Analysis of Slice-Based Metrics for Aspect-Oriented Programs

B.Tech Project Thesis

by

Dishant Munjal
111CS0609

under the guidance of

Prof. D.P. Mohapatra
Department of Computer Science & Engineering

National Institute of Technology, Rourkela
Rourkela-769008, Odisha, INDIA

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

May 11, 2015

Certificate

This is to certify that the work in the thesis entitled *Analysis of Slice-Based Metrics for Aspect-Oriented Programs* by Dishant Munjal is a record of an original research work carried out under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Computer Science and Engineering. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Durga Prasad Mohapatra
Associate Professor
Dept. of Computer Science and Engineering
NIT Rourkela

Acknowledgement

First of all, I would like to express my deep sense of respect and gratitude towards my supervisor, Prof. Durga Prasad Mohapatra, who has been the guiding force behind this work. I want to thank him for introducing me to the field of Program Slicing and giving me the opportunity to work under him. His undivided faith in this topic and ability to bring out the best of analytical and practical skills in people has been invaluable in tough periods. Without his invaluable advice and assistance it would not have been possible for me to complete this thesis. I am greatly indebted to him for his constant encouragement and invaluable advice in every aspect of my academic life. I consider it my good fortune to have got an opportunity to work with such a wonderful person.

I would also like to thank all faculty members, PhD scholars and all colleagues to provide me their regular suggestions and encouragements during the whole work. I would like to specially thank Mr. Jagannath Singh and Mr. Subhrakanta Panda for their constant words of enlightenment and wisdom, and their ideas and help whenever required. I also want to thank Mr. Shaswat Rungta for his consistent encouragement and understanding. His help can never be penned with words.

At last but not the least I am in debt to my family to support me regularly during my hard times.

I wish to thank all faculty members and secretarial staff of the CSE Department for their sympathetic cooperation.

Dishant Munjal

Abstract

To improve separation of concerns in software design and implementation, the technique of Aspect-Oriented Programming (AOP) was introduced. But AOP has a lot of features like aspects, advices, point-cuts, join-points etc., and because of these the usage of the existing intermediate graph representations is rendered useless. In our work we have defined a new intermediate graph representation for AOP. The construction of SDG is automated by analysing the bytecode of aspect-oriented programs that incorporates the representation of aspect-oriented features. After constructing the SDG, we propose a slicing algorithm that uses the intermediate graph and computes slices for a given AOP. Program slicing has numerous applications in software engineering activities like debugging, testing, maintenance, model checking etc. To implement our proposed slicing technique, we have developed a prototype tool that takes an AOP as input and compute its slices using our proposed slicing algorithm. To evaluate our proposed technique, we have considered some case studies by taking open source projects. The comparative study of our proposed slicing algorithm with some existing algorithms show that our approach is an efficient and scalable approach of slicing for different applications with respect to aspect-oriented programs. Software metrics are used to measure certain aspects of software. Using the slicing approach we have computed eight software metrics which quantitatively and qualitatively analyse the whole aspect project. We have compiled a metrics suite for AOP and an automated prototype tool is developed for helping the process of SDLC.

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Objective 2

1.3 Organization of the Thesis 2

2 Basic Concepts **4**

2.1 Slicing 4

2.1.1 Forward Slicing 4

2.1.2 Backward Slicing 4

2.1.3 Static Slicing 5

2.1.4 Dynamic Slicing 5

2.2 Program Dependence Graph 5

2.3 System Dependence Graph 5

2.4 AOP 6

2.4.1 Features of AOP 6

3 Literature Review **7**

3.1 Slicing 7

3.1.1 Object-Oriented Slicing 7

3.1.2 Aspect-Oriented Slicing 7

3.2 AOP Software Metrics 8

4 Aspect-Oriented Intermediate Graph Representation **10**

4.1 Dependencies 10

4.2 EAOSDG Generation 11

4.3 Implementation 12

4.4 Summary 16

5 Slicing of Aspect-Oriented Programs **17**

5.1 Computation of Slices 17

5.2 Complexity Analysis 19

5.3 Working of Algorithm 19

5.4 Implementation 20

5.5 Comparison with other work 21

5.6 Summary 22

6 Software Metrics **23**
6.1 C&K Metrics 23
6.2 Aspect-Oriented Metrics 25
6.3 Implementation 27
6.3.1 Lines of code 27
6.3.2 Weighted Operations in Module 27
6.3.3 Depth of Inheritence Tree 27
6.3.4 Number of Children 28
6.3.5 Avg. Aspect Complexity 28
6.3.6 Crosscutting Degree of an Aspect 28
6.3.7 Coupling on Advice Execution 28
6.3.8 Aspect Cohesion 29
6.4 Summary 29

7 Conclusion **31**

Chapter 1

Introduction

Weiser introduced program slicing as a decomposition technique that extracts program elements related to a particular computation, from a program [1]. Basically, slicing is a process of simplifying programs by targeting selected aspects of semantics. It contains all those parts of a program that may affect the values computed at some program point of interest (also called slicing criterion). The different types of slicing include static slicing, dynamic slicing, forward slicing, and backward slicing [2]. There exist many algorithms that are introduced by various researchers for procedural as well as object-oriented programs [3, 4], but few work is reported on slicing of aspect-oriented programs.

In the present scenario the dominant programming paradigm in the industries is object-oriented programming (OOP). It is based on an idea that one creates a software system by decomposing the problem statement into *objects* and coding for those objects. These objects abstract the behaviour and data of the whole project together into a single conceptual entity. Most of the current software development methodologies and tools reflect the presence of object-oriented paradigm. It is a brilliant idea with certain limitations. OOP has difficulty localizing concerns which involve global constraints and pandemic behaviours, applying domain-specific knowledge, and appropriately separating concerns. Aspect-Oriented Programming (AOP) [5] is a new language paradigm proposed for cleanly modularizing the scattered and tangled code known as cross-cutting concerns (like synchronization, exception handling, and resource sharing). AOP is based on the impression that computer systems are programmed in a better way if the several concerns (properties or areas of interest) of a system and some descriptions of the relationships between them are separately specified. If we are able to do this properly we then rely on mechanisms in the underlying AOP environment to *weave* or compose those concerns together into a coherent program. Concerns are flexible as they range from low level concepts, like caching and buffering, to high level concepts, such as security and quality of service. They can be non-functional (systemic), like synchronization and transaction management, or functional, such as features or business rules. The presence of such cross-cutting concerns in standard language constructs (such as Java) usually results in poorly structured code. AOP controls the scattering and tangling of such code that in turns improve the structure of the program, thus making it easier to develop and maintain the project.

Software measurement is an essential component of good software engineering [6]. Software metrics have been studied and used as a quantitative means of assessing the process of software development as well as the quality of software products [7]. The effective use of software metrics is dependent on the statistical validation of the metrics.one of the most costly and difficult process of SDLC is considered as software maintenance. The software metrics when used can help us in evaluation of several quality characteristics of AOP like modularity, reusability, size etc. For example size metrics can help in the identification of modularisation problems i.e. bigger modules can be broken into smaller ones which have fewer tasks or have their features merged into other modules, whichever is deemed necessary.

1.1 Motivation

With the introduction of features like advices, point-cuts, code introductions etc., the existing intermediate representations used for representing procedural and object-oriented programs become obsolete. This creates the need to create a new intermediate representation of AOP for better program comprehension of AOPs.

With the new representation, the existing slicing algorithms also tend to be useless as the slicing algorithms are usually based on the intermediate representations. So with the introduction of the new intermediate representations, we also require a new slicing algorithm which corresponds to that representation.

Several studies are done for procedural and object-oriented paradigm, but less work is done Aspect-Oriented paradigm for the same. Therefore, there is a need to work on compiling a metrics suite that can give not only quantitative analysis but also the qualitative measurement of the AOP.

1.2 Objective

Depending on the motivations, the objectives of my research is set.

- To develop an intermediate representation of the dependencies in AOP.
- To develop a tool for generation of dependence graph of AOP.
- To develop and implement a slicing algorithm for computation of slices.
- To define different metrics associated with AOP and implement those.

By the end of our project, we aim to create a prototype tool that will take input the bytecode of the project and create the intermediate graph and then will slice or calculate the metrics based on the requirements of the user.

1.3 Organization of the Thesis

The rest of the thesis is organized as follows. We briefly introduce the final year project in the first chapter of the thesis. In Chapter 2, all the basic concepts used in the whole project are defined. We present the literature review where we have described some existing works on Slicing and Software Metrics in Chapter 3. In the Chapter 4 we explain the new intermediate representation for AOP. We give a pseudocode for the representation as well as some case studies are taken into account for showing the implementation of the generator of the graph.

In the Chapter 5 we explain the slicing algorithm proposed by us which is associated with the intermediate graph representation proposed by us. This chapter also includes the working of the algorithm and the comparison of our work with a few existing works. In Chapter 6, we explain the software metrics for AOP and show the depicted results. We have defined and explained the whole metrics suite for AOP compiled by us in this chapter. At the end we conclude our work in Chapter 7 and show the dissemination of the work.

Chapter 2

Basic Concepts

In this chapter, we define and discuss some of the general concepts required for a clear understanding of the proposed approach.

2.1 Slicing

Program Slicing [8, 1] is a program analysis technique which reduces the program to those statements that are relevant for a particular context. It checks the dependency relation between the program statements to identify those programs parts that affect or are affected by a point of interest, called the *slicing criterion.* The approach of slicing as reported in the existing literature [4, 9] is based on an intermediate graphical representation of the input program. Program slicing can be used and have been suggested in many applications. It is helpful in software maintenance, program debugging, software measurement, testing, program parallelization, program comprehension, and many more [10]. For example, the dynamic slicing of programs has played an important role in debugging large programs [11]. Debugging has always been and still is a costly part of SDLC and program slicing helps in breaking the large programs into program statements applicable to particular computation.

There are different forms of slicing techniques. The basic overview of those are given below:

2.1.1 Forward Slicing

It computes all those parts that might be affected by the slicing criterion, using their dependence on the slicing criterion. Through the intermediate representation, this slicing technique traverses in the forward direction thus computing only those nodes that might get affected by the execution of the slicing criterion. This technique doesnt give an executable set of nodes or an executable sub-program.

2.1.2 Backward Slicing

The backward slices are those that are computed by gathering the statements and control predicates by way of a backward traversal of the programs control flow graph (CFG) or Program Dependence Graph (PDG), starting at the slicing criterion [2, 9]. Basically, it consists of all the program statements that might have affected the slicing criterion at any point of execution directly or indirectly.

2.1.3 Static Slicing

Static program slicing is a well-established method for analysing sequential programs, which can be used for program understanding, debugging, and testing. Computation of slices is done by checking for the consecutive sets of transitively relevant statements based on data and control dependencies [8]. As the information used for computation of slices is available statically, this type of slice is termed static slice [2]. It consists of the program statements affecting the value of a variable at any program point of interest, referred as slicing criterion. Basically, it is used to identify the program statements which potentially contribute to the computation of the slicing criterion for any possible programs inputs.

2.1.4 Dynamic Slicing

In dynamic program slicing, the dependencies occurring in a particular program execution are taken into account. A dynamic slicing criterion takes into account the input while distinguishing between different occurrences of any statement in the execution history; usually, it comprises of triple (input, occurrence of a statement, variable). Basically, it consists of only those statements that actually affect the value of a variable at a program point of view during the given execution trace. Therefore dynamic slices are typically smaller than static slices. These are found to be quite useful in software maintenance, program debugging, testing etc. [12]. In simple words, the difference between dynamic and static slicing is that the former assumes fixed input for a program, whereas latter does not make any assumptions concerning the input [2].

2.2 Program Dependence Graph

A Program Dependence Graph (PDG) is a directed graph and its nodes represent lines of code of the source program. Its edges denote dependence relations (data dependence or control dependence) between statements. An edge drawn from node N_s to node N_d represents node N_d depends on node N_s. PDG also includes special nodes which represent method call and parameter passing [13].

The control and data dependence edges can be defined as:

Control Dependence: There is a control dependency between N_s and N_d if N_s is a conditional predicate and execution of N_d is determined by the result of N_s.

Data Dependence: There is a data dependency between N_s and N_d if, for any variable v, N_s assigns the value to v, N_d refers to v and there exists atleast one execution path in between N_s and N_d without the value of v being changed.

2.3 System Dependence Graph

A collection of PDGs (one for each procedure) is called a system dependence graph (SDG) [4]. A PDG represents a procedure as a graph in which vertices are statements or predicate expressions. The flow of data between statements or expressions is represented by data dependence edge, while the control dependence edges represent control conditions on which the execution of a statement or expression depends. Each and every PDG consists of an entry vertex that represents entry into the procedure. Every procedure entry vertex is associated with formal-in and formal-out vertices in the SDG to model parameter passing. This is done by having a formal-in vertex for every formal parameter of the procedure and a formal-out vertex for each formal parameter that may be modified by the procedure. An SDG associates each call site in a procedure with a call vertex and a set of actual-in and actual-out vertices. An SDG contains an actual-in vertex for each actual parameter at the call site and an actual-out vertex for each actual parameter that may be modified by the called procedure.

2.4 AOP

The technique of improvement of separation of concerns in software design and implementation [14] is known as Aspect-Oriented Programming (AOP). It works by providing explicit procedures for capturing the structure of cross-cutting concerns. Like all programming techniques, AOP addresses both what the programmer could say and how the computer system would realize a particular program in a working system. Therefore, a goal of AOP systems comprises of providing a way of expressing crosscutting concerns in computational systems and ensuring the conceptual straightforwardness and efficient implementations of those mechanisms [15].
Cross-cutting concerns : The parts of the program, scattered across multiple modules of the program as well as tangled with other modules are termed cross-cutting concerns [16]. The most simple and common example of a crosscutting concern is logging, as it affects many modules or classes across the software and it intrudes on business logic.

2.4.1 Features of AOP

Some of the features of AOP [16] are explained below:

- *Aspects*: These correspond to the classes in OOP that also contain functionalities. But, unlike classes in OOP, these are meant to compute crosscutting concerns to be injected into other parts of the code.

- *Join-points*: Aspects cross-cut objects at only well-defined points, such as at object construction, method calls or member variable access points. Such well-defined points are termed as join-points.

- *Point-cut*: The specification for naming join-points is called a point-cut. It is a collection of join-points.

- *Advice*: Once the join-points are spotted in a program, its intended behavior must be defined. This behavior is called advice. An advice has the same level of accessibility as that of an arbitrary Java method.

- *Code Introduction*: The ability of AOP through which programmers add variables and methods into a program entity by using the defined aspects is called code introduction.

Chapter 3

Literature Review

The literature review of this project is done in two different parts. The first part is for the literature on slicing of AOPs, while the second part focuses on the software metrics.

3.1 Slicing

3.1.1 Object-Oriented Slicing

In order to represent a sequential procedural program with multiple procedures, Horwitz et al. [9] extended the program dependence graph to introduce the system dependence graph named *Horwitz-Reps-Binkley SDG.*

Larsen and Harrold [4] extended Horwitz-Reps-Binkley SDG so as to represent Object-Oriented Programs. Their SDGs could represent various object-oriented features. As the SDG defined was an extension of Horwitz-Reps-Binkley SDG, they used the two-phase slicing algorithm defined by Horwitz et al.[9] to compute the static slices of object-oriented programs.

3.1.2 Aspect-Oriented Slicing

Zhao [1] proposed an approach for slicing *aspect-oriented program (AOP).* The dependence based intermediate representation of the input AOP proposed by Zhao is named as *aspect-oriented system dependence graph (ASDG).* ASDG is in turn an extension to the previously defined dependence graph [4], that represent the features of an aspect-oriented program. The two-phase slicing algorithm in [9] is used to compute static slices of AOP on those ASDG.

Singh et al. [16], proposed a different approach for slicing AOP. He separately sliced non-aspect and aspect part of the programs and introduced the concept of a point-cut table. Singh et al., instead of introducing new nodes or dependencies in the SDG of non-aspect part, simply stored the information of point-cuts in the table, to decrease the complexity of the algorithm.

Braak [17], proposed a detailed and refined construction algorithm of an aspect-oriented system dependence graph, defined by him. He is basically extending Zhao [1] and creating the SDG by using the base code of the program. He has modeled the inter-type declarations in

ASDG. He has given the scope of addition and integration of other features of AOP. He then used the slicing algorithm defined by Horwitz et al. [9] to slice AOP.

3.2 AOP Software Metrics

Zakaria et al. [18], gave a brief overview of the need of metrics for AOP. They explained the C&K metrics suite and also gave an explaination of the affect of Aspect-Oriented methodolgies on the various metrics defined under C&K metrics suite. They based their analysis on case studies found in the literature about re-designing some existing software systems to incorporate the aspect-oriented paradigm.

Zhao [19], proposed some metrics for aspect-oriented software which have been designed to quantify the information flows in the AOP. The metrics proposed by Zhao can be used to measure the complexity of aspect-oriented software from various viewpoints. He defined those metrics based on the dependence graphs defined at three levels (*Module-Level, Aspect-Level, System-Level*) to explicitly represent various dependencies in AOP.

Ceccato et al. [20], extended the C&K metrics suite, for OOPs, in order to make them applicable to the AOP software. They proposed 10 metrics to properly measure the AOPs.

- ***WOM (Weighted Operations in Module):*** WOM is the number of operations for any given module.
- ***DIT (Depth of Inheritance Tree):*** DIT is equal to the length of the longest path from a given module to the class/aspect hierarchy root.
- ***NOC (Number of Children):*** NOC is the cardinality of immediate subclasses or sub-aspects of a given module.
- ***CAE (Coupling on Advice Execution):*** CAE is equal to the number of aspects which contain advices, which are possibly triggered by the execution of operations in a given module.
- ***CIM (Coupling on Intercepted Modules):*** CIM is the number of modules or interfaces clearly named in the pointcuts that belongs to a given aspect.
- ***CMC (Coupling on Method Call):*** CMC is the number of interfaces or modules declaring methods that are possibly called by the given module.
- ***CFA (Coupling on Field Access):*** CFA is the number of interfaces or modules declaring fields that are accessed by the given module.
- ***RFM (Response for a Module):*** RFM depicts the methods and advices which are potentially executed in response to a message received by the given module.
- ***LCO (Lack of Cohesion in Operations):*** LCO refers to the pairs of operations working on different class fields except for the pairs of operations working on common fields (zero if negative).

- ***CDA (Crosscutting Degree of an Aspect):*** CDA shows the number of modules which are affected by the pointcuts or introductions in the given aspect.

These metrics are proposed by extending the C&K suite and then implemented by Ceccato et al.

Piveta et al. [18], made a subset of metrics after considering two different sets of metrics. The different sets of metrics are as follows:

- Metrics adapted from C&K suite.
  1. Lines of Code(LOCC)
  2. Weighted Operations in Module(WOM)
  3. Depth of Inheritance Tree(DIT)
  4. Number of Children(NOC)
- Metrics specifically defined for aspect-oriented softwares from the metrics suite proposed by Ceccato et al. [20].
  1. Crosscutting Degree of an Aspect(CDA)
  2. Coupling on Advice Execution(CAE)

They discussed how each of the metrics taken into account could be used to identify shortcomings in existing AOPs. They gave *rigorous definitions* and *usage scenarios* of the metrics. They also *interpreted the collected empirical data*, while discussing the scope of values and comparing those in aspects and classes. At the end they did an *analytical evaluation* of the selected metrics against an established standards for validation of the results.

Zhao et al. [21], proposed an approach for measuring the cohesion of aspects based on dependence. He discussed the tightness of the aspect based on three different dependencies namely

- Inter-attribute
- Module-attribute
- Inter-module

These three types could be easily used to measure the aspect cohesion be it independently or after integration of all the three. They discussed different properties of these dependencies and proved based on those properties, that their proposed approach satisfy the properties of cohesion. The cohesion measures proposed by Zhao et al. focuses mainly on the features of aspect only and the environment is not taken much into account.

Chapter 4

Aspect-Oriented Intermediate Graph Representation

We have proposed an intermediate graph representation called Extended Aspect-Oriented System Dependence Graph (EAOSDG), which represents the features of the Aspect-Oriented Programs. Some new types of edges (such as *weaving edges*) are required 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.

4.1 Dependencies

The EAOSDG is a directed graph, $G = (V, E)$, where V is the set of nodes. Each node $n \in V$ corresponds to the bytecode version of the statements of the AOP written in AspectJ [1]. Each edge $e \in E$ corresponds to different dependencies present in an AOP as shown in Figure 4.4. The EAOSDG shown in Figure 4.3 contains the following set of edges as defined below:

- ***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 data dependent on n_1.

- ***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 transfer of control from n_1 to n_2.

- ***Class Membership Edge*:** The class dependence 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.

- **Summary Edge:** If the parameter-out node (n_1) is transitively dependent on the parameter-in node (n_2), then there is a summary edge $n_2 \overset{call}{\rightarrow} n_1 \in E$.

- **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 $n_1, n_2 \in V$ wherein n_1 is the method calling node and n_2 the method declaration node.

[1]eclipse.org/aspectj/

- **Parameter-In Edge:** The Parameter-In edge, $n_1 \overset{P_{in}}{\to} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$ wherein n_1 is the actual parameters and n_2 is formal parameters.
- **Parameter-Out Edge:** The Parameter-Out edge, $n_1 \overset{P_{out}}{\to} n_2 \in E$, is defined between two nodes n_1 and n_2, where $n_1, n_2 \in V$ wherein n_1 is the return nodes and n_2 node accepts the value of the calling method.
- **Weaving Edge:** Weaving edge, $n_1 \overset{Weav}{\to} n_2 \in E$, connects the non-aspect part with the aspect part of EAOSDG.
- **Inheritance Edge:** Inheritance edge, $n_1 \overset{Inh}{\to} n_2 \in E$, is defined between two modules denoted by n_1 and n_2 where n_2 inherits the properties of n_1.

4.2 EAOSDG Generation

EAOSDG is generated using a series of steps as explained in Algorithm 1. First, the nodes are created for each statement of the program. Then, the *data dependence*, *control dependence* and *class membership* edges are added depending on the nodes and the respective usage of the edges. We then add the *call edges* for the method callings and *param-in/param-out edges* for the parameters. After that the *summary edges* are added for the transitive dependency between *param-out* and *param-in* nodes. After the creation of pointcut nodes for the aspect part of the projects, the *weaving edges* are added to connect the aspect and non-aspect part of the program.

Algorithm 1

```
INPUT: AOP Program.
OUTPUT: A SDG $G < V, E >$.

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 Data Dependency, Control Dependency, and Class Membership Edge in between nodes
   by analysing the programs.
     • Add a Data Dependent edge, $n_1 \overset{dd}{\to} n_2$, if $n_2$ is data dependent on $n_1$.
     • Add a Control Dependent edge, $n_1 \overset{cd}{\to} n_2$, if $n_1$ transfers the control to $n_2$.
     • 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 corre-
       sponding 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 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.
```

Table 4.1: Graph construction time for different projects.

Sl. No	Project Name	No. of Nodes	No. of Edges	Time for EAOSDG Generation	Details
1	Addition	38	65	85 ms	This program take input of 2 integers and return the sum if the sum is not zero. Else it returns 1.
2	Prime	44	82	115 ms	This program take input of an integer(n) and gives output of all the prime numbers from 1 to n.
3	Server - Client	119	195	118 ms	This project uses socket programming to create a server-client connection in between two systems.
4	Elevator	540	997	302 ms	This project simulates elevator system.
5	ATM Simulation	887	1650	1391 ms	This project simulates the ATM system on a distributed environment.
6	Tetris Project	1566	2317	1672 ms	This is a very popular game, where we arrange blocks.
7	Design Patterns	4137	3752	2671 ms	This is the AspectJ implementation of GoF design patterns.

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.

```
4: Add Summary Edge between nodes if the Param-Out node is transitively dependent on the
   Param-In node.
5: Create nodes for pointcut nodes for Aspect part of the project.
     • Add call edge between pointcut nodes and advices.
6: Add Weaving Edge to connect the Aspect and Non-Aspect part of the project.
     • Add a weaving edge, $n_1 \overset{Weav}{\rightarrow} n_2$, if $n_1$ is the before advice node and $n_2$ is the corre-
       sponding method entry node.
     • Add a weaving edge, $n_1 \overset{Weav}{\rightarrow} n_2$, if $n_2$ is the after advice node and $n_1$ is the corre-
       sponding method entry node.

end
```

4.3 Implementation

It takes the bytecode of the program as input and then it generates the intermediate graph named *Extended Aspect-Oriented System Dependence Graph (EAOSDG)*. We used a sample program shown in Figure 4.2. The EAOSDG generated by the tool is shown in Figure 4.3.

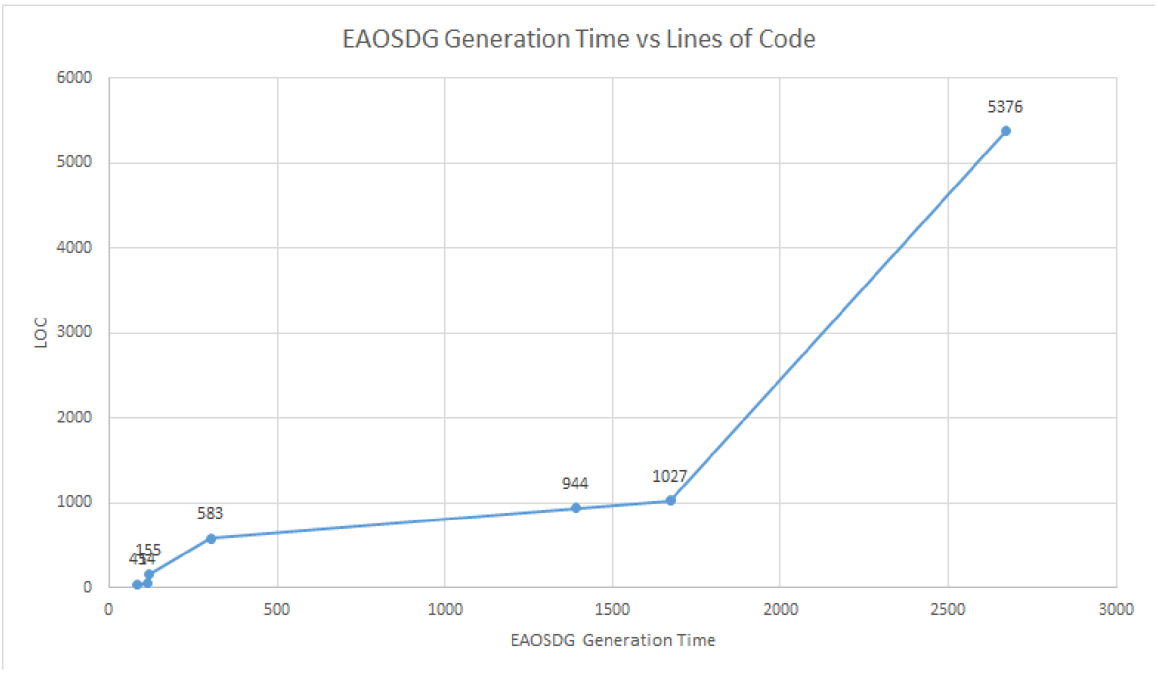
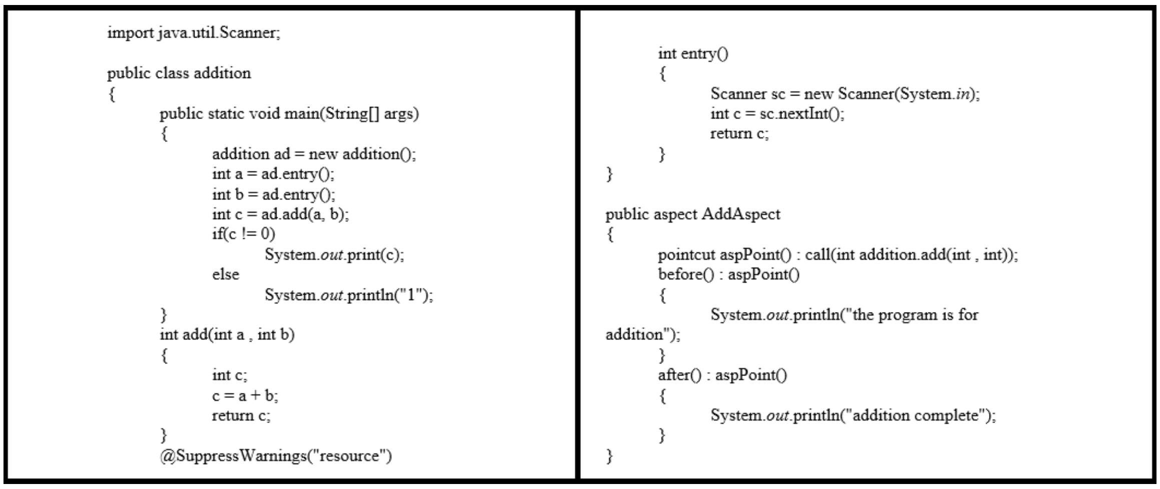


Figure 4.1: EAOSDG Generation Time vs Lines of Code.



```
import java.util.Scanner;

public class addition
{
        public static void main(String[] args)
        {
                addition ad = new addition();
                int a = ad.entry();
                int b = ad.entry();
                int c = ad.add(a, b);
                if(c != 0)
                        System.out.print(c);
                else
                        System.out.println("1");
        }
        int add(int a , int b)
        {
                int c;
                c = a + b;
                return c;
        }
        @SuppressWarnings("resource")
        int entry()
        {
                Scanner sc = new Scanner(System.in);
                int c = sc.nextInt();
                return c;
        }
}

public aspect AddAspect
{
        pointcut aspPoint() : call(int addition.add(int , int));
        before() : aspPoint()
        {
                System.out.println("the program is for
addition");
        }
        after() : aspPoint()
        {
                System.out.println("addition complete");
        }
}
```

Figure 4.2: Example Program.

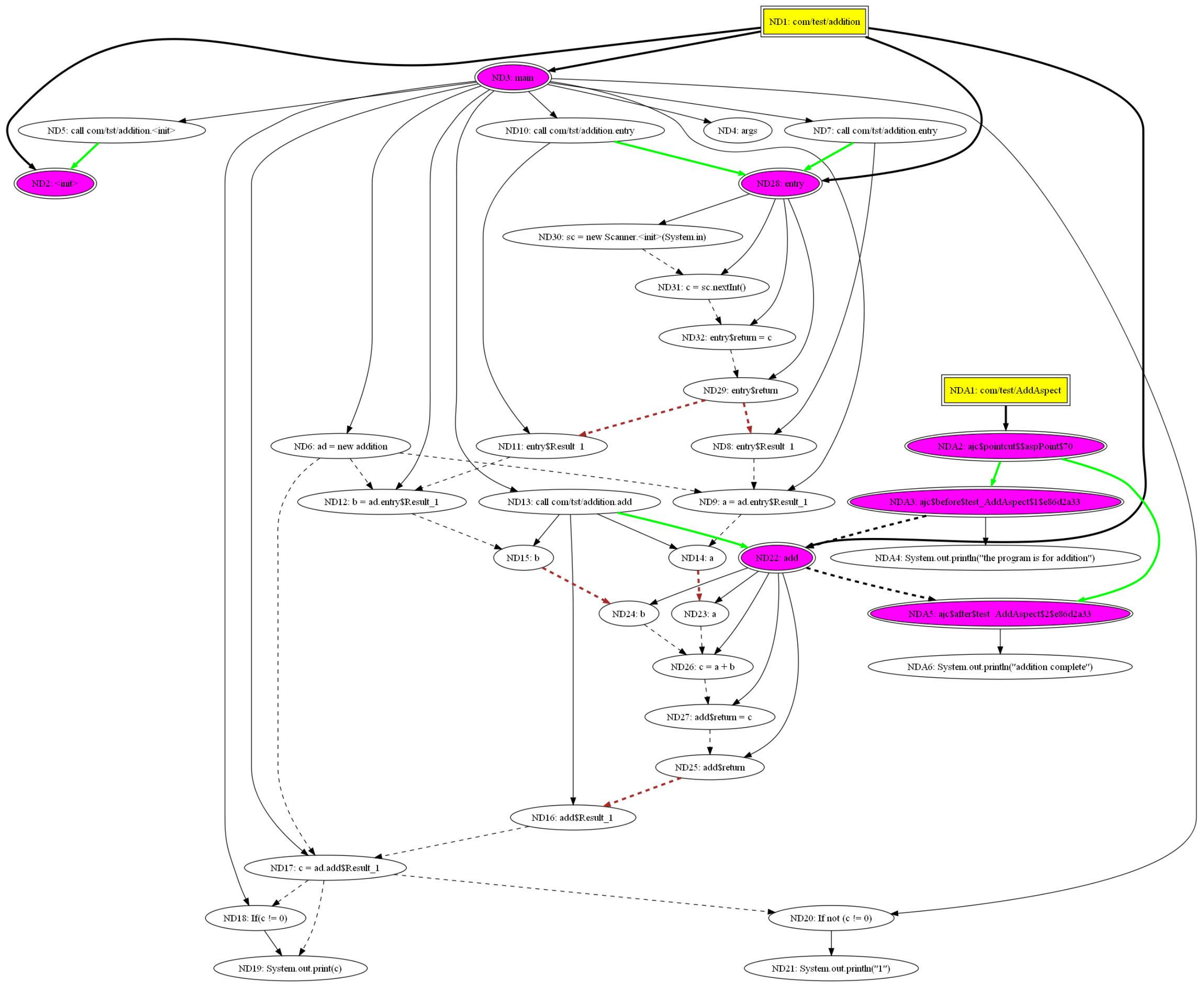

ND1: com/test/addition
ND3: main
ND5: call com/tst/addition.<init>
ND10: call com/tst/addition.entry
ND4: args
ND7: call com/tst/addition.entry
ND2: <init>
ND28: entry
ND30: sc = new Scanner.<init>(System.in)
ND31: c = sc.nextInt()
ND32: entry$return = c
ND29: entry$return
NDA1: com/test/AddAspect
NDA2: ajc$pointcut$$aspPoint$70
ND6: ad = new addition
ND11: entry$Result_1
ND8: entry$Result_1
ND12: b = ad.entry$Result_1
ND13: call com/tst/addition.add
ND9: a = ad.entry$Result_1
NDA3: ajc$before$test_AddAspect1e86d2a33
ND15: b
ND14: a
ND22: add
NDA4: System.out.println("the program is for addition")
ND24: b
ND23: a
NDA5: ajc$after$test_AddAspect2e86d2a33
ND26: c = a + b
NDA6: System.out.println("addition complete")
ND27: add$return = c
ND25: add$return
ND16: add$Result_1
ND17: c = ad.add$Result_1
ND18: If(c != 0)
ND20: If not (c != 0)
ND19: System.out.print(c)
ND21: System.out.println("1")

Figure 4.3: EAOSDG of the example program given in Figure 4.2.

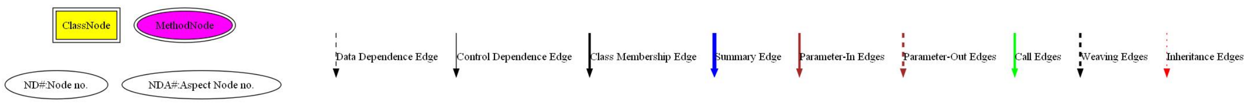


Figure 4.4: Legends for EAOSDG

We used bytecode instead of source code for creation of the graph. ASM is an all-purpose Java bytecode manipulation and analysis framework. This framework gives us the information of all the dependencies, variables and different states of the program (in the form of nodes) after taking input of the bytecode of Java in the form of *.class* file. A Java class file is a file containing Java bytecode that can be executed on the *Java Virtual Machine (JVM).*

We developed a tool that automates the SDG generation by correctly identifying the required dependencies present in an aspect-oriented program. This process is a scalable approach to generate the graph for a project with multiple classes as shown in Table 4.1. Due to automation of the graph generation, the time required to analyze the code is saved. For example, a project having thirty classes, fourteen aspects, and three thousand eight hundred and fifty six lines of code, takes not more than $2671ms$ to generate the graph as shown in Table 4.1. This otherwise, would have taken a lot of time to manually generate the graph as given in the existing literature [17, 10]. A lot of dependencies that may remain undetected during manual graph generation are addressed in this automated process.

The prototype tool developed to implement the slicer is an extension of an open-source API, *Java System Dependence Graph API* [2]. The developed tool automates the process of SDG generation of aspect-oriented programs. The generated graph is named *Extended Aspect-Oriented System Dependence Graph (EAOSDG).* The SDG generation tool takes the bytecode of the aspect-oriented program as input and generates the SDG as output as shown in Fig. 4.5.

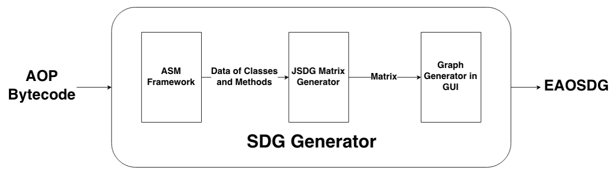


Figure 4.5: EAOSDG Generation Process.

The bytecode of AOP is given as input to the tool. It is then sent to the ASM framework inside the tool. 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 used in this tool are summarized in Table 4.2.

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.

For the evaluation of our proposed technique, we have implemented some case studies. We have given the path of the folder, that contains the bytecode generated by the program, as input to the tool. The tool generates the EAOSDG for the program. Then the tool prompts for entering the slicing criterion node from the EAOSDG. We obtained several slices by entering different slicing criteria for individual case studies and the outcome of this experiment is given below. We have downloaded a few 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 *Addition* and *ATM Simulation*) are developed as laboratory assignments. We constructed different EAOSDGs for these programs

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

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 4.1.

4.4 Summary

In this chapter we defined an intermediate graph representation for aspect-oriented programs, and coined the term Extended Aspect-Oriented System Dependence Graph (EAOSDG). AOP is a technique for improving separation of concerns in software design and implementation. It does so by introducing various features into object-oriented paradigm such as aspects, join-points, point-cuts, advices, code-introductions etc. These features make the existing representations obsolete thus creating the need for introduction of a new scheme. We have defined the new scheme with respect to the new features of the AOP and also all the dependencies introduced to accommodate the features of AOP. We have also given the algorithm for the generation of the graph. The implementation of the EAOSDG generation is done using ASM framework, which is an all-purpose Java bytecode manipulation and analysis framework, which gives us the data on the dependencies between different states of the program. We have used seven case studies from different benchmark software repositories and implemented our algorithm on those case studies and shown the results we got by implementing that. At the end of this chapter, we have EAOSDG for any aspect-oriented program which can be used for program comprehension.

Chapter 5

Slicing of Aspect-Oriented Programs

In this chapter, we discuss the proposed approach to compute the static slices of AOP on the graph constructed by parsing the bytecode of the input AOP.

5.1 Computation of Slices

The two phase slicing algorithm given by Horwitz, Reps and Binkley[9] and used by Zhao[1] does not 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 part of the program respectively. The aspect part of the program is not handled properly by the algorithm.

We extended the two-phase slicing algorithm by Horwitz et al.[9] to bytecode slicing algorithm. This algorithm finds a static slice for a given slicing criterion ‘s‘, which comprises of those program statements that affect the value of the slicing criterion.

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 those vertices in EAOSDG that are reached during the first phase of traversal. Then 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. In the third and last phase, it traverses backward from all the vertices which were marked during 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 in EAOSDG marked during the first, second and third phases of traversal.
Algorithm 2 presents the pseudo code of the proposed slicing algorithm.

Algorithm 2

```
INPUT: A SDG $G < V, E >$, a slicing criterion $s$.
OUTPUT: The Slice $S$ for $s$.
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 $n$
```

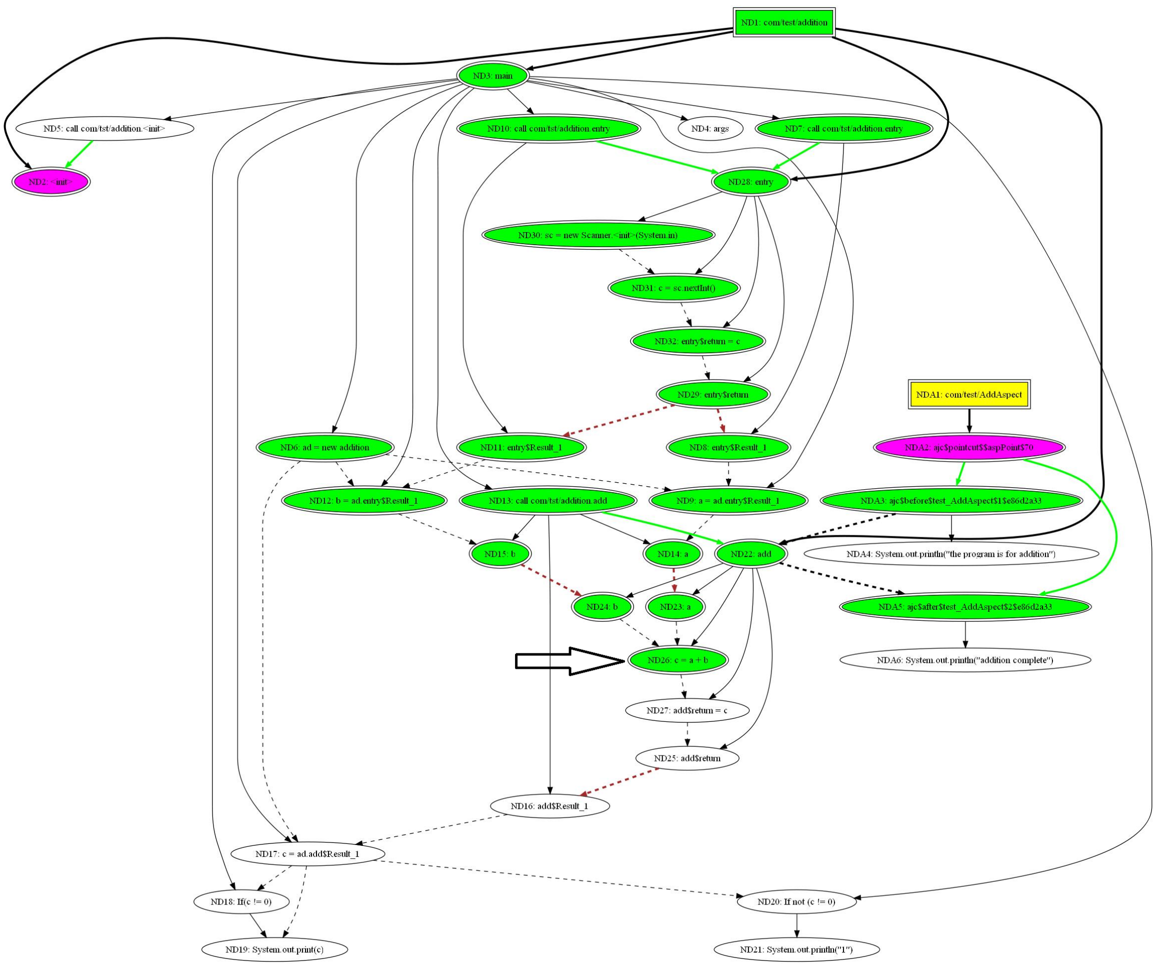

ND1: com/test/addition
ND3: main
ND5: call com/tst/addition.<init>
ND10: call com/tst/addition.entry
ND4: args
ND7: call com/tst/addition.entry
ND2: <init>
ND28: entry
ND30: sc = new Scanner.<init>(System.in)
ND31: c = sc.nextInt()
ND32: entry$return = c
ND29: entry$return
NDA1: com/test/AddAspect
ND6: ad = new addition
ND11: entry$Result_1
ND8: entry$Result_1
NDA2: ajc$pointcut$$aspPoint$70
ND12: b = ad.entry$Result_1
ND13: call com/tst/addition.add
ND9: a = ad.entry$Result_1
NDA3: ajc$before$test_AddAspect1e86d2a33
ND15: b
ND14: a
ND22: add
NDA4: System.out.println("the program is for addition")
ND24: b
ND23: a
NDA5: ajc$after$test_AddAspect2e86d2a33
ND26: c = a + b
NDA6: System.out.println("addition complete")
ND27: add$return = c
ND25: add$return
ND16: add$Result_1
ND17: c = ad.add$Result_1
ND18: If(c != 0)
ND20: If not (c != 0)
ND19: System.out.print(c)
ND21: System.out.println("1")

Figure 5.1: Sliced EAOSDG of the example program given in Figure 4.2.

```
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** $e \notin \{p_o\}$ **then**
9:                  $W_2 = W_2 + \{m\}$
10:             **else**
11:                 $W_3 = W_3 + \{m\}$
12:     **for all** $n \rightarrow m$ **do**
13:         **if** $m \notin S \&\& e \in \{weav\}$ **then**        ▷ if e is an outgoing weaving edge
14:             $W_3 = W_3 + \{m\}$
15: **while** $W_2 ! = \phi$ **do**        ▷ phase 2
16:     $W_2 = W_2 - \{n\}$        ▷ process the next node in $W_2$
17:     **for all** $m \rightarrow n$ **do**        ▷ handle all incoming edges of n
18:         **if** $m \notin S$ **then**
19:             $S = S + \{m\}$
20:             **if** $e \notin \{p_i, call, weav\}$ **then**        ▷ if e is not a parameter in, call or weaving edge
21:                 $W_2 = W_2 + \{m\}$
```

```
22:             else if $e \in weav$ then
23:                 $W_3 = W_3 + \{m\}$
24:     for all $n \rightarrow m$ do
25:         if $m \notin S \&\& e \in \{weav\}$ then          ▷ if e is an outgoing weaving edge
26:             $W_3 = W_3 + \{m\}$
27: while $W_3 ! = \phi$ do                                    ▷ phase 3
28:     $W_3 = W_3 - \{n\}$                          ▷ process the next node in $W_3$
29:     for all $m \rightarrow n$ do                    ▷ handle all incoming edges of $n$
30:         if $m \notin S$ then
31:             if $e \notin \{p_i, call\}$ then     ▷ if e is not a parameter in or call edge
32:                 $S = S + \{m\}$
33:                 $W_3 = W_3 + \{m\}$
    return $S$
```

end

5.2 Complexity Analysis

The EAOSDG is a graph stored in a specific data structure, a modified adjacency list, which has nodes and edges as objects of different classes. If the number of nodes in the graph is n and the number of edges is e, then the space complexity of storing the graph is of order $O(ne)$.

This algorithm has three phases. For each phase there is an inner and an outer loop. If the number of edges in the EAOSDG is e and number of nodes in the EAOSDG is 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 complexity, the worst case time complexity of bytecode slicer is $O(ne)$.

5.3 Working of Algorithm

We have used a simple *Addition* program for showing the working of our algorithm as shown in Figure 4.2. This program take input of 2 integers and return the sum if the sum is not zero. Else it returns 1. In the EAOSDG given in Figure 4.3, *ND26* representing the statement $c = a + b$ is taken as the slicing criterion. In the proposed slicing algorithm, the initial state of the data structure used is given as follows:
$S = ND26\{c = a + b\}$
$W_1 = \{ND26\}$
$W_2 = \phi$
$W_3 = \phi$

In phase 1, we pop one node at a time from W_1, then add the node into S_{Phase1} (if it is not present before) and check for all incoming edges onto the present node. Then, we add the source nodes of these edges into W_2, if the edge is *parameter-out* edge. If the edge is a *weaving* edge, then we add the source node into W_3. 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 those edges into W_3.
This process is continued till W_1 is empty.

After phase 1 we have:
S_{Phase1} = {ND26, ND23, ND24, ND14, ND15, ND13, ND9, ND12, ND8, ND11, ND6, ND7, ND10, ND3, ND1}
$W_1 = \phi$
W_2 = {ND29}
W_3 = {NDA3, NDA5}

In phase 2, we pop one node from W_2, add the node into S_{Phase2} (if it is not present before) and check for all incoming edges onto the present node. If the edge is a *weaving* edge, then add the source nodes of these edges into W_3. Otherwise, we check if the edge is not a *parameter-in* or *call* edge. If it is not so, then we add the source node into W_2. Then, we check for the outgoing *weaving* edges from the popped node and add the destination nodes of those edges into W_3. This process is repeated till W_2 is empty.
After phase 2, we have:
S_{Phase2} = {ND29, ND32, ND28, ND31, ND30}
$W_1 = \phi$
$W_2 = \phi$
$W_3 = W_3$

In phase 3, we pop from W_3, and add the node into S_{Phase3} (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 W_3.
This process is continued till W_3 is empty.
After phase 3, we have:
S_{Phase3} = {NDA3, NDA5}
$W_1 = \phi$
$W_2 = \phi$
$W_3 = \phi$

$S = S_{Phase1} \cup S_{Phase2} \cup S_{Phase3}$.
Hence, for the given slicing criterion, $c = a + b$, the slice computed is:
S = {ND26, ND23, ND24, ND14, ND15, ND13, ND9, ND12, ND8, ND11, ND6, ND7, ND10, ND3, ND1, ND29, ND32, ND28, ND31, ND30, NDA3, NDA5}
The sliced nodes are shown as shaded nodes in Figure 5.1.

5.4 Implementation

We developed a prototype tool that works as shown in Figure 5.2.

The prototype tool developed during the course of the project first creates EAOSDG as mentioned in Chapter. Then the tool prompts for entering the slicing criterion node from the EAOSDG. Then the slicer takes the generated EAOSDG and the slicing criterion as input to produce the slices. We obtained several slices by entering different slicing criteria for individual case studies and the outcome of this experiment is given below.

Based on the EAOSDGs generated for different programs, the corresponding slices are computed. Different number of slices for different programs are computed depending upon the input

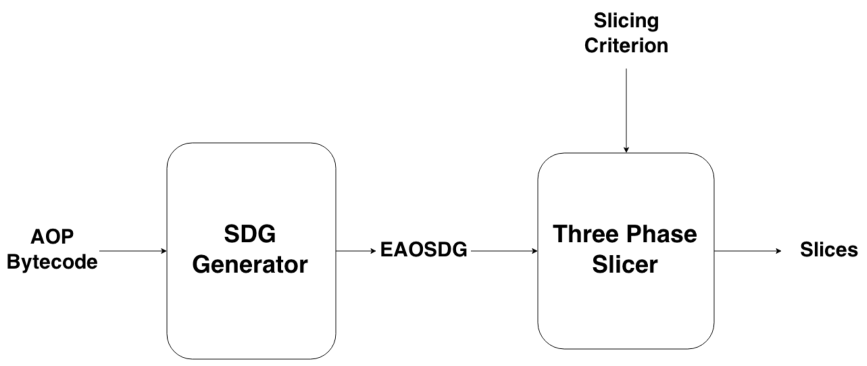


Figure 5.2: Working of the tool.

Table 5.1: Average slicing time for different projects using our approach.

Project Name	Classes	Aspects	LOC	No of Slicing Criterions	Average Slice Size	Average Slicing Time
Addition	1	1	41	4	23	1.39 ms
Prime	1	2	54	5	28	1.68 ms
Server - Client	2	3	155	8	31	1.75 ms
Elevator	5	3	583	14	96	2.48 ms
ATM Simulation	9	3	944	20	132	4.37 ms
Tetris Project	15	4	1027	23	61.26	5.02 ms
Design Patterns	30	14	5376	92	27.78	1.98 ms

slicing criterion. The details of the slices computed such as *Lines of Codes (LOC)*, the *number of slicing criteria* given, *the average number of computed slices* and the *average slice computation time* are shown in Table 5.1.

5.5 Comparison with other 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 [1] for the first time computed the slices for AOPs. In the absence of 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. Singh et al. [16] distinguished the aspect and non-aspect parts in their SDG by creating point-cut table. The SDG in [16] also lacks scalability because of manual graph generation. Braak [17] also gave an approach for aspect slicing based on an intermediate graph that is also manually generated. Unlike the slicing algorithm extended by Zhao [1] and Braak [17], the proposed approach extends

Table 5.2: Comparison of our work with other related work.

Sr. No	Literature Work	Type of Slicing	No. of Types of Edges	Automated Graph Generation
1.	Zhao [1]	Static	4	No
2.	Braak [17]	Static	4	No
3.	Singh et al. [16]	Dynamic	6	No
4.	Our Approach	Static	9	Yes

the slicing alogorthm in [9] 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.

In this final year project, we automated the generation of intermediate graph thus making it possible to correctly represent the dependencies present in a program. Thus, slicing large projects with multiple number of classes becomes feasible and accurate. We identified nine types of dependencies required to construct the EAOSDG. The input of our tool is a *.class* file and the slicing criterion, and the output obtained constitutes the computed slices. The summary of comparison is given in Table 5.2.

5.6 Summary

In this chapter, we proposed a slicing algorithm for AOP. In the previous chapter, we had introduced a new intermediate graph representation. With the introduction of a new intermediate representation, the existing slicing algorithms become useless as they are usually based on the representation itself. So, we extended the standard two phase slicing algorithm by adding a third phase which accommodate the aspect part of the project. We have given the proper pseudocode for the three phase slicer and the complexity analysis for the algorithm. The working of the slicing algorithm is also explained properly. Also in this chapter, we have used the slicing algorithm on all the seven case studies introduced in the previous chapter. Various slicing criterions are given as input and the average results are given. At the end of the chapter, we compared our work with various existing works in the literature and given the summarised information regarding the comparison. So, by the end of the chapter our tool can create an EAOSDG for any given AOP and the slice it for a particular slicing criterion given as input in an automated fashion, thus improving the productivity in the software development life cycle by decreasing the manual intervention during the process.

Chapter 6

Software Metrics

Software engineering is the study of creation of high quality software with its cost and schedules anticipated beforehand. One of the important tasks in SDLC or software engineering is controlling of the process of software development, which in turn helps in controlling costs and schedules as well as the softwares quality. Software metrics have been brought up by many researchers as a means to provide quantitative as well as qualitative control of software products. But, the effective use of software metrics is somewhat reliant on the statistical validation of the metrics [22].

Software metrics are used to measure certain characteristics of software. Software metrics are broadly distributed into two categories: software process metrics and software product metrics.

Software product metrics: Software product metrics measure software products such as same code or design documents.

Software process metrics: Software process metrics checks for the degree of software development process like the number of man hours charged to the development activities in the design and coding phases.

Several sets of object-oriented metrics have been proposed as a means of measuring if systems under investigation exhibit features of a quality software. One such set which is considered as a standard *Software Metrics Suite*, which is also the first of its kind, is Chidamber and Kemerer [23].

6.1 C&K Metrics

Keeping in mind, the features and properties of OOPs, Chidamber and Kemerer [23] developed a set of six metrics. These metrics attempt to identify definite design traits in object-oriented software like inheritance, coupling, cohesion etc.
The six metrics can be summarised as:

1. ***Weighted Methods per Class (WMC):*** The number of methods in a class is counted in this metrics. WMC was designed to measure the complexity of a class. This metric measures understandability, maintainability, and reusability as follows:

    - The time and effort needed to develop and maintain a particular class is reflected by this metric.

- With the increase in the number of methods there is an increase in the potential impact on the inherited classes as all the methods defined in the class are inherited by the children.
- A class with a large number of methods is more application-specific, and therefore is not likely to be reused.

2. ***Depth of Inheritance Tree (DIT):*** The maximum level of inheritance hierarchy of a class is measured in this metric, where the root of the inheritance tree is inherited from a no class and is at level zero of the tree. DIT was intended to indicate the potential for reuse, and to indicate the complexity of the design and it does so as follows:

    - The classes in the deeper level of the tree or hierarchy are likely to inherit higher number of methods. This in result makes the deeper level classes more complex and difficult in predicting its behaviour.
    - The deeper the inheritance tree is, the more the potential for reuse.

3. ***Number of Children (NOC):*** The number of subclasses belonging to a class is counted in this metric. C&K suggest that the NOC can be used to show the level of reuse in a system, and hence be used as a possible indicator of the degree of testing needed for a system. The efficiency, reusability and degree of testability is depicted by this metrics in the following ways:

    - With the increase in the NOC there is an increase in possibility of improper abstraction of the parent and may be a case of misuse of sub-classing. Also it leads to the increase in reusability since inheritance is a form of reuse.
    - It may require more testing of the methods of that class, thus increase the testing time.

4. ***Lack of Cohesion in Methods (LCOM):*** This metric gives a measure to the lack of cohesion in the methods of a class. The basis of the metrics is that when one entity is occurring in many methods of same class, it results in less cohesive behaviour. This metric evaluates efficiency and reusability as follows:

    - Better class subdivision is indication of high cohesion.
    - complexity is increased because of low cohesion which results in increase in the possibility of errors during the development process. Classes with low cohesion can probably be subdivided into more number of classes which have increased cohesion.

5. ***Coupling Between Objects (CBO):*** This metric measures the coupling level in between classes. Coupling between two classes occurs when methods or variables of a class are used by another class. Excessive coupling prevents reuse. It gives an indication of reusability as follows:

    - The more independent a class is(less coupling value), the more likely it can be reused.
    - With increase in the level of coupling, the system becomes more sensitive to the changes in the design. This makes maintenance more and more difficult.

- Higher value of coupling also reduces the systems understandability. This is because of the fact that the module becomes harder to be understood, changed or correct as it is interconnected with other modules.

6. ***Response for a Class (RFC):*** The occurrences of calls from a class to other classes is counted by this metrics. To be put in simple words, this metric measures the amount of communication of a classs entities with other classes. This metric helps in understanding the maintainability and understandability of a system as follows:

    - With the increase in the number of methods which can be called from a class through messages, the complexity of the class increases.
    - This also leads to the complication of testing and debugging of the class as it demands higher level of understanding for the developer or maintenance engineer.

6.2 Aspect-Oriented Metrics

Not much definite work has been done in metrics for AOP. There have been few works in literature regarding this but one proper suit for AOP metrics is not yet been proposed which can measure the qualitative and quantitative features of AOPs, which can then be used for various purposes such as refactoring, maintenance etc. Three of the major software metrics dedicated to AOPs which mostly covers all the features required, are explained in detail here.

1. ***Average Aspect Complexity (AAC):*** This metrics provides the average aspect size. The assumption behind this metrics is that a large aspect, which contains more code tends to introduce more faults than a small method. Also with large and complex aspects, the basic idea behind AOP of decreasing the complexity of the code is violated. The considerations to be made are:

    - Low value of AAC is desirable as its easy to handle simple aspects rather than complex ones.
    - Higher number of AAC may denote higher number of pointcuts in a single aspect which is not desirable.

2. ***Crosscutting Degree of an Aspect (CDA):*** This metrics checks for the number of modules which are affected by the pieces of advice, inter-type method declarations and constructor declarations for any given aspect. It is used in a lot of cases targeting the separation of concerns. The considerations to be made are:

    - The higher value of this metrics is desired as it is an indicator of the number of modules affected by an aspect and the usefulness of that aspect.
    - It has also being pointed out that even though the higher value of CDA metrics is desirable, the number of modules, explicitly named, in the pointcuts of an aspect must be kept low [20].
    - If the CDA value is unity, the developer needs to refactor the aspect by using inheritance or association mechanisms so as to separate the concerns which have been encapsulated by the aspects.

3. ***Coupling on Advice Execution (CAE):*** This metric weighs the number of aspects affecting a given module [20]. If behaviour of an operation could be altered by an advice, due to interception of a pointcut, there is an implied dependence in between the operation and the advice. In this sense the given module is coupled with the aspect containing that particular advice. It also lets us to believe that any modification in the aspect or advice will lead to the change in the module as well. The values of this metrics can be used as an indicator of interaction of aspect with the non-aspect part of the program. The considerations to be made are:

   - Low values of CAE are good, as higher value of CAE may result to more coupling factor in the class to the aspects affecting it. If the value of CAE is null then that implies no affect of aspects on that particular module.
   - Higher number of affecting aspects may denote aspect interactions and possible precedence conflicts or inconsistencies between the applied aspects.

4. ***Aspect Cohesion:*** Cohesion [23] is a well-recognized structural attribute which represents the degree to which the components are bound together within a software module. It is considered to be a desirable goal in software development, which might result in better values for external characteristics like reusability, maintainability, and reliability. Cohesion has been studied extensively for procedural as well as OOPs, its effects on AOP are not studied in extensive detail up until now. As with the introduction of aspects, new modules like introductions, advice, and pointcuts are also introduced which are different from methods in a class; the present cohesion metrics are not directly applicable to AOP. This results in need of new appropriate measures for proper evaluation of aspect cohesion. There are three dependencies [21] which are to be used in this metrics. The dependencies are based on the attributes and modules defined within the aspect. The cohesion defined using these dependencies is internal to the aspect and is not affected by the external modules or parts of the program. The dependencies are:

   - ***Inter-attribute dependencies:*** If a_1, a_2 are two attributes of an aspect, a_2 is inter-attribute dependent on a_1, denoted by $a_2 \overset{A-A}{\rightarrow} a_1$, if either of the following conditions hold true:
     (a) a_1 is referred (directly or indirectly) in the definition of a_2.
     (b) If the possibility of definition of a_2 is dependent on the state of a_1.
   - ***Inter-module dependencies:*** Let m_1, m_2 be two modules and a be an attribute in a particular aspect. Then m_2 will be inter-module dependent on m_1, denoted by $m_2 \overset{M-M}{\rightarrow} m_1$, if either of the following conditions are true:
     (a) m_2 calls m_1. (*inter-module call dependence.*)
     (b) a is defined in m_1 and is used in m_2 before it is defined in m_1. (*inter-module potential dependence.*)
   - ***Module-Attribute dependencies:*** Let m be a module and a be an attribute in any aspect. Then m is module-attribute dependent on a, denoted by $m \overset{M-A}{\rightarrow} a$, if m refers a.

There are four possible types of modules in any aspect. Because of that, this dependency can have four types: *introduction-attribute, advice-attribute, method-attribute* or *pointcut-attribute dependencies.*

These different Aspect-Oriented Metrics help in evaluation of various quality attributes of AOPs such as size, reliability, reusability, and modularity.

6.3 Implementation

We developed a prototype tool that takes input the EAOSDG. The EAOSDG is generated as shown in Figure 4.5. This tool then works in different ways to measure different metrics which are used in evaluation of the quality of the software.

6.3.1 Lines of code

We developed a small program that takes input the path of the source files of the project. It then iterates over all the files and counts the number of lines. Special considerations are made to make sure no comments or blank lines are taken into account. It adds the value of LOC of all the files (java programs) of the project and give the information. The LOC calculated for the case studies is shown in Table 6.1.

6.3.2 Weighted Operations in Module

This part of the tool takes input the EAOSDG generated and traverses different modules. It then collects the information and give the resultant metric based on the formula:

$$WOM(m) = |M| + |A| + |IM| + |IC|$$

Here,

- $|M|$ is the number of methods associated with module .
- $|A|$ is the number of advices associated with module m.
- $|IM|$ is the number of inter-type method declarations associated with module m.
- $|IC|$ is the number of inter-type constructor declarations associated with m.

The data retrieved from the calculation of WOM metric is shown in Table 6.1.

6.3.3 Depth of Inheritence Tree

Calculation of this metric is relatively easy by using the EAOSDG as input. The inheritance edge, as explained in Section 4.1, is in between two nodes when one module inherits the properties from another module. We traverses through these edges and see the longest path. This gives the value of DIT. The data retrieved from the calculation of WOM metric is shown in Table 6.1.

Table 6.1: Metrics Calculations for LOC, WOM and NOC.

Sr. No	**Project Name**	**Lines of Codes**	**Avg. WOM**	**NOC**	**DIT**
1	Addition	41	2.5	0	0
2	Prime	54	2	0	0
3	Server - Client	155	2.2	2	1
4	Elevator	583	4.88	0	0
5	ATM Simulation	944	1.42	5	1
6	Tetris	1027	4.36	2	1
7	Design Patterns	5376	1.95	30	1

6.3.4 Number of Children

The EAOSDG has inheritance edge which shows that module m_1 (sub-class) and m_2 (super-class) are connected in a way that $m_2 \overset{Inh}{\rightarrow} m_1$. This tool takes input EAOSDG and for any module checks the number of nodes connected to it by inheritance edge and shows the value which can be used from different viewpoints depending on our requirements. The values of NOC for all the case studies is shown in Table 6.1.

6.3.5 Avg. Aspect Complexity

Calculation of this metrics is done using a small program made by us. This program takes input the path of the source files of the project. It then iterates over all the files with .aj extension (the aspects). Special considerations are made to make sure no comments or blank lines are taken into account. The AAC calculated is shown in Table 6.2.

6.3.6 Crosscutting Degree of an Aspect

Calculation of this metrics is done through the three phase slicer. This part of the tool takes input the EAOSDG and run the three phase slicer taking the advices as slicing criterion. As the slice contains all the nodes that are affected by the slicing criterion, so if the advice is given as slicing criterion it will also have the information of the modules affected by the advice. Keeping this definition in mind, we have given the advices for our slicing criterion. From the slice computed it then checks for all the class nodes in it, and makes a list of those. The number of class nodes is the value of CDA metrics. The values of this metrics for different case studies is shown in Table 6.2.

6.3.7 Coupling on Advice Execution

For each module head node or class node, we perform the DFS of EAOSDG on all the edges except for the inheritance edge. During the traversal if we move across the weaving edge we add the advice node on the other end of the edge to the stack of CAE. The size of that stack is the value of CAE metrics for that particular module. The values of this metrics for different case studies is shown in Table 6.2.

Table 6.2: Metrics Calculations for CDA, CAE and Aspect Cohesion.

Sr. No	Project Name	AAC	CDA	CAE	Aspect Cohesion
1	Addition	20	1	2	4
2	Prime	16.5	1	3	1
3	Server - Client	23	1.33	2.5	1.33
4	Elevator	29.33	2.67	3	1.67
5	ATM Simulation	32.67	3.33	2	1.33
6	Tetris	47.75	2.25	2.33	3.88
7	Design Patterns	33.25	1.54	0.81	1.93

6.3.8 Aspect Cohesion

In our tool we calculate the cohesion in three different phases depending on the three dependencies explained before.
The dependencies are based on the attributes and modules defined within the aspect.

- For *inter-attribute* dependency, the tool checks for the definitions of different attributes in EAOSDG. If some another attribute is called in the definition of that attribute, the tool increments the value of the variable holding the value for *inter-attribute* dependency of Aspect Cohesion.

- For *inter-module* dependency, the tool checks the part of EAOSDG which is showing the module. If any other module is called from that module (call edge in the module to another module), there is *inter-module* dependency then, and the tool increments the value of the variable holding the value for *inter-module* dependency of Aspect Cohesion.

- For *module-attribute* dependency, the tool checks the part of EAOSDG that is showing the module. If the module uses the pre-defined value of that attribute then the tool increments the value for *module-attribute* dependency of Aspect Cohesion.

After these phases, the tool adds the values of all the three variables and gives output Aspect Cohesion of that particular aspect. The values of this metrics for different case studies is shown in Table 6.2.

6.4 Summary

In this chapter, we explained the different metrics that are present in the literature for object-oriented as well as aspect-oriented programs. There have been various metrics suite in the literature but the one which is considered as the standard set is C&K Metrics suite, which is explained properly. This metrics suite is for object-oriented paradigm. There have been a few metrics defined for the AOP and they have been explained as well. After thorough study of all the metrics, we have compiled a metrics suite applicable for AOP and implemented those eight metrics and incorporated those in the tool. The usages of all the eight metrics are explained properly and these are used for measuring quantitatively and qualitatively the different aspects of all the case studies used so far. There are five size based (quantitative measures) and three process based (qualitative measures) metrics in the compiled suite that are extended so as to be able to extract values from the EAOSDG using graph manipulation techniques and/or the three phase slicer. By the end of the chapter, we have a compiled metrics suite for AOPs, and an automated tool which can extract information of the metrics using the bytecode of the project by creating EAOSDG and slicing.

Chapter 7

Conclusion

Aspect-Oriented programming is a new programming technique which is defined for improvement in separation of concerns in software design and implementation. In this work, we have introduced a new intermediate graph representation for AOP. We have coined the term EAOSDG for that representation and explicitly defined all the various dependencies being used in that graphical representation. This SDG properly depict the various features of AOP using these dependencies.

After the creation of EAOSDG, we have proposed a slicing algorithm which can work on that particular representation that is an extension of the standard two phase slicer. We have added one more phase to accommodate the aspect part of the program. We have given the proper pseudocode and the complexity analysis of the algorithm. We have also compared the proposed algorithm with the existing ones in the literature and explained the pros of our proposed approach.

We have also compiled a metrics suite for quantitative and qualitative measurements of AOP. This suite is compiled by using a few existing metrics and extending those to accommodate the changes brought by AOP to OOP. We have also introduced a few metrics for that particular suite. We have compiled the whole suite with eight metrics comprising of five for quantitative measurement and three for qualitative measurements.

We have made a prototype tool which takes input the bytecode of the project and gives detailed analysis of the metrics by using the EAOSDG generator and three phase slicer. There are a few graph manipulation techniques that are also used as per the definition of the metrics. This tool is automated thus decreasing the manual intervention and increasing the productivity in the software development life cycle.

Dissemination of Work

1. **Dishant Munjal**, Jagannath Singh, Subhrakanta Panda, D.P. Mohapatra. **Automated Slicing of Aspect-Oriented Programs using Bytecode Analysis.** *In proceedings of The 39th Annual International Computers, Software & Applications Conference (COMPSAC 2015)*, 2015 (Accepted).

2. Jagannath Singh, **Dishant Munjal**, D.P. Mohapatra. **Context Sensitive Dynamic Slicing of Concurrent Aspect-Oriented Programs.** *In proceedings of The 21st Asia-Pacific Software Engineering Conference (APSEC 2014)*, 2014.

References

[1] J. Zhao, "Slicing Aspect-Oriented Software," in *Proceedings of the 10th International Workshop on Program Comprehension*, pp. 251–260, IEEE, 2002.

[2] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[3] J. T. Lallchandani and R. Mall, "Computation of Dynamic Slices for Object-Oriented Concurrent Programs," in *Proceedings of the 12th Asia-Pacific Software Engineering Conference, APSEC'05.*, IEEE, 2005.

[4] L. Larsen and M. J. Harrold, "Slicing Object-Oriented Software," in *Proceedings of the 18th International Conference on Software Engineering (ICSE)*, pp. 495–505, IEEE, 1996.

[5] G. Kiczales and E. Hilsdale, "Aspect-Oriented Programming," in *ACM SIGSOFT Software Engineering Notes*, vol. 26, p. 313, ACM, 2001.

[6] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach.* CRC Press, 2014.

[7] W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability," *Journal of Systems and Software*, vol. 23, no. 2, pp. 111 – 122, 1993. Object-Oriented Software.

[8] M. Weiser, "Program Slicing," in *Proceedings of the 5th International Conference on Software Engineering(ICSE)*, pp. 439–449, IEEE Press, 1981.

[9] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing using Dependence Graphs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 1, pp. 26–60, 1990.

[10] S. N. Singh and L. Singh, "Study of Current Program Slicing Techniques," in *Proceedings of the 5th International Conference-Confluence The Next Generation Information Technology Summit (Confluence)*, pp. 810–814, IEEE, 2014.

[11] B. Xu, Z. Chen, and H. Yang, "Dynamic Slicing Object-Oriented Programs for Debugging," in *Proceedings of the 2nd IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 115–122, IEEE, 2002.

[12] D. P. Mohapatra, R. Mall, and R. Kumar, "Computing Dynamic Slices of Concurrent Object-Oriented Programs," *Information and software technology*, vol. 47, no. 12, pp. 805–817, 2005.

[13] T. Ishio, S. Kusumoto, and K. Inoue, "Program Slicing Tool for Effective Software Evolution using Aspect-Oriented Technique," in *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pp. 3–12, IEEE, 2003.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, *Aspect-Oriented Programming.* Springer, 1997.

[15] T. Elrad, R. E. Filman, and A. Bader, "Aspect-Oriented Programming: Introduction," *Communications of the ACM*, vol. 44, no. 10, pp. 29–32, 2001.

[16] J. Singh and D. P. Mohapatra, “A Unique Aspect-Oriented Program Slicing Technique,” in *International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pp. 159–164, IEEE, 2013.

[17] T. ter Braak, “Extending Program Slicing in Aspect-Oriented Programming with Intertype Declarations,” in *Proceedings of the 5th Twente Student Conference on IT*, 2006.

[18] A. A. Zakaria and H. Hosny, “Metrics for Aspect-Oriented Software Design,” in *Proceedings of Third International Workshop on Aspect-Oriented Modeling, AOSD*, vol. 3, Citeseer, 2003.

[19] J. Zhao, “Towards a Metrics Suite for Aspect-Oriented Software,” *Rapport technique*, 2002.

[20] M. Ceccato and P. Tonella, “Measuring the Effects of Software Aspectization,” in *1st Workshop on Aspect Reverse Engineering*, vol. 12, Citeseer, 2004.

[21] J. Zhao and B. Xu, “Measuring Aspect Cohesion,” in *Fundamental Approaches to Software Engineering*, pp. 54–68, Springer, 2004.

[22] W. Li and S. Henry, “Maintenance Metrics for the Object-Oriented Paradigm,” in *Proceedings of First International Software Metrics Symposium.*, pp. 52–60, May 1993.

[23] S. Chidamber and C. Kemerer, “A Metrics Suite for Object-Oriented Design,” *IEEE Transactions on Software Engineering,*, vol. 20, pp. 476–493, Jun 1994.