall, D. Kumar, and M. Bhasin, ‘‘Distributed dynamic slicing of java programs,’’ *Journal of Systems and Software*, vol. 79, no. 12, pp. 1661–1678, 2006.

[107] R. Ettinger and M. Verbaere, ‘‘Untangling: a slice extraction refactoring,’’ in *Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM, 2004, pp. 93–101.

[108] P. Sudhaman, ‘‘Evaluating software quality using software metrics for a software product,’’ *International Journal of Information Systems and Change Management*, vol. 5, no. 2, pp. 180–190, 2011.

[109] P. Green, P. C. Lane, A. Rainer, and S. Scholz, ‘‘An introduction to slice-based cohesion and coupling metrics,’’ 2009.

[110] L. M. Ott and J. J. Thuss, ‘‘Slice based metrics for estimating cohesion,’’ in *Software Metrics Symposium, 1993. Proceedings., First International*. IEEE, 1993, pp. 71–81.

[111] N. Walkinshaw, M. Roper, and M. Wood, ‘‘The java system dependence graph,’’ in *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*. IEEE, 2003, pp. 55–64.

[112] X. Wang, L. Pollock, and K. Vijay-Shanker, ‘‘Automatic segmentation of method code into meaningful blocks to improve readability,’’ in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 35–44.

Dissemination

Internationally indexed journals (*Web of Science, SCI, Scopus, etc.*)[1]

1. Jagannath Singh, Subhrakanta Panda, Pabitra Mohan Khilar, and Durga Prasad Mohapatra, "Context-Sensitive Dynamic Slicing of Distributed AOP", ACM Sigsoft Software Engineering Journal, Vol. 41, No. 2, pp 1-8, March 2016.

2. Jagannath Singh, Pabitra Mohan Khilar, and Durga Prasad Mohapatra, "Refactoring of Software using Program Slicing and AOP", International Journal of Business and Information Systems, Inderscience.

Conferences [1]

1. Jagannath Singh, Dishant Munjal, and Durga Prasad Mohapatra, ''Context-Sensitive Dynamice Slicing of Concurrent AOP", 21st Asia-Pacific Software Engineering Conference (APSEC-2014), IEEE, Jeju, South Korea, 1-4 December 2014.

2. Dishant Munjal, Jagannath Singh, Subhrakanta Panda and Durga Prasad Mohapatra, "Automated Slicing of Aspect-Oriented Programs using Bytecode Analysis", In 39th International Computers, Software and Applications Conference (COMPSAC-2015), IEEE, Taichung, Taiwan, 1-5 July 2015.

3. Jagannath Singh, Pabitra Mohan Khilar, and Durga Prasad Mohapatra, ''Component-Aspect Separation Based Slicing of Aspect-Oriented Programs", In International Conference on Advanced Computing, Networking and Informatics (ICACNI-2013), Springer, Raipur, 12-14 June, 2013.

4. Jagannath Singh and Durga Prasad Mohapatra, ''A Unique Aspect-Oriented Program Slicing Technique", In 2nd International Conference on Advances in Computing, Communications and Informatics (ICACCI-2013), IEEE, Mysore, 22-25 August, 2013.

[1]Articles already published, in press, or formally accepted for publication.

Article under preparation [2]

1. Jagannath Singh, Pabitra Mohan Khilar, and Durga Prasad Mohapatra, ‘‘Dynamic Slicing of Distributed Aspect-Oriented Programs: A Context-sensitive Approach”, Computer Standards and Interfaces, Elsevier (Major Revision).

2. Jagannath Singh, Pabitra Mohan Khilar, and Durga Prasad Mohapatra, “Exclusive Context-Sensitive Dynamic Slicing of Concurrent Aspect-Oriented Programs”, Software: Practice and Experience, Wiley Publications (Under Review).

3. Jagannath Singh and Durga Prasad Mohapatra, “Source Code Analysis Based Context-Sensitive Dynamic Slicing of Web Applications”, CSI Transaction ICT, Springer (Under Review).

[2]Articles under review, communicated, or to be communicated.