# DYNAMIC SLICING OF ASPECT-ORIENTED PROGRAMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Bachelor of Technology**
**In**
**Computer Science and Engineering**

By
**DEEPAK KUMAR DAS**
**SHAILESH PANCHAM KHAPRE**


**Department of Computer Science and Engineering**
**National Institute of Technology**
**Rourkela**
2007

# DYNAMIC SLICING OF ASPECT-ORIENTED PROGRAMS

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Bachelor of Technology**

**In**

**Computer Science and Engineering**

By

**DEEPAK KUMAR DAS**

**SHAILESH PANCHAM KHAPRE**

Under the Guidance of

**Dr.D.P.Mohapatra**


**Department of Computer Science and Engineering**

**National Institute of Technology**

**Rourkela**

2007


**National Institute of Technology**

**Rourkela**

# CERTIFICATE

This is to certify that the thesis entitled **"Dynamic Slicing of Aspect Oriented Programs''** Submitted by **Deepak Kumar Das, Roll No:10306027**and **Shailesh Pancham Khapre**, **Roll No: 10306031** in the partial fulfillment of the requirement for the degree of **Bachelor of Technology** in **Computer Science Engineering**, National Institute of Technology, Rourkela , is being carried out under my supervision.

To the best of my knowledge the matter embodied in the thesis has not been submitted to any other university/institute for the award of any degree or diploma.

**Dr.D.P.Mohapatra**

Date: Department of Computer Science

and Engineering

National Institute of Technology

Rourkela-769008

## ACKNOWLEDGEMENT

We avail this opportunity to extend our hearty indebtedness to our guide **Dr.D.P.Mohapatra**, Computer Science Engineering Department, for their valuable guidance, constant encouragement and kind help at different stages for the execution of this dissertation work.

We also express our sincere gratitude to **Dr. S.K.JENA**, Head of the Department, Computer Science Engineering, for providing valuable departmental facilities.


**Submitted by:**


**Deepak Das**
Roll No: 10306027
Computer Science Engineering
National Institute of Technology
Rourkela

**Shailesh Pancham khapre**
Roll No: 10306031
Computer Science Engineering
National Institute of Technology
Rourkela

# CONTENTS

# ABSTRACT

As software application grows larger and become more complex, program maintenance activities such as adding new functionality, debugging and testing consume increasing amount of available resources for software development. In order to cope with this increased complexity, programmer need effective computer supported methods for decomposition and dependence analysis of programs. Program slicing is one method for such decomposition and dependence analysis.

Program slicing is a decomposition technique which extracts program elements related to a particular computation from a program. A program slice consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a slicing criterion.

A program slice can be static or dynamic. Static slice contains all the statements that may affect the slicing criterion for every possible inputs to the program. Dynamic slice contains only those statements that actually affect the slicing criterion for a particular input to the program.

Aspect-oriented programming is a new programming technique proposed for cleanly modularizing the cross-cutting structure of concerns. An aspect is an area of concern that cuts across the structure of a program. The main idea behind aspect-oriented programming (AOP) is to allow a program to be constructed by describing each concern separately.

AspectJ is an aspect-oriented extension to the Java programming language. AspectJ adds new concepts and associated constructs called join points, pointcuts, advices, introductions, and aspects to Java.

Zhao developed the aspect-oriented system dependence graph (ASDG) to represent aspect-oriented programs and used two-pass slicing algorithm to compute static slice of aspect-oriented programs. But the disadvantage of his ASDG is that the weaving process is not represented correctly and this graph cannot be used for dynamic slicing.

Our objective was to develop a suitable intermediate representation of an aspect-oriented program and to develop suitable dynamic slicing technique

# LIST OF FIGURES

# CHAPTER 1

## INTRODUCTION

Program slicing

Categories of Program Slicing

Issues in Program Slicing

# 1. INTRODUCTION

**1.1 PROGRAM SLICING:** Program slicing is a decomposing technique that extracts from program statement relevant to a particular computation. Informally a slice provides the answer to a question "What program statement potentially affect the computation of variable '**V**' at statement '**S**'?". Program slicing is also a program analysis technique. The main application of program slicing includes various software engineering such as program understanding, debugging, testing, program maintenance; etc .A program slice consists of the parts or components of a program that affects the values computed at some point of interest. Program slice are computed with respect to a slice criterion. Typically a slicing criterion consists of a pair (**S, V**), where **S** is the statement number and **V** is the set of variable. A slice of a program **P** with respect to a slicing criterion (**S, V**) is the set of programs **P** that might affect the slicing criterion for every possible input to the program.

## 1.2 Categories of Program Slicing:

Several categories of program slicing exist. The reason for the existence of so many categories of slicing is the fact that different applications require different types of slices. The various categories are:-

1) static slicing and dynamic slicing
2) intra-procedure slicing and inter-procedure slicing

**Static Slicing**

Static slicing technique uses static analysis to derive slices. That is, the source code of the program is analyzed and the slices are computed for all possible input values. A static slice contains all statements that may affect the value of a variable at a program point for every possible input.

**Dynamic Slicing:-**

Dynamic slicing makes use of the information about a particular execution of a program. The execution of the program is monitored, and the dynamic slices are computed with respect to the execution history. Dynamic slicing is more suited to object- oriented program s than static slicing as the computed dynamic slice will contain only those statement that actually affect the slicing criterion.

**Intra-procedure & Inter–Procedure slicing:-**

Intra-procedure slicing computes slices within a single procedure call to other procedure are either not handled at al or handled conservatively. If the program consists of more than one procedure, inter-procedure slicing can be used to derive slices that span multiple procedures. So far object-oriented program, inter –procedure slicing is more useful.

**Other Slicing Categories**:

Program slices can be computed at different abstraction levels. The parts of the program that are considered to be included into the slice can be as big as procedures or as small as nodes of the syntax tree of the program (e.g., statements, expressions, variables, parameters etc.). Slicing at the level of syntax tree nodes (also called at the expression level) gives the most detailed and precise information. However, the resulting slices are no longer executable programs. There are variants of slicing in between the two extremes of static and dynamic, where some but not all properties of the initial state are not known. These are known as *conditioned slices or constrained slices.*

*Modular monadic slicing* has been developed where slices are computed based on the modular monadic semantics of the program analyzed. This method computes slices directly on abstract syntax of the program without constructing intermediate representations such as dependence graphs.

*Hybrid slicing* it is a new form of slicing; it's an approach for redefining static slices using dynamic information.

## 1.3 ISSUE IN PROGRAM SLICING:-

Intermediate representation: In order to slice an object-oriented program, first the program should be represented by a suitable intermediate representation, this representation should correctly represent the object-oriented features.

Memory Representation: the memory requirement for both the intermediate representation and the dynamic slicing algorithm should be as small as possible.

Time Requirement: the time requirement for any slicing algorithms should also be as small as possible.

Correctness: The slicing algorithm should compute correct slices with respect to any given slicing criterion a slice is said to be correct if it contains all the statement that affect the slicing criterion.

# CHAPTER 2

## BACKGROUND

Some Definitions

Program Representation

Applications of Program Slicing

## 2. BACKGROUND

The technique of program slicing has been extended to handle unstructured and multi-procedure programs, structured as well as object-oriented and aspect-oriented programs. Also, these slicing techniques have been applied to diverse problem areas.

### 2.1Some Definitions:

2.1.1 Directed Graphs

A directed graph **G** is a pair (**N, E**) where **N** is a finite non-empty set of nodes, and **E∈N*N** is a set of directed edges between the nodes.

2.1.2 Flow Graph

A flow graph is a quadruple (**N**, **E**, start, stop) where (**N, E**) is a graph. Start belongs to **N** is a distinguished node of in degree 0, & stop belongs to **N** has an out degree 0.

2.1.3 Dominance

If **X** and **Y** are two nodes is a flow graph then **X** dominates **Y** iff every path start to **Y** passes through **X**. **Y** post-dominates **X** iff every path from **X** to stop passes throw **Y**.

### 2.2 Program Representation:

Various types of program representation schemes exist which include high level Source code, pseudo-code, a set of machine instructions in a computer's memory, a flow chart and others. The purpose of each of these representations depends upon the exact context of use. Different representations may be required to facilitate human readability, annotation for verifiability, and transformation for running a program on platforms such as multiprocessors and distributed computers, etc. In the context of program slicing, program representations are used to support efficient automation of slicing.

```
1. main ()
2. {
3.    int x, y, prod;
4.    cin>> x;
5.    cin>> y;
6.    prod = 1;
7.    while(x < 5) {
8.          prod = prod * y;
9.          ++ x; }
10.  cout<< prod;
11.  prod = y;
12.  cout<< prod;
13.  }
```

Figure 2.1:An example program

2.2.1 Control Flow Graph –

The control flow graph is an intermediate representation for program that is useful for dataflow analysis and for many optimizing code transformation. Let the set **N** represent the set of statements of program **P**. The control flow graph of the program **P** is the flow graph **G** = ($\mathbf{N}_1$, **E**, start, stop) where $\mathbf{N}_1 = \mathbf{N} \cap$ {start, stop. An edge (*m*, *n*) belongs to **E** indicates the possible control flow from *m* to *n*.

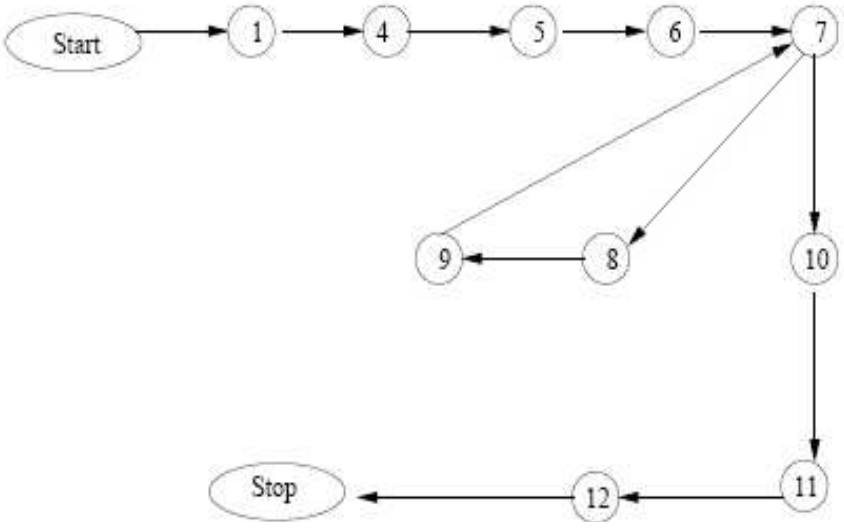


Figure 2.2: CFG of above Program

2.2.2 Data Dependence Graph –

The CFG of a program represent the flow of control through the program. However the concept that is more useful is program analysis is the flow of data through a program. Data flow describes the flow of the values of variable from the point they defined to the points where they are used.

Let **G** be the CFG of a program. A node '*n*' is said to be data dependent on a node '*m*' if there exits a variable '*var*' of the program P such that the following hold:

1) Node '*m*' defines '*var*',
2) Node '*n*' uses '*var*', and

There exits a directed path from '*m*' to '*n*' along which there is node intervening definition of var.

2.2.3 Control Dependence Graph –

Let **G** be the CFG of a program **P**. Let '*x*' and '*y*' be the two nodes in **G**. Node '*y*' is control dependent on a node '*x*' if the following holds :

1) there exits a directed path **D** from '*x*' to '*y*',
2) '*y*' post-dominates every '*z*' is **D** and
3) '*y*' does not post-dominates '*x*'.

2.2.4 Program Dependence Graph –

An important feature of PDG is that it explicitly represents both control and data dependencies in a single program representation. A PDG model of a program as a graph in which the nodes represent the statements, and the edges represents inter- statement data or control dependencies.

The program dependence graph **G** of a program **P** is the graphs **G** = (**N, E**), where each *n*Є**N** represents a statement of the program **P**. The graph contains two kind of directed edges: Control dependence edge and Data dependence edges.

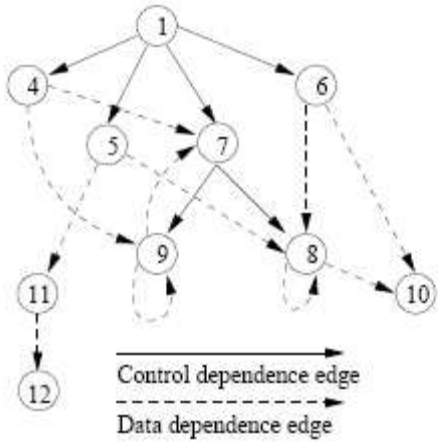


Figure 2.3: PDG of the given Program

2.2.5 System Dependence Graph –

System Dependence Graph the PDG cannot handle the procedure calls. Hence the PDG representation models the main program together with all associated procedures. The models programs is a language with the following properties:-

- A complete program consists of a main program and a collection of auxiliary procedure.
- Procedures end with return statements. A return statement does not include a list of variable.
- Parameters are passed by value- results.

The technique for construction of an SDG consist of first construction a PDG for every procedure, including the main procedure and then adding auxiliary dependence edges which link the various sub graphs together. An SDG includes several types of nodes to model procedure calls and parameters passing:

- Call sites nodes represents the procedure call statements in a program.
- Actual-in and actual-out nodes represent the input and output parameter at call-sites. They are control dependent on the procedure entry node. Control dependence edges and Data dependence edges are used to link the individual PDGs in an SDG. The additional edges that are used to link the PDGs together as follows:
- Call edges link the call-sites nodes with the procedure entry nodes.

- Parameters-in nodes.
- Parameter-out edges link the actual-out nodes.

```
1. main( )
2.   int s, i;
3.   {
4.      s = 0;
5.      i = 1;
6.       while (i < 10) do
7.            {
8.               add(s, i);
9.               inc(i);
10.           }
11.      write(s);
12.   }

13. void add(int a, int b)
14.    {
15.        a = a + b;
16.        return;
17.     }

18. void inc(int z)
19.     {
 20.        add(z,1);
 21.        return;
 22.     }
```

Figure 2.4:An example program

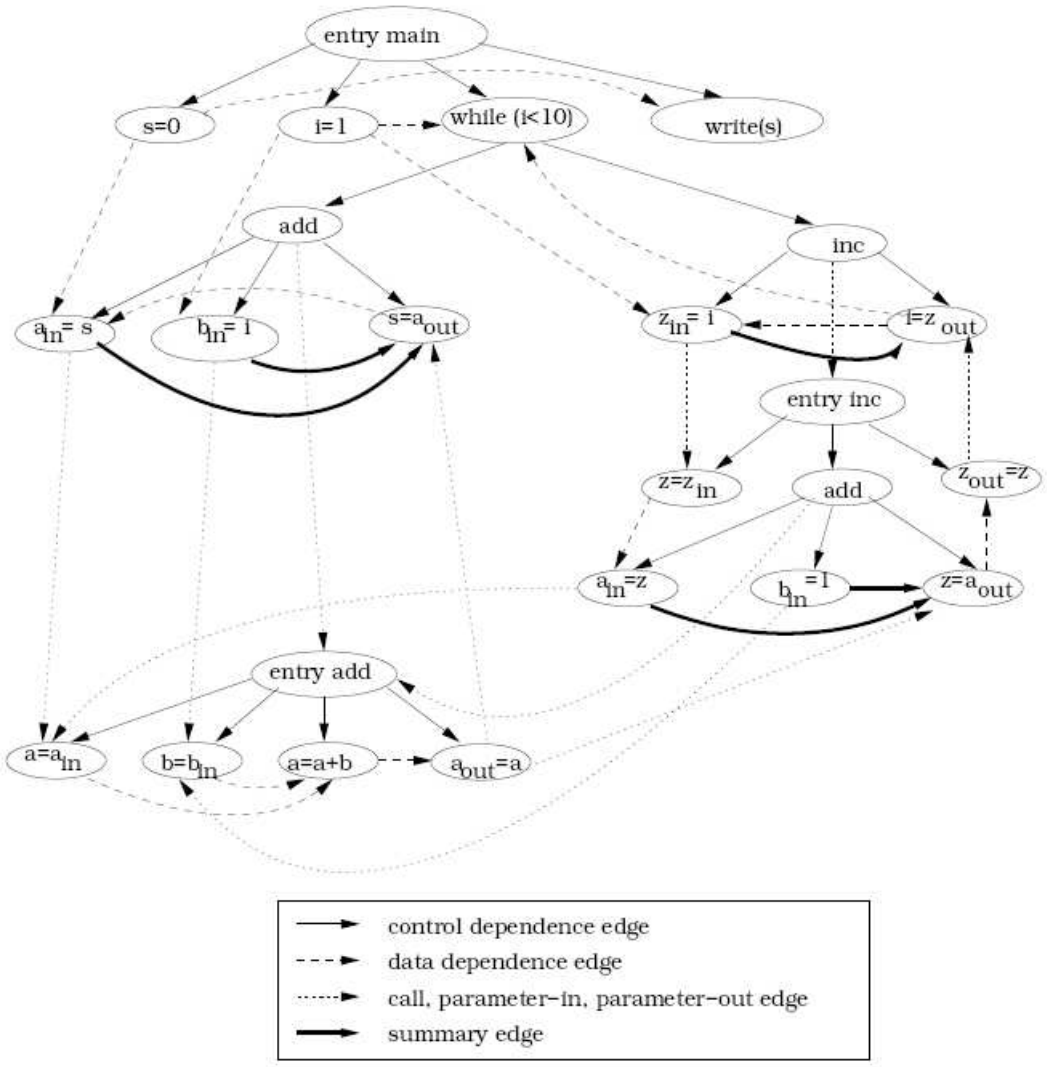


Figure 2.5: SDG of the program

Finally, summary edges are added to represent the transitive dependencies that arise due to procedure calls. A summary edge is added from an actual-in node A to an actual-out node B, if the value associated with the actual–out node B, due to transitive flow of dependence. The transitive flow of dependence may be caused by data dependencies, control dependencies or both.

## 2.3 Application of Program Slicing:-

### Debugging

Program Slicing was originally proposed by observing the operation typically carried out by programmers while debugging a piece of code. Programmers mentally slice a code while debugging it. Program debugging is a major application area of slicing techniques.

### Software Maintenance

Software Maintenance is a costly process because each modification to a program must take into account many complex dependence relationships is the existing software. The main challenge in effective software maintenance is to understand various dependencies in existing software and to make changes to the existing software without introducing new bugs. Hence program slicing is very important to software maintenance.

### Testing

Program slicing is also very useful in testing. It is helpful in reducing the number of test cases required to debug a program.

Other applications of program slicing include:-

- Anomaly detection
- Program specification and reuse.

# CHAPTER 3

## ASPECT ORIENTED PROGRAMMING

Basic Concepts

AspectJ: An Aspect-Oriented Programming Language

Features of AspectJ

### 3.1 Basic concepts

An aspect is an area of concern that cuts across the structure of a program. Concern is defined as some functionality or requirement necessary in the system, which has been implemented in a code structure .Example of aspects are data storage, user interface, platform-specific code, security, distribution, logging, class structure, threading etc. The strength of aspect oriented programming is allowing separation of concerns, by permitting the programmer to create cross-cutting concerns as program modules. Cross-cutting concerns are those parts, or aspects, of the program that end up scattered across multiple program module, and tangled with other module in standard design. For example let us consider the example shown in program. The objective of this program is to transfer an amount from one account to another in banking application. In this example, various cross-cutting concerns such as transaction, security, logging etc. are tangled with the basic functionality.

```
void transfer(Account fromAccount, Account toAccount, int amount){
  if (!getCurrentUser().canPerform(OP_TRANSPER)){
    throw new SecurityException();
  }
  if (amount<0){
    throw new NegativeTransferException();
  }
  if (fromAccount.getBalance()<amount){
    throw new InsufficientFundsException();
  }
  Transaction tx=database.newTransaction();
  try{
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
    tx.commit();
    systemLog.logOperation(OP_TRANSPER,fromAccount,toAccount,amount);
  }
  catch(Exception e){
    tx.rollback();
  }
}
```

Figure3.1 : An example of aspect oriented program

If there is some change in the security consideration for the application, then it would require a major effort since security-related operation appears scattered across numerous methods. This means that the cross-cutting do not get properly encapsulated in there own modules and this increases the system complexity. The goal of Aspect-Oriented programming (AOP) is to make it possible to deal with cross-cutting aspects of system's behaviors as much in isolation as possible. Aspect-Oriented Programming provides specific language mechanisms to explicit capture the cross-cutting structure. To better support the expression of cross-cutting design decision, AOP uses a component language to describe the basic functionality of the system and an aspect language to describe the different cross-cutting properties. The components and the aspects are then combined into a system using an aspect weaver. The aspect weaver makes it possible for an advice to be activated at an appropriate join point during run time. Thus a source code is modified by inserting aspect specific statement at join point.

## 3.2 AspectJ: An Aspect-Oriented Programming Language

AspectJ created by Chris Maeda at Xerox Paul Alto Research Center (PARC), is essentially an aspect-oriented extension to java programming language. In other words AspectJ is compatible with java platform. There are four type of compatibility:

- *Upward compatibility*-All legal java programs must be legal AspectJ programs,
- *Platform compatibility*-All legal java programs must run on java virtual machines,
- *Tools compatibility*-It must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs documentation tools, and design tools.
- *Programmer compatibility*-Programming with AspectJ must feel like a natural extension of programming with java.

## 3.3 Features of AspectJ

AspectJ adds some new features to java. This feature includes join points, pointcuts, advice, aspect, introduction or inter-type declaration.

- *Join Points*- These are well-defined points in the execution of a program, such as methods calls, method execution and method reception join points.

- *Pointcut*-This is mean of referring to collection of join points and certain values of join points.
- *Advice*- It is a method-like construct used to define additional behavior at joint points. This is used too define some code that is executed when a point cut is reached. Advice brings together a point cut and a body of code. There are three types of advice in AspectJ: after, before, around.

  (1). *After*: After advice on a particular join point runs after a program proceeds with that join point.

  (2). *Before*: Before advice runs as a joint point is reached, before a program proceeds

  With the join point .

  (3). *Around*: Around advice on a joint point runs as a join point is reached, and has explicit control over whether a program proceeds with the join point.

**The aspect** is the modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. Like a class, an aspect can be instantiated, can contain state and methods, and also may be specialized with sub aspects.

**Introduction or Inter-Type declaration** allows an aspect to add methods, fields or interface to existing classes. It can be public or private. Pointcut Designator is a formula that specifies the sets of join points to which a piece of advice is applicable. A pointcut Designator identifies all types of join points. A pointcut Designator matches certain join points at runtime. An AspectJ program can be divided into two parts:

- *Base code* which includes classes, interface, and standard Java constructs, and
- *Aspect code* which implements the crosscutting concerns in the program.

Since advice and introduction can be regarded as method-line units, we can also represent each piece of advice' or 'introduction' by a method dependence Graph (MDG).

```
    import java.util.*;
    public class TestFactorial{
         private static int n;
1:       public static void main(String[] args){
2:       n=Integer.parseInt(args[0]);
3:       System.out.println("Result:  "+factorial(n)+"\n");
        }

4:      public static long factorial(int n){
        long p;
5:      if(n>0){
6:         p=1;
7:         while(n>0){
8:              p=p*n;
9:              n--;
          }
    }
         else
10:         p=1;
11:   return p;
        }
        }
```

*Base Code*

```
    import java.util.*;
12:   public aspect OptimizeFactorialAspect{
13:    public pointcut factorialOperation(int n):
           call(long TestFactorial.factorial(int)) && args(n);
14:     before(int n): factorialOperation(n){
15:        System.out.println("Seeking factorial for  "+n);
    }
16:    after(int n) returning (long result): factorialOperation(n){
17:       System.out.println("Getting the factorial for  "+n);
        }
        }
```

*Aspect code*

Figure 3.2 : An AspectJ Program

# CHAPTER 4

## DYNAMIC SLICING OF ASPECT ORIENTED PROGRAMS

Basic concepts and Definitions

The Dynamic Aspect Oriented Dependence Graph

Computing Dynamic Slice

A major goal of any dynamic slicing technique is efficiency since the results are normally used during interactive applications such as program debugging. Efficiency is an especially important concern in slicing object-oriented programs, since the size of practical object-oriented programs is often very large. The response time of an inefficient dynamic slicer would be unacceptably large for such programs. We first statically construct an extended Dynamic aspect-oriented system dependence graph (DADG) as the intermediate representation of an object-oriented program. Our algorithm is based on marking and unmarking the edges of the DADG as and when dependencies arise and cease at run-time.

### 4.1 Basic concepts and Definitions

Let var be a variable in a program **P**. In the program **P**, several statements may define the variable var. Let u be a statement that uses the value of the variable var. In the PDG of the program **P**, the node representing the statement u, will have an incoming data dependence edge corresponding from each of the nodes representing the statements where the variable var is defined.

```
1.    main( ) {
      int  m, a, b, i, x, y, z;
2.       cin >> m;
3.       a = 0;
4.       i = 1;
5.       b = 2;
6.       while (i <= m) {
7.             cin >> x;
8.             if (x <= 0)
9.                  y = x + 5;
                   else
10.                 y = x – 5;
11.            z = y + 4;
12.            if (z  > 0)
13.                 a = a + z;
                   else
14.                 b = a + 5;
15.            i = i + 1;
                   }
16.      cout << a;
17.      cout << b;
          }
```

Figure 4.1: An example program

In the example program, the statements 9 and 10 define the same variable y. In the PDG of the example program, node 11 has two incoming edges (9, 11) and (10, 11) corresponding to the definitions of the variable y. Any iteration of the while loop executes exactly one of the statements 9 and 10, and skips the other. Each of the statements 9 and 10 defines the variable y afresh without using its previous value directly or indirectly. Therefore, to get a precise dynamic slice for the slicing criterion < 11,*z* > in any iteration of the while loop, we should consider only the edge (9, 11) or (10, 11) depending on whether the statement 9 or the statement 10 is executed in that iteration.

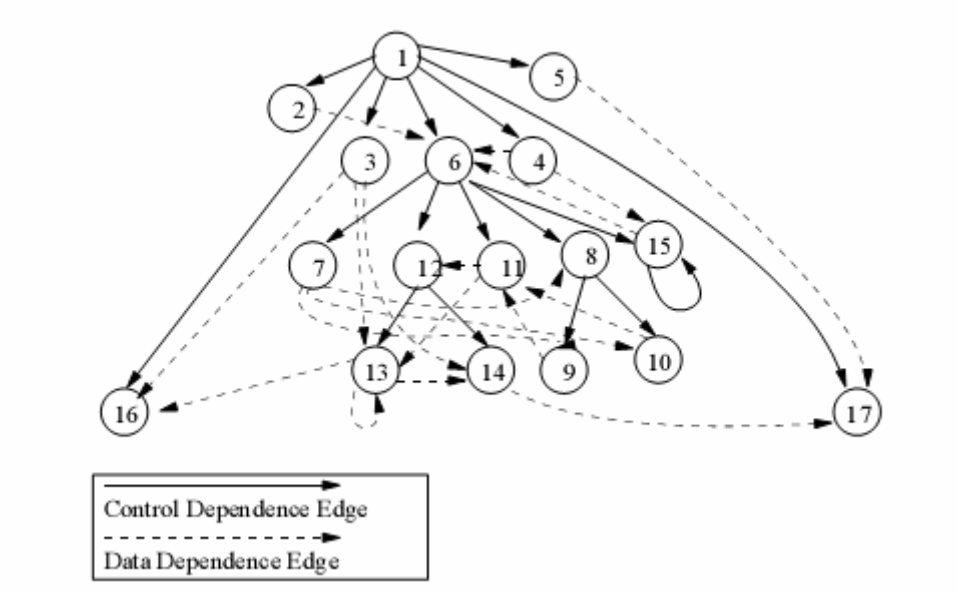


Figure 4.2:The PDG of the example program given in Figure 4.1

**Some Definitions**:

**Definition 4.1 def(*var*):** Let *var* be a variable in a class in the program **P**. A node *u* of the dynamic aspect oriented dependence graph (DADG) of **P** is said to be a def(*var*) node if *u* represents a definition (assignment) statement that defines the variable *var*.

**Definition 4.2 defSet(*var*)**: The set defSet(*var*) denotes the set of all def(*var*) nodes.

**Definition 4.3 use(*var*)**: Let *var* be a variable in a class in the program **P**. A node *u* of the dynamic aspect oriented dependence graph (DADG) of **P** is said to be a use(*var*) node if *u* represents a statement that uses the variable *var*.

**Definition 4.4 useSet(*var*)**: The set useSet(*var*) denotes the set of all use(*var*) nodes.

**Definition 4.5 recentDef(*var*)**: For each variable var in a class, recentDef(*var*)represents the node corresponding to the most recent definition of variable var with respect to some point *s* in an execution

**Definition 4.6 Dynamic Slice(*u, var*)**: Let **G** be the dynamic aspect oriented dependence graph (DADG) of an aspect-oriented program **P**, and u be a node in **G** which corresponds to statement s in program **P**. Let var be a variable defined or used at the statement s. Before execution of the program P, Dynamic Slice(*u*,*var*) = NULL. During execution of the program **P**, Dynamic Slice(*u*,*var*) represents the dynamic slice with respect to variable var for the most recent execution of the statement *s*.

### 4.2 The Dynamic Aspect Oriented Dependence Graph

The DADG is an arc classified digraph (**V,A**),where **V** is the set of vertices that correspond to the statements and predicates of the aspect oriented programs, and **A** is the set of arcs between vertices in **V** representing dynamic dependence relationships that exist between the statements . In DADG of an aspect oriented program, following types of dependence arcs may exist.

- Control dependence arc
- Data dependence arc
- Weaving arc

**Control dependence** represents the control flow relationship of a program, i.e. the control predicates on which a statement or an expression depends during execution

**Data Dependence** represents the relevant data flow relationships of a program, i.e. the flow of data between statements and expressions.

**Weaving arc** reflects the joining of aspect code and non aspect code at join points.

DADG of a complete program is constructed by first creating a partial system dependence graph for the function main and then connecting the partial system dependence graph to called methods in each class dependence graph for each class .To do this, we need to connect call vertices to method entry vertices by call edges, ‘actual-in’ vertices to ‘formal-in’ vertices by parameter-in edges and ‘formal-out’ vertices to ‘actual-

out' vertices by parameter-out edges. The summary edges are added between the actual-in and actual-out vertices at call sites.

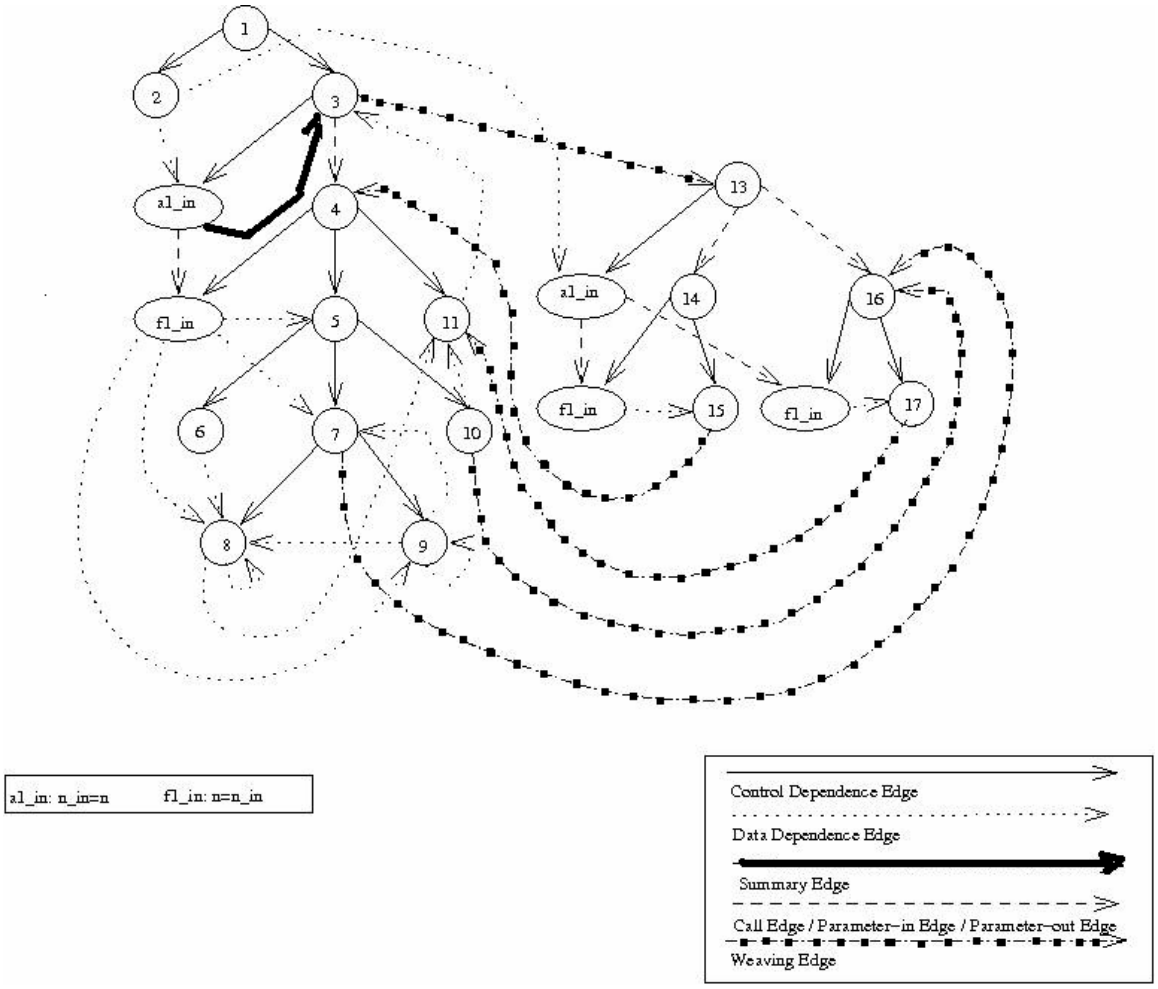


Figure 4.3: DADG of the program given in figure 3.1

For example in figure 4.3 there is a weaving edge from 3 to vertex 13 to connect the vertex 13 to vertex 3 at the corresponding join point because, there is a function call at statement 3 and the corresponding *pointcut* at statement captures that function call. Statement 14 represents a *before* advice .This means that the advice is executed before the control goes to the corresponding function .So, we add a weaving arc from vertex 4 to vertex 15.similarly statement 16 represents a *after* advice. This means that the advice is executed after the function has been executed and before the control goes to the calling function. That's why we add a weaving arc from vertex 16 to vertex 7.After the execution

of *after* advice at statement 17, the control goes to statement 11 where it returns a value to the calling function .So, a weaving arc is added from vertex 11 to vertex 17.Our construction of dynamic aspect oriented dependence graph of an aspect oriented program is based on the dynamic analysis of control flow and data flow of the program. We had denoted the graph in the form of a dependency matrix as shown below:

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 13 | 14 | 15 | 16 | 17 | a1_in | f1_in |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| 3 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 7 | 0 | 0 | 0 |
| 11 | 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 4 | 0 | 1 | 0 |
| 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 15 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a1_in | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 |
| f1_in | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 2 | 0 | 2 | 0 | 0 |

| Code | Edge |
|---|---|
| 0 | No Edge |
| 1 | Control Dependence Edge |
| 2 | Data Dependence Edge |
| 3 | Summary Edge |
| 4 | Call Edge |
| 5 | Parameter-in Edge |
| 6 | Parameter-out Edge |
| 7 | Weaving Edge |

Figure 4.4: DADG of the given example program in a dependency matrix in figure 3.2

### 4.3 Computing Dynamic Slice

Before execution of an object-oriented program **P**, its dynamic aspect-oriented system dependence graph (DADG) is constructed statically. During execution of the program **P**, we *mark* an edge when its associated dependence exists, and *unmark* when its associated dependence ceases to exist. We consider data dependence edges, call edges, parameter-in and parameter-out edges and summary edges for marking and unmarking. To handle

method calls, when a statement invokes a method, we mark the corresponding call edge between the *call vertex* and the method entry vertex. Simultaneously, we mark the corresponding parameter edges between the actual parameter vertices and the formal parameter vertices. Then, we mark the summary edges if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. After the method call is completed and the dynamic slice is recorded at the invoking node, all the marked edges associated with the method entry node are unmarked.

Let the node *u* in DADG correspond to the statement *s* in program P. During execution of P, let *DynamicSlice(u,var)* denote the dynamic slice with respect to variable *var*, for the most recent execution of the statement corresponding to node *u*. Let (*x1, u*) , …, (*xk, u*) be all the marked incoming edges of *u* in the ESDG after an execution of the statement corresponding to node *u*. Then, it is clear that the dynamic slice with respect to variable *var*, for the present execution of the statement corresponding to node *u* is given by:

*DynamicSlice(u,var)* = {*x1,x2,...,xk*} Ụ *DynamicSlice(x1,var)* Ụ *DynamicSlice(x2,var)* Ụ... Ụ *DynamicSlice(xk,var).*

**Steps while computing the dynamic slice**

1. **DADG Construction:** Construct the DADG of the aspect-oriented program P before execution starts.
2. **Initialization:** Do the following before execution of the program P starts
   a) Unmark all the edges of ESDG.
   **b)** Set *DynamicSlice(u,var*) = f for every node u of the ESDG**.**
3. **Runtime Updations:** At run-time, until the program ends or a slicing command is given; carry out the following after each statement s of the program P is executed. Let the node u in DADG correspond to the statement *s*.
   (a) For every variable var used at node *u* do the following:
      i.) unmark the marked dependence edges, if any, associated with the variable var, which may have been marked by the previous execution of the node *u*.
      ii.) mark the data dependence edge (*x,u*) where *x* = recentDef (*var*).

//Node *u* uses the value computed by the most recent definition of variable var. Mark the dependence edge corresponding to the present execution of the node *u*.

(b)Mark an edge when its associated dependence exists, and unmark when its associated dependence ceases to exist

Update *DynamicSlice(u, var)* to

*DynamicSlice(u,var)* = {*x1,x2,...,xk*} Ụ *DynamicSlice(x1,var)* Ụ *DynamicSlice(x2,var)* Ụ... Ụ *DynamicSlice(xk,var),*where *x1,x2,...,xk* are the initial vertices of the corresponding marked incoming edges of *u*

**4. Slice Look Up:**

If a slicing command < *s,var* > is given, look up Dynamic Slice(*u,var*) for variable var for the content of the slice**.**

// node *u* corresponds to statement *s*.

**Working of the Algorithm**

We illustrate the working of the algorithm with the help of the example program given in figure 3.1. The DADG corresponding to the AspectJ program is given in figure 4.2. During the initialization stage the algorithm first unmarks all the edges of the DADG and sets *dslice(u)=Ø* for every node u of the DADG .Now for the input data '4' the program will execute the statements 1, 2, 3, 13, 14, 15, 4, 5, 6, 7, 8, 9, 7, 8, 9, 7, 8, 9, 7, 8, 9, 7, 16, 17, 11, 3 in order. So the algorithm marks the edges (1,2), (1,3), (3,4), (3,13), (13,14), (13,16), (14,15), (15,4), (4,5), (5,6), (5,7), (7,8), (7,9), (7,16), (16,17), (17,11), (11,3).Also the corresponding parameter edges are marked. During runtime the dynamic slice for each statement is computed immediately after the execution of the statement. If the slicing criterion is <3,p> the dynamic slice is given by *dslice(3).*

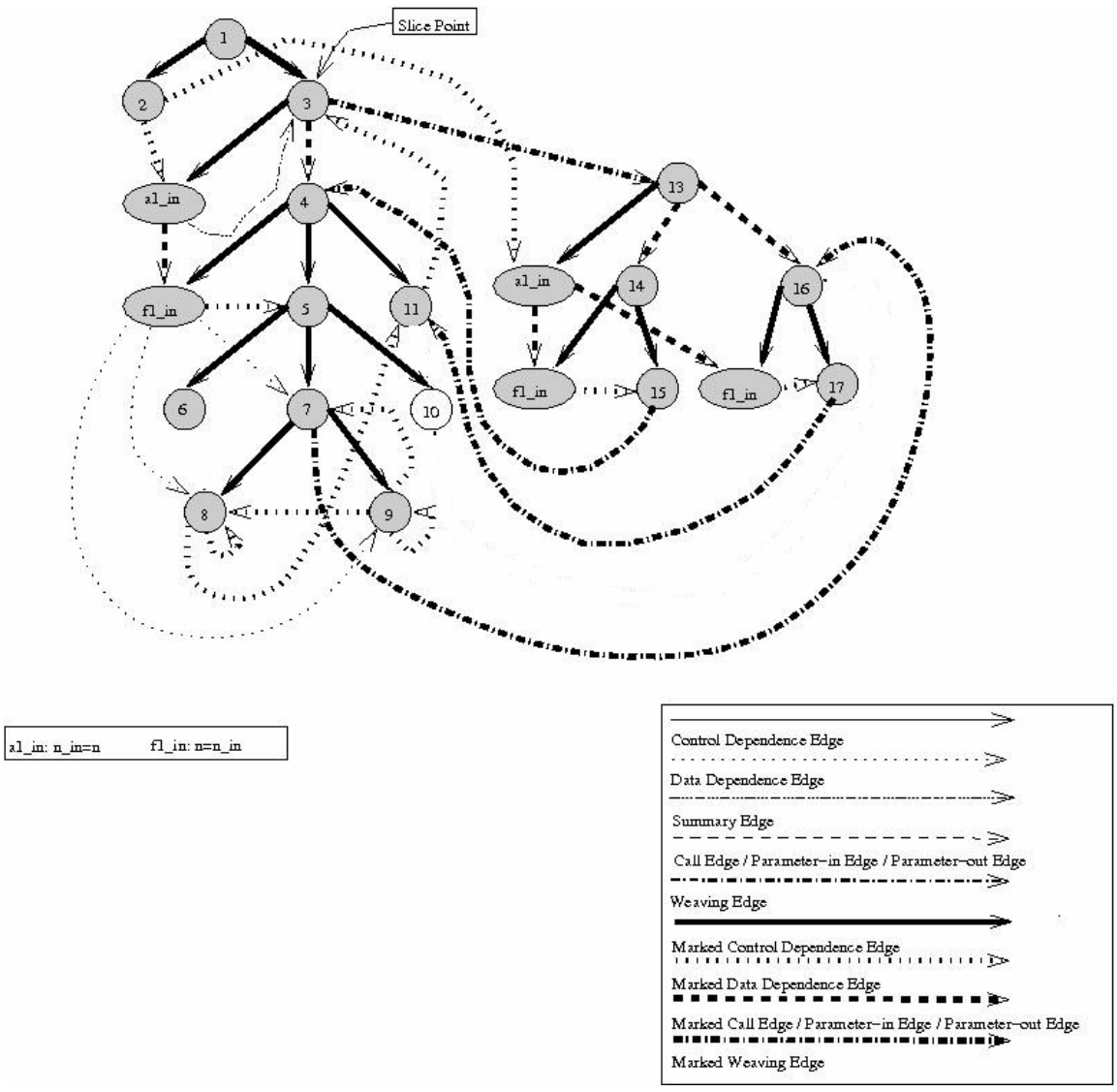


Figure 4.5: The updated DADG after applying slicing algorithm with respect to slicing criterion <3,p>

The Time required to compute the slice for a given statement in a program increases with the increase in the number of lines in the program.

| SL NO | Program Size (Number of lines) | Time taken to compute slice in seconds |
|---|---|---|
| 1 | 17 | 0.11 |
| 2 | 28 | 0.13 |
| 3 | 41 | 0.15 |

Fig 4.6: Table for time required for computation of dynamic slice

# CHAPTER 5

## CONCLUSION

**Conclusion:**

The primary aim of our work was to develop efficient dynamic slicing algorithms for aspect-oriented programs. We first developed an intermediate representation for representing simple aspect oriented programs. We statically constructed the DADG of the given aspect oriented program. Then we computed the dynamic slice for a given slicing criterion. We wrote a C++ program which computed the Dynamic Aspect oriented system dependence graph (DADG) .The DADG is an arc classified digraph which represents various dependencies between the statements of an aspect oriented program for a particular execution. Then we found out the dynamic slice of the Aspect oriented Program by marking and unmarking the edges of the DADG. The time to compute the slice of a statement in a program increased with the number of lines in the program.

# CHAPTER 6

# REFERENCES

**Reference:**


[1]. MOHAPATRA D. P. ***Dynamic slicing of object oriented programs***, **PhD paper ,IIT Kharagpur 2005**

[2]. BINKLEY, D., and GALLAGHER, K. B. ***Program Slicing, Advances in Computers,* vol. 43. Academic Press , San Diego, CA, 1996.**

[3]. JIANJUN ZHAO. ***Slicing Aspect oriented software*** ,**Fukuoka Institute of Technology**

[4]. AGRAWAL, H., and HORGAN, J. **Dynamic program slicing**. ***In Proceedings of the ACM SIGPLAN'90 Conference on Programmimg Lanuages Design and Implementation, SIGPLAN Notices, Analysis and Verification*** **(White Plains, NewYork, 1990),vol. 25, pp. 246–256.**

[5]. Wikipedia. http://www.wikipedia.com

[6]. AspectJ. http://www.eclipse.org/aspectj