Dynamic Slicing of Aspect-Oriented Programs

Durga Prasad Mohapatra
Department of CSE
National Institute of Technology
Rourkela-769008, India
E-mail: durga@nitrkl.ac.in

Madhusmita Sahu
Department of MCA
C V Raman Computer Academy
Bhubaneswar-752054,India
E-mail: madhu_sahu@yahoo.com

Rajeev Kumar and Rajib Mall
Department of CSE
Indian Institute of Technology
Kharagpur-721302, India
E-mail: {rkumar, rajib}@cse.iitkgp.ernet.in

Keywords: program slice, aspect-oriented programming, AspectJ, dynamic aspect-oriented dependence graph (DADG), dynamic dependence slicing tool (DDST), trace file based dynamic slicing (TBDS) algorithm

Received: July 2, 2007

Program slicing is a decomposition technique which has many applications in various software engineering activities such as program debugging, testing, maintenance etc. Aspect-oriented programming (AOP) is a new programming paradigm that enables modular implementation of cross-cutting concerns such as exception handling, security, synchronization, logging etc. The unique features of AOP such as join-point, advice, aspect, introduction etc. pose difficulties for slicing of AOPs. We propose a dynamic slicing algorithm for aspect-oriented programs. Our algorithm uses a dependence-based representation called Dynamic Aspect-Oriented Dependence Graph (DADG) as the intermediate program representation. The DADG is an arc-classified digraph which represents various dynamic dependences between the statements of the aspect-oriented program. We have used a trace file to store the execution history of the program. We have developed a tool called Dynamic Depenedence Slicing Tool (DDST) to implement our algorithm. We have tested our algorithm on many programs for 40-50 runs. The resulting dynamic slice is precise as we create a node in the DADG for each occurrence of the statement in the execution trace.

Povzetek: Opisana je modularna gradnja objektno orientiranih programov.

1 Introduction

The concept of a program slice was first introduced by Weiser [26]. Program slicing [38] is a decomposition technique which extracts program statements related to a particular computation from a program. A program slice is constructed with respect to a slicing criterion. A *slicing criterion* is a tuple $< s, v >$ where s is a statement in a program and v is a variable used or defined at s. A program slice can be *static* or *dynamic*. A *static* slice consists of all statements of a program that might affect the value of a variable at a program point of interest for *every possible* inputs to the program. In contrast, a *dynamic* slice consists of only those statements that actually affect the value of a variable at a program point of interest for a *particular* set of inputs to the program.

Aspect-oriented programming (AOP) is a new programming paradigm that enables modular implementation of cross-cutting concerns [17] such as exception handling, security, synchronization, logging. This concept was proposed by Gregor Kiczales et al. [16]. Expressing such cross-cutting concerns using standard language constructs produces poorly structured code since these concerns are tangled with the basic functionality of the code. This increases the system complexity and makes maintenance considerably more difficult.

AOP [2, 3] attempts to solve this problem by allowing the programmer to develop cross-cutting concerns as full stand-alone modules called *aspects*. The main idea behind AOP is to construct a program by describing each concern

separately.

Aspect-oriented programming languages present unique opportunities and problems for program analysis schemes. For example, to perform program slicing on aspect-oriented software, specific aspect-oriented features such as *join-point*, *advice*, *aspect*, *introduction* must be handled appropriately. Although these features provide great strengths to model the cross-cutting concerns in an aspect-oriented program, they introduce difficulties to analyze the program.

A major aim of any slicing technique is to realize as small a slice with respect to a slicing criterion as possible since smaller slices are found to be more useful for different applications. Much of the literature on program slicing is concerned with improving the algorithms for slicing in terms of reducing the size of the slice and improving the efficiency of the slicing algorithm. Now-a-days, many programs are aspect-oriented. These aspect-oriented programs are quite large and complex. It is much difficult to debug and test these products. Program slicing techniques have been found to be useful in applications such as program understanding, debugging, testing, software maintenance and reverse engineering etc. [12, 27, 29, 31]. Particularly dynamic program slicing is used in interactive applications such as debugging and testing of programs. Therefore the dynamic slicing techniques need to be efficient. This requires to develop efficient slicing algorithms as well as suitable intermediate representations for aspect-oriented programs. Researchers have developed many representations for procedural and object-oriented programs [11, 21, 24, 27, 28, 29, 33, 39], but very few work has been carried out for representation of aspect-oriented programs [22, 25]. Due to the specific features of aspect-oriented programming language, existing slicing algorithms for procedural or object-oriented programming languages cannot be applied directly to aspect-oriented programs. Therefore, there is a pressing necessity to devise suitable intermediate representations and efficient algorithms for dynamic slicing of aspect-oriented programs.

With this motivation for developing techniques for dynamic slicing of aspect-oriented programs, we identify the following objective. The main objective of our research work is to develop an efficient dynamic slicing algorithm. To address this broad objective, we identify the following goals:

- to develop a suitable intermediate representation for aspect-oriented programs on which the slicing algorithm can be applied.
- to develop a dynamic slicing algorithm for aspect-oriented programs, using the proposed intermediate representation.

In this paper, we propose a new intermediate representation for aspect-oriented programs. We call this representation as **Dynamic Aspect-Oriented Dependence Graph (DADG)**. Then, we propose a dynamic slicing algorithm for aspect-oriented programs. We have used a trace file to store the execution history of the source code. So, we have named our algorithm *Trace file Based Dynamic Slicing* (TBDS) algorithm. Our algorithm computes *precise* dynamic slices as we create a node in the DADG for each occurrence of the statement in the execution trace.

The rest of the paper is organized as follows. In Section 2, we discuss some related works. In Section 3, we present a brief introduction to Aspect-Oriented Programming (AOP). In Section 4, we describe some notions of dynamic slices of aspect-oriented programs. Section 5 discusses the dynamic aspect-oriented dependence graph (DADG) for aspect-oriented programs and also describes the construction of DADG. In Section 6, we discuss the computation of dynamic slices of aspect-oriented programs using DADG. In Section 7, we present the implementation details of our work. Section 8 concludes the paper.

2 Related work

Horwitz et al. [33] developed a system dependence graph (SDG) as an intermediate program representation for procedural programs with multiple procedures. They proposed a two-phase graph reachability algorithm on the SDG to compute inter-procedural slice. The slice consists of the union of vertices marked in both the phases.

Later, Larsen and Harrold [24] extended the SDG of Horwitz et al. [33] to represent object-oriented programs. Their [24] extended SDG incorporates many object-oriented features such as classes, objects, inheritance, polymorphism etc. After constructing the SDG, Larsen and Harrold [24] used the two-phase algorithm to compute the static slice of an object-oriented program. Later, Liang and Harrold [11] developed a more efficient intermediate representation of object-oriented programs which is an extension to the SDG of Larsen and Harrold [24]. Their [11] SDG represents objects that are used as parameters or data members in other objects, the effects of polymorphism on parameters and parameter bindings. The data members for different objects can be distinguished using this approach. Later many researchers have extended the work on static slicing of object-oriented programs [11, 39]. But they [11, 39] have not considered the aspect-oriented features.

Also dynamic slicing of OOPs have been addressed by several researchers [21, 27, 28, 29]. Korel and Laski [7] introduced the concept of dynamic program slicing. Agrawal and Horgan [19] presented the first algorithm for finding dynamic slices of procedural programs using dependence graphs.

Zhao [21] extended the *dynamic dependence graph* (DDG) of Agarwal and Horgan [18] for the representation of various dynamic dependences between statement instances for a particular execution of an object-oriented program. Zhao [21] named this graph *dynamic object-oriented dependence graph* (DODG). He used a two-phase algorithm on the DODG for the computation of dynamic

slices of object-oriented programs.

Song et al. [36] proposed a method to compute forward dynamic slices of object-oriented programs using *dynamic object relationship diagram* (DORD). They computed the dynamic slices for each statement immediately after the statement is executed. The dynamic slices of all executed statements have been obtained after the execution of the last statement.

Xu et al. [9] extended their earlier work [39] to dynamically slice object-oriented programs. Their method uses *object program dependence graph* (OPDG) and other static information to reduce the information to be traced during execution and computes dynamic slices combining static dependence information and dynamic execution of the program.

Wang et al. [37] proposed a new algorithm for dynamic slicing of Java programs which operates on compressed bytecode traces. According to their approach, first, the bytecode stream corresponding to an execution trace of a Java program is compactly represented. Then, a backward traversal of the compressed program trace is performed to compute data/control dependences on-the-fly. The slice is updated as these dependences are encountered during trace traversal.

Mohapatra et al. [29, 30] have developed edge-marking and node-marking dynamic slicing techniques for object-oriented programs. Their algorithms are based on marking and unmarking the edges (nodes) of the graph appropriately, as and when dependences arise and cease. Many researchers [8, 21, 28, 31] have extended the work on dynamic slicing of object-oriented programs. But, none of the researchers [8, 21, 28, 29, 30, 31] have considered the aspect-oriented features.

Zhao [22] was the first to develop the aspect-oriented system dependence graph (ASDG) to represent aspect-oriented programs. The ASDG is constructed by combining the SDG for non-aspect code, the aspect dependence graph (ADG) for aspect code and some additional dependence arcs used to connect the SDG and ADG. Then, Zhao [22] used the two-phase slicing algorithm proposed by Larsen and Harrold [24] to compute static slice of aspect-oriented programs.

Braak [34] extended the ASDG proposed by Zhao [22, 23] to include inter-type declarations in the graph. Each inter-type declaration was represented in the form of a field or a method as a successor of the particular class. Then, Braak [34] used the two-phase slicing algorithm of Horwitz et al. [33] to find the static slice of an aspect-oriented program. Braak [34] has not addressed the dynamic slicing aspects.

We have proposed an approach for computation of dynamic slice using a dependence graph based intermediate representation called *dynamic aspect-oriented dependence graph* (DADG). We have used a trace file to store the execution history of the aspect-oriented program. Our DADG correctly represents the aspect-oriented features such as pointcuts, advices etc. Also, weaving process is correctly represented in the DADG. The TBDS algorithm computes *precise* dynamic slices as we create separate vertices in the DADG for each occurrence of the statement in the execution trace.

3 Aspect-oriented programming

In this section, we first discuss the basic concepts of aspect-oriented programming. Then, we briefly describe AspectJ: an aspect-oriented programming language. Next, we present some features of AspectJ.

3.1 Basic concepts

Gregor Kiczales et al. [16] introduced the concept of Aspect-Oriented Programming (AOP) at Xerox Palo Alto Research Center (PARC) in 1996. 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 a system, which has been implemented in a code structure [4, 6, 17, 35]. Examples of aspects are data storage, user interface, platform-specific code, security, distribution, logging, class structure, threading etc.

The strength of aspect-oriented programming is the enabling of a better separation of concerns, by allowing the programmer to create cross-cutting concerns as program modules. *Cross-cutting concerns* are those parts, or aspects, of the program that are scattered across multiple program modules, and tangled with other modules in standard design.

```
void transfer(Account fromAccount, Account toAccount, int amount){
  if (!getCurrentUser().canPerform(OP_TRANSFER)){
    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_TRANSFER,fromAccount,toAccount,amount);
  }
  catch(Exception e){
    tx.rollback();
  }
}
```

Figure 1: An example program

Let us consider the example program given in Figure 1. The objective of this program is to transfer an amount from

one account to another in a banking application. In this example, various cross-cutting concerns such as transactions, security, logging etc. are tangled with the basic functionality (sometimes called as the *business logic concern*). If there is a need to change the security considerations for the application, then it would require a major effort since security-related operations appear scattered across numerous methods. This means that the cross-cutting concerns do not get properly encapsulated in their 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 a system's behavior as separately as possible. Although the hierarchical modularity of object-oriented languages is extremely useful, they are inherently unable to modularize cross-cutting concerns in complex systems. Aspect-oriented programming provides language mechanisms to explicitly capture the cross-cutting structure.

To better support the expression of cross-cutting design decisions, AOP uses a component language to describe the basic functionality of the system and aspect languages to describe the different cross-cutting properties. The components and the aspects are then combined into a system using an *aspect weaver* [10]. The *aspect weaver* makes it possible for an *advice* to be activated at appropriate *join points* during run-time. Thus, a source code is modified by inserting aspect-specific statements at join points.

3.2 AspectJ: an aspect-oriented programming language

Several different Aspect-oriented programming systems have been built, including AML (Aspect Markup Language), an environment for sparse matrix computation [20], RG (Reverse Graphics), an environment for creating image processing systems [5] etc. The most popular AOP language is AspectJ. An AspectJ program is divided into two parts: base code or non-aspect code and aspect code. The *base code* includes classes, interfaces and other standard Java constructs. The *aspect code* implements the cross-cutting concerns in the program. Other aspect-oriented frameworks include COOL (COOrdination Language) for expressing synchronization concerns [13], RIDL (Remote Invocation Data transfer Language) for expressing distribution concerns [13], JBOSS, Spring AOP, AspectWerkz [10, 15] etc.

AspectJ was created by Chris Maeda [16] at Xerox Palo Alto Research Center (PARC). This is an aspect-oriented extension to Java programming language. In other words, we can say that AspectJ is compatible with current Java platform [14]. There are four types of compatibility:

- *Upward compatibility*- all legal Java programs must be legal AspectJ programs.
- *Platform compatibility*- all legal AspectJ programs must run on standard Java virtual machines.
- *Tool 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. These features include *join points*, *pointcut*, *advice*, *aspect*, *introduction* or *inter-type declaration*. We explain these features below.

- *Join Points*- These are well-defined points in the execution of a program, such as, method call (a point where method is called), method execution (a point where method is invoked) and method reception join points (a point where a method received a call, but this method is not executed yet).
- *Pointcut*- This is a means of referring to collections of join points and certain values at those join points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. For example, in Figure 2, the pointcut *factorialOperation* at statement 13 picks out join points i.e. the pointcut *factorialOperation* picks out each call to the method *factorial()* of an instance of the class *TestFactorial*, where an *int* is being passed as an argument and it makes the value of that argument to be available to the enclosing advice or pointcut.
- *Advice*- It is a method-like construct which is used to define cross-cutting behavior at join points. This is used to define some code that is executed when a pointcut is reached. Advice brings together a pointcut (to pick out join points) and a body of code (to run at each of those join points). There are three types of advice in AspectJ: *after*, *before*, *around*.
  - (i) *After- After advice* on a particular join point runs after the program proceeds with that join point. For example, *after* advice on a method call join point runs after the method body has run, just before control is returned to the caller. For example, in Figure 2, the *after* advice at statement 16 runs just after each join point picked out by the pointcut *factorialOperation* and before the control is returned to the calling method.
  - (ii) *Before- Before* advice runs as a join point is reached, before the program proceeds with the join point. For example, *before advice* on a method call join point runs before the actual method starts running, just after the arguments to the method call are evaluated. For example, in Figure 2, the *before* advice at statement 14 runs just before the join points picked out by the pointcut *factorialOperation*.

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

Additionally, there are two special cases of after advice: *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point.

(i) *After returning- After returning* advice runs just after each join point picked out by the pointcut, but only if it returns normally. The return value can be accessed. After the advice runs, the return value is returned. For example, in Figure 2, the *after returning* advice at statement 16 runs just after each join point picked out by the pointcut *factorialOperation*, but only if it returns normally. The return value can be accessed and it is named *result* in Figure 2 at statement 16. After the advice runs, the return value is returned.

(ii) *After throwing- After throwing* advice runs just after each join point picked out by the pointcut, but only when it throws an exception. The advice re-raises the exception after it is done.

– *Aspect-* These are units of modular cross-cutting implementations composed of pointcuts, advices, and ordinary JAVA member declarations. An *aspect* is a cross-cutting type, defined by the aspect declaration. Aspects are defined by *aspect* declarations, which have a similar form of class declarations. For example, in Figure 2, there is one aspect named *OptimizeFactorialAspect* at statement 12.

– *Introduction* or *Inter-Type Declaration-* It allows an aspect to add methods, fields or interfaces to existing classes. It can be *public* or *private*. An introduction declared as *private* can be referred or accessed *only* by the code in the aspect that declared it. An introduction declared as *public* can be accessed by *any* code.

– *Pointcut Designator-* It is a formula that specifies the set of join points to which a piece of advice is applicable. A pointcut designator identifies all types of join points. A pointcut designator simply matches certain join points at runtime. For example, in Figure 2, the pointcut designator

call (long TestFactorial.factorial(int))

at statement 13 matches all method calls to *factorial* from an instance of the class *TestFactorial*.

Pointcuts can be combined using logical operators *and* (&&), *or* (||) and *not* (!). For example, in Figure 2, the compound pointcut designator

call (long TestFactorial.factorial(int)) && args(n)

at statement 13 refers to all method calls to *factorial()* of an instance of *TestFactorial*, where the argument of type *int* is passed to the method *factorial()*.

User-defined pointcut designators are defined with *pointcut* declaration. For example, in Figure 2, the declaration

public pointcut factorialOperation(int n):
call (long TestFactorial.factorial(int)) && args(n)

at statement 13 defines a new pointcut designator, *factorialOperation*, that specifies a call to the method *factorial()* of an instance of *TestFactorial* and the argument passed to the method to be of type *int*.

For example, Figure 2 shows an AspectJ program for finding the factorial of a number. The program is divided into two parts: the base code or non-aspect code contains the class *TestFactorial* and the aspect code *OptimizeFactorialAspect* contains the advices and pointcuts. Any AspectJ implementation ensures that both the codes i.e., aspect code and base code run together in a properly coordinated fashion. Such type of process is called *aspect weaving*. The key component for this process is *aspect-weaver* which makes the applicable advices to run at the appropriate join points.

4 Dynamic slicing of aspect-oriented programs

In this section, we present the basic concepts and the definitions which will be used in our algorithm.

Definition 1 (Digraph): A *digraph* is an ordered pair (V, A), where V is a finite set of elements called *vertices* and A is a finite set of elements called *edges* and $A \subseteq V \times V$.

Definition 2 (Arc-classified digraph): An *arc-classified digraph* is an n-tuple $(V, A_1, A_2, \ldots, A_{n-1})$ such that every (V, A_i), $(i = 1, 2, \ldots, n-1)$ is a digraph and $A_i \cap A_j = \emptyset$ for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$ and $i \neq j$.

Definition 3 (Path): A path from vertex u to vertex v in a digraph (V, A) is a sequence of vertices $u, i_1, i_2, \ldots, i_k, v$ in V such that $(u, i_1), (i_1, i_2), \ldots, (i_k, v)$ are edges in A.

Definition 4 (Flow graph): The *flow graph* of an aspect-oriented program is a quadruple $(V, A, Start, Stop)$ where (V, A) is a digraph, *Start* $\in V$ is a distinguished node of in-degree 0 called the *start node*, *Stop* $\in V$ is a distinguished node of out-degree 0 called the *stop node*, there is a path from *Start* to every other node in the graph, and there is a path from every other node in the graph to *Stop*.

Definition 5 (Control flow graph(CFG)): Let the set

```
    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;
       }
       }
```

Non−aspect Code (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 2: An example AspectJ program

```
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 3: An example program

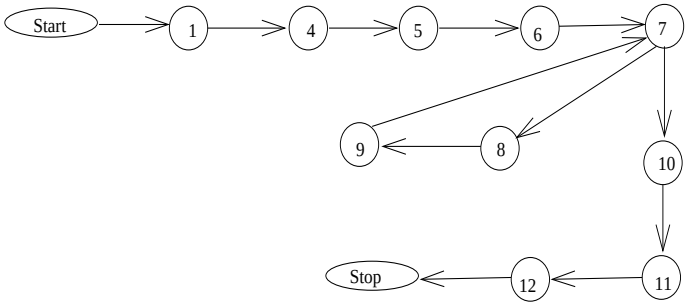

Figure 4: The CFG of the example program given in Figure 3

V represent the set of statements of a program P. The *control flow graph* of the program P is the flow graph G = $(V_1, A, Start, Stop)$ where $V_1 = V \cup \{Start, Stop\}$. An edge $(m, n) \in A$ indicates the possible flow of control from the node m to the node n. Note that the existence of an edge (x, y) in the control flow graph means that control *must* transfer from x to y during program execution. Figure 4 represents the CFG of the example program given in Figure 3.

Definition 6 (Dominance): If x and y are two nodes in a flow graph G then x dominates y iff every path from *Start* to y passes through x. y post-dominates x iff every path from x to *Stop* passes through y.

Let x and y be nodes in a flow graph G. Node x is said to be immediate post-dominator of node y iff x is a post-dominator of y, $x \neq y$ and each post-dominator $z \neq x$ of y post-dominates x. The post-dominator tree of a flow graph G is the tree that consists of the nodes of G, has the root Stop, and has an edge (x, y) iff x is the immediate post-dominator of y.

Consider the flow graph of the example program of Figure 3, which is given in Figure 4. In the flow graph, each of the nodes 4, 5 and 6 dominates 7. Node 8 does not dominate node 10. Node 10 post dominates each of the nodes 4, 5, 6, 7, 8 and 9. Node 9 post dominates node 8. Node 9 post dominates none of the nodes 4, 5, 6, 7, 10, 11 and 12. Node 6 is the immediate post dominator of node 5. Node 10 is the immediate post dominator of node 7.

Definition 7 (Execution trace): An *execution trace* is a path that has actually been executed for some input data.

For example, for the input data $argv[0]$ = 4, the order of execution of the statements of the program given in Figure 2 is 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. This *execution trace* is given in Figure 5.

Definition 8 (Def(var)): Let var be a variable in a class in

the program P. A vertex u of the DADG of P is said to be a *Def(var)* vertex if u represents a definition (assignment) statement that defines the variable var.

In the DADG given in Figure 6, vertices 6 and 8 are the *Def(p)* vertices.

Definition 9 (DefSet(var)): The set *DefSet(var)* denotes the set of all *Def(var)* vertices.

In the DADG given in Figure 6, *DefSet(p)*={6, 8}.

Definition 10 (Use(var)): Let var be a variable in a class in the program P. A vertex u of the DADG of P is said to be a *Use(var)* vertex if u represents a statement that uses the variable var.

In the DADG given in Figure 6, vertices 8 and 11 are the *Use(p)* vertices.

Definition 11 (UseSet(var)): The set *UseSet(var)* denotes the set of all *Use(var)* vertices.

In the DADG given in Figure 6, *UseSet(p)*={8, 11}.

5 The dynamic aspect-oriented dependence graph (DADG)

In this section, we describe the definition and construction of the dynamic aspect-oriented dependence graph (DADG).

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 statements. In the DADG of an aspect-oriented program, following types of dependence arcs may exist.

- control dependence arc
- data dependence arc
- weaving arc

Control dependences represent the control flow relationships of a program i.e., the predicates on which a statement or an expression depends during execution.

Data dependences represent the relevant data flow relationships of a program i.e., the flow of data between statements and expressions.

For example, in the DADG given in Figure 6, there is a data dependency between vertex 6 and 8, because vertex 6 is *Def(p)* vertex and vertex 8 is *Use(p)* vertex.

Weaving arcs reflect the joining of aspect code and non-aspect code at appropriate join points.

For example, in Figure 6 there is an weaving arc from vertex 13 to vertex 3 to connect 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 13 captures that function call. Statement 14 represents a *before* advice. This means that the *advice* is executed before control flows to the corresponding function i.e., to the function *factorial()*. So, we add a *weaving arc* from vertex 4 to vertex 15. Similarly, statement 16 represents an *after* advice. This means that the *advice* is executed after the function *factorial()* has been executed and before control flows to the calling function i.e., the function *main()*. That's why we add a weaving arc from vertex 16 to vertex 7. After the execution of *after* advice at statement 17, control transfers to statement 11 where it returns a value i.e., the value of p to the calling function *main()*. 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 dynamic analysis of control flow and data flow of the program. The DADG of the program in Figure 2 corresponding to the execution trace in Figure 5 is given in Figure 6. In this figure, circles represent program statements, dotted lines represent data dependence arcs, solid lines represent control dependence arcs and dark dashed lines represent weaving arcs.

6 Computation of dynamic slices of aspect-oriented programs

Dynamic slicing of aspect-oriented programs is similar to that of object-oriented programs. However, due to the presence of pointcuts and advices, the tracing of dependences becomes much more complex.

Here, we formally define some notions of dynamic slicing of aspect-oriented programs. Let P be an aspect-oriented program and $G = (V, A)$ be the DADG of P. We compute the dynamic slice of an aspect-oriented program with respect to a slicing criterion.

- A *slicing criterion* for an aspect-oriented program is of the form $< p, q, e, n >$, where p is a statement, q is a variable used at p and e is an execution trace of the program with input n.
- A *dynamic slice* of an aspect-oriented program for a given slicing criterion $< p, q, e, n >$ consists of all the statements that have actually affected the value of the variable q at statement p.

Let DS_G be the dynamic slice of G on a given slicing criterion $< p, q, e, n >$. Then, DS_G is a subset of vertices of G i.e., $DS_G(p, q, e, n) \subseteq V$, such that for any $p' \in V$, $p' \in DS_G(p, q, e, n)$ if and only if there exists a path from p' to p in G. Since we have used a trace file to store the execution history of the aspect-oriented program, we have named our algorithm *Trace file Based Dynamic Slicing* (TBDS) algorithm for AOPs.

In this section, we present the TBDS algorithm in pseudo-code form to compute the dynamic slice of an aspect-oriented program.

Algorithm: TBDS algorithm

1. Creation of execution trace file: To create an execution trace file, do the following:

```
          public class TestFactorial
          private static int n;
1(1):     public static void main(String[ ] args)
2(1):     n=Integer.parseInt(args[0]);
3(1):     System.out.println("Result:  "+factorial(n)+"\n");
13(1):    public pointcut factorialOperation(int n): call(long TestFactorial.factorial(int)) && args(n);
14(1):    before(int n): factorialOperation(n)
15(1):    System.out.println("Seeking factorial for  "+n);
4(1):     public static long factorial(int n)
5(1):     if(n>0)
6(1):     p=1;
7(1):     while(n>0)
8(1):     p=p*n;
9(1):     n--;
7(2):     while(n>0)
8(2):     p=p*n;
9(2):     n--;
7(3):     while(n>0)
8(3):     p=p*n;
9(3):     n--;
7(4):     while(n>0)
8(4):     p=p*n;
9(4):     n--;
7(5):     while(n>0)
16(1):    after(int n) returning(long result): factorialOperation(n)
17(1):    System.out.println("Getting the factorial for  "+n);
11(1):    return p;
```

Figure 5: Execution trace of the program given in Figure 2 for argv[0] = 4

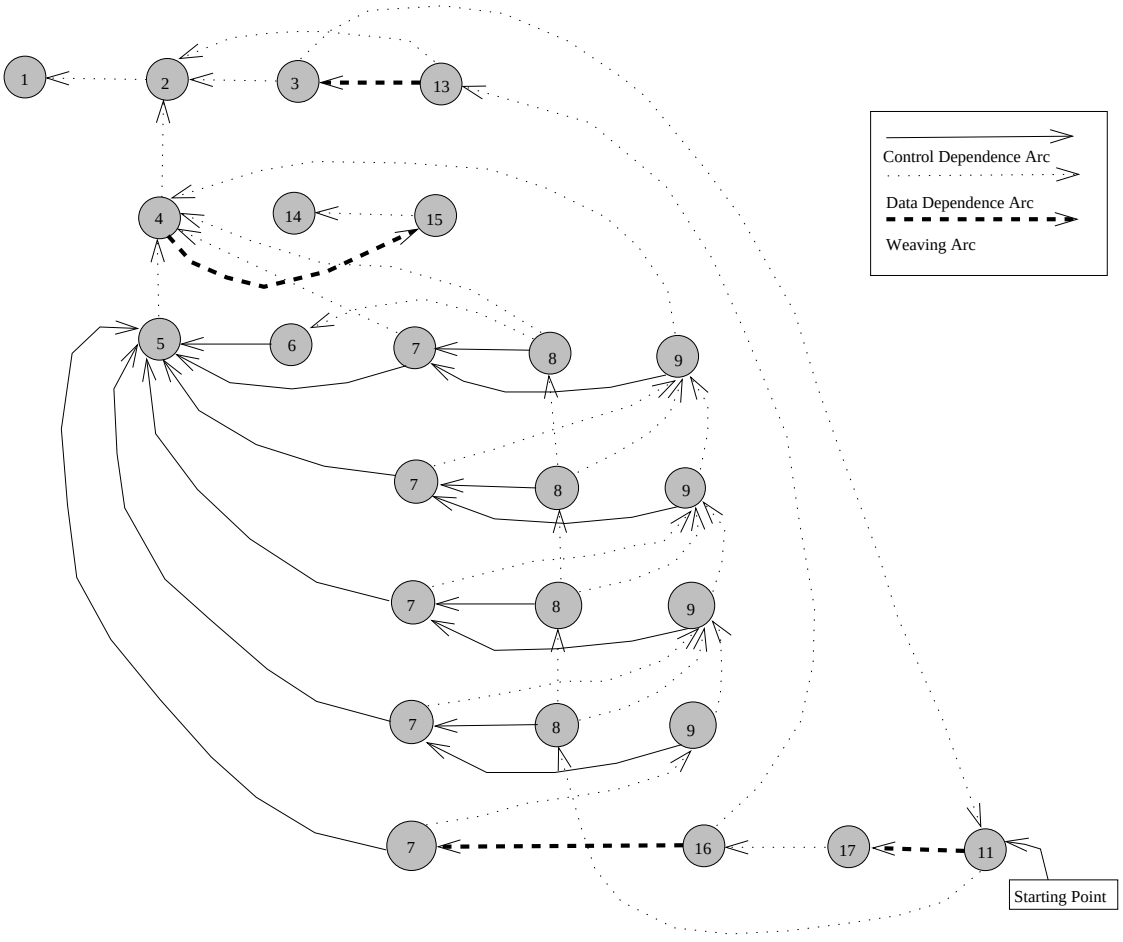

Figure 6: Dynamic aspect-oriented dependence graph for the execution trace given in Figure 5

(a) For a given input, execute the program and store each statement s in the order of execution in a file after it has been executed.

(b) If the program contains loops, then store each statement s inside the loop in the trace file after each time it has been executed.

2. Construction of DADG: To Construct the DADG of the aspect-oriented program P with respect to the trace file, do the following:

   (a) For each statement s in the trace file, create a vertex in the DADG.

   (b) For each occurrence of a statement s in the trace file, create a separate vertex.

   (c) Add all control dependence edges, data dependence edges and weaving edges to these vertices.

3. Computation of dynamic slice: To compute the dynamic slice over the DADG, do the following:

   (a) Perform the breadth-first or depth-first graph traversal over the DADG taking any vertex corresponding to the statement of interest as the starting point of traversal.

4. Mapping of the slice: To obtain the dynamic slice of the aspect-oriented program P, do the following:

   (a) Define a mapping function $f : DS_G(p, q, e, n) \rightarrow P$.

   (b) Map the resulting slice obtained in Step 3(a) over the DADG to the source code P using f since the slice may contain multiple occurrences of the same vertex.

Working of the algorithm: We illustrate the working of the TBDS algorithm with the help of an example. Consider the example AspectJ program given in Figure 2. Now, for the input data $argv[0]$ = 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 in order. These statements are stored in a trace file. Figure 5 shows the corresponding execution trace file. Then, the *Dynamic Aspect-Oriented Dependence Graph* (DADG) is constructed with respect to this trace file in accordance with the step 2 of the TBDS algorithm. Figure 6 shows the DADG of the example program given in Figure 2 with respect to the trace file given in Figure 5. Since, for the input data $argv[0]$ = 4, the statements 8 and 9 are executed four times and statement 7 is executed five times, separate vertices are created for each occurrence of these statements.

Now, let us suppose that we have to compute the dynamic slice for the slicing criterion $< 11, p >$. Starting from the vertex 11, we can perform either the breadth-first search algorithm or depth-first search algorithm on the DADG. The breadth-first search algorithm yields the vertices 11, 17, 8, 16, 7, 8, 9, 7, 13, 5, 9, 7, 8, 9, 9, 3, 2, 4, 7, 8, 9, 1, 15, 7, 6, 14 and the depth-first search algorithm yields the vertices 11, 8, 9, 9, 9, 4, 15, 14, 2, 1, 7, 5, 7, 7, 8, 8, 8, 6, 7, 17, 16, 13, 3, 7, 9. The traversed vertices are shown as shaded vertices in Figure 6. Using the mapping function f, we can find the statements corresponding to these vertices. This gives us the required dynamic slice which is shown in rectangular boxes in Figure 7.

6.1 Complexity analysis

In the following, we discuss the space and time complexity of our DADG algorithm.

Space complexity: Let P be an aspect-oriented program and S be the length of execution of P. Each executed statement will be represented by a single vertex in the DADG. Thus, it can be stated that there are S number of vertices in the DADG corresponding to all executed statements of program P. Also, S numbers of statements are stored in the execution trace file. So, the space complexity of the trace file based algorithm is $O(S)$.

Time complexity: Let P be an aspect-oriented program and S be the length of execution of P. The total time complexity is due to four components:

1. time required to store each executed statement in a trace file which is $O(S)$.
2. time required to construct the DADG with respect to the execution trace file which is $O(S)$.
3. time required to traverse the DADG and to reach at the specified vertex which is $O(S^2)$.
4. time required to map the traversed vertices to source program P which is $O(S)$.

So, the time complexity of the trace file based algorithm is $O(S^2)$.

7 Implementation

In this section, we briefly describe the implementation of our algorithm. We have named our dynamic slicing tool *dynamic dependence slicing tool* (DDST) for aspect-oriented programs. First, we present an overview of our slicing tool and then, we discuss briefly the implementation of the slicing tool. Next we present some experimental results and then we compare our work with existing work.

7.1 Overview of DDST

The working of the slicing tool is schematically shown in Figure 8. The arrows in the figure show the data-flow among the different blocks of the tool. The blocks shown in rectangular boxes represent executable components and the blocks shown in ellipses represent passive components of the slicing tool.

```
    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;
       }
       }
```

Non−aspect Code (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 7: The dynamic slice of the program given in Figure 2 for the slicing criterion (11,p)

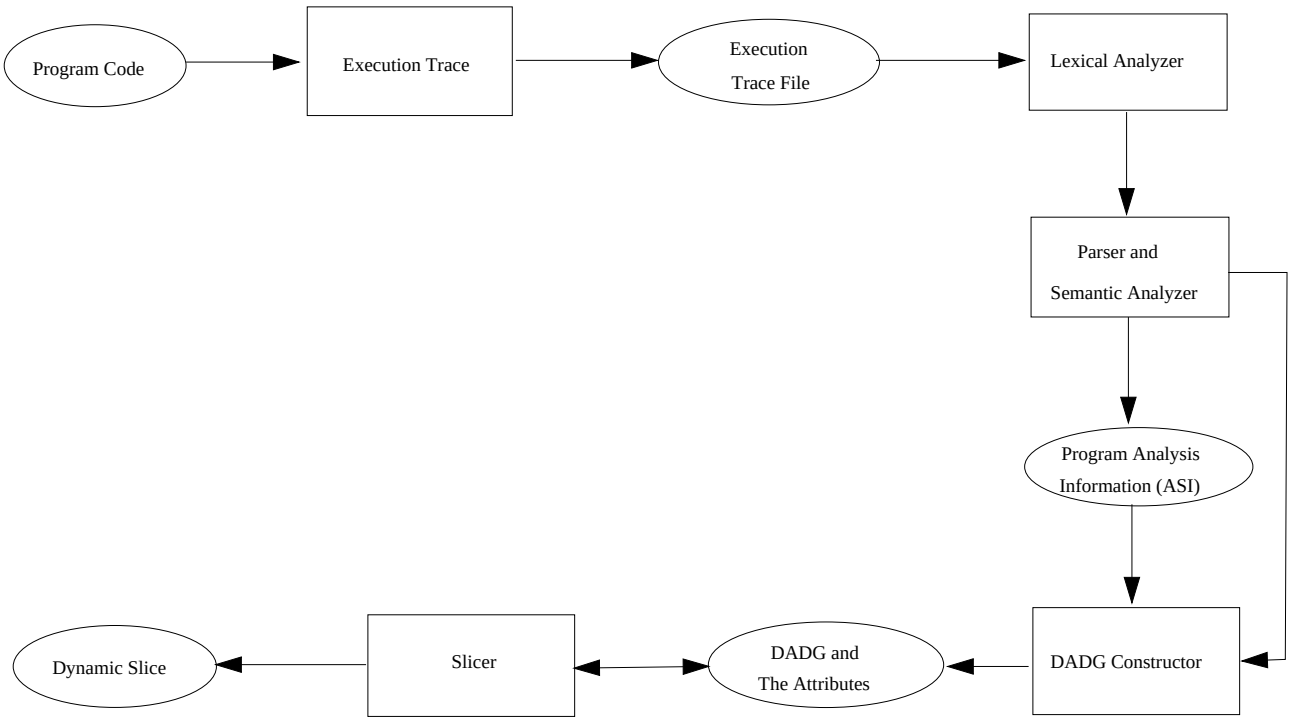

Figure 8: Schematic diagram of the slicing tool

A program written in AspectJ is given as input to DDST. The overall control for the slicer is done through a *coordinator* with the help of a *graphical user interface* (GUI). The *coordinator* takes user input from the GUI, interacts with other relevant components to extract the desired results and returns the output back to the GUI.

The *execution trace* component creates an *execution trace file* for a particular execution of the program. This component takes the user input from the coordinator, stores each executed statement for that input in a file and outputs that file back to the coordinator. This file is called *execution trace file*.

The *lexical analyzer* component reads the execution trace file and breaks it into tokens for the grammar expressed in the parser. When the lexical analyzer component encounters a useful token in the program, it returns the token to the parser describing the type of encountered token.

The *parser and semantic analyzer* component functions as a state machine. The parser takes the token given by the lexical analyzer and examines it using the grammatical rules laid for the input programs. The semantic analyzer component captures the following important information of the program.

- For each vertex u of the program
  - the lexical successor and predecessor vertices of u,
  - the sets of variables defined and used at vertex u,
  - the type of the vertex: assignment or test or method call or return etc.

The lexical component, parser and semantic analyzer component provide the necessary program analysis information to the *DADG constructor* component. The *DADG constructor* component first constructs the CFG and the post-dominator tree of the program using the basic information provided by the lexical and semantic analyzer components. The inter-statement control dependences are captured using the CFG and the post-dominator tree. Then, it constructs the DADG of the program with respect to the trace file along with all the required information to compute slices and stores it in appropriate data structures.

The *slicer* component traverses the DADG. It takes the user input from the *coordinator* and outputs the computed information back to the *coordinator*. The graphical user interface (GUI) functions as a front end to the slicing tool.

7.2 Implementation of the slicing tool

We have implemented our algorithm in Java. We have used the compiler writing tool *ANTLR* (Another Tool for Language Recognition) [1, 32] for *Lexical Analyzer*, *Parser* and *Semantic Analyzer* components of our slicer. ANTLR is a tool that lets one define language grammars in EBNF (Extended Backus-Naur Form) like notations. ANTLR is more than just a grammar definition language. However, the tools provided allow one to implement the ANTLR defined grammar by automatically generating lexers and parsers in Java or other supported languages. ANTLR is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing programming languages such as C++, Java, AspectJ etc. ANTLR is a LL(k) based recognition tool.

Table 1: Encoding used for different types of edges of DADG

Code	Edge Type
0	No Edge
1	Control Dependence Edge (True Case)
2	Control Dependence Edge (False Case)
3	Data Dependence Edge (Loop Independent Case)
4	Data Dependence Edge (Loop Carried Case)
5	Weaving Edge

The sample AspectJ program is executed for a given input. The executed statements are stored in a trace file. This trace file is given as input to the ANTLR program. The lexer part of the ANTLR extracts program tokens and stores the data in a data structure called *statement_info*. The DADG of the AspectJ program is automatically constructed by taking input from the *parser* and *semantic analyzer component*. For constructing the DADG, we have used many flags such as *if_flag* to check whether the statement is an if statement or not, *while_flag* to check whether the statement is a while statement or not etc.

We have used an adjacency matrix *dadg[][]* to store the DADG of the given AspectJ program P. This matrix is of the following type:

```
typedef struct edge {
int exist, type;
} edge;
```

- The attribute *exist* has value 0 or 1. `dadg[i][j].exist` is 1 if there is an edge between node number i and j, otherwise 0.
- The data member *type* specifies the type of the edge. The codes used for this are given in Table 1.

We store the following additional information along with the DADG:

- The set *Def(var)* for each variable var in the aspect-oriented program P.
- The set *Use(var)* for each variable var in the aspect-oriented program P.

The sets *Def(var)* and *Use(var)* are stored using arrays.

7.3 Experimental results

With different slicing criteria, the algorithm has been tested on many programs for 40-50 runs. The sample programs contain loops and conditional statements. Table 2 summarizes the *average run-time requirements* of the trace file based algorithm for several programs. Since we have computed the dynamic slices at different statements of a program, we have calculated the average run-time requirements of our trace file based algorithm. The program sizes are small since right now the tool accepts only a subset of AspectJ constructs. However, the results indicate the overall trend of the performance of the trace file based algorithm.

Table 2: Average runtime

Sl No.	Prg. Size (# stmts)	Trace file Based Algorithm (in Sec.)
1	17	0.48
2	43	0.71
3	69	0.92
4	97	1.14
5	123	1.36
6	245	2.46
7	387	3.96
8	562	5.52

The results in Table 2 indicate that the run-time requirement for the trace file based algorithm increases gradually. This is due to the fact that separate vertices are created in the DADG during run-time for different executions of the same statement. This is followed by a depth-first or breadth-first graph traversal on DADG to compute the dynamic slice. Thus, average run-time requirement becomes high since considerable time is required to perform the traversal on DADG. Furthermore, the algorithm uses a trace file to store the execution history. The time required to read the data from a trace file is significant and is added to the average run-time while computing dynamic slice. All these result in the increase of average run-time requirement gradually.

7.4 Comparison with existing work

Very few work has been done on slicing of aspect-oriented programs [22, 23]. Zhao [22] has proposed an intermediate representation called *Aspect-Oriented System Dependence Graph* (ASDG). The ASDG for the example program of Figure 2 as proposed by Zhao [22] is shown in Figure 9. In this ASDG, the *pointcuts* are not represented. But, our DADG correctly represents the pointcuts.

Zhao and Rinard [23] developed an algorithm to construct the SDG for aspect-oriented programs. Figure 10 shows the SDG of the program given in Figure 2 as proposed by Zhao and Rinard [23]. But, the drawback of this

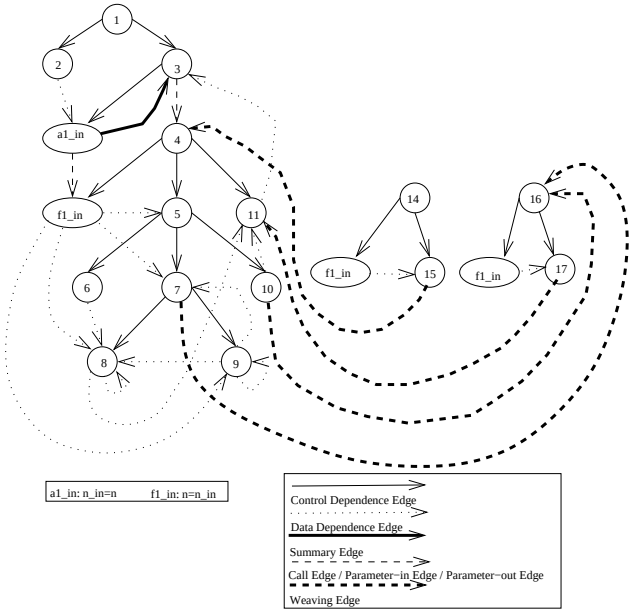

Figure 9: ASDG of the program given in Figure 2 as proposed by Zhao [22]

SDG is that the *weaving process* is not represented correctly. For example, there should be a weaving edge between vertices 15 and 4, beacause, after the execution of *before advice* at statement 14, the actual execution of the method *factorial()* at statement 4 will be started. The use of this SDG to compute dynamic slice results in missing of some statements. So, we cannot use this approach to compute dynamic slice of an aspect-oriented program correctly. Also, they [23] have not considered the dynamic slicing aspects. But in our approach, we have considered the weaving process by adding weaving edges at the appropriate join points in the DADG. Again our algorithm computes precise *dynamic* slices.

8 Conclusion

We proposed an algorithm for dynamic slicing of aspect-oriented programs. First, we have constructed a dependence-based intermediate representation for aspect-oriented programs. We named this representation *Dynamic Aspect-Oriented Dependence Graph* (DADG). Then, we have developed an algorithm to compute dynamic slices of AOPs using the DADG. We have used a trace file to store the execution history. So, we have named our algorithm *Trace file Based Dynamic Slicing* (TBDS) algorithm for AOPs. The resulting dynamic slice in our approach is precise since we create a node in the DADG for each occurrence of a statement in the execution trace. We have developed a tool called *Dynamic Depenedence Slicing Tool* (DDST) to implement our algorithm.

Our algorithm can be extended to compute dynamic

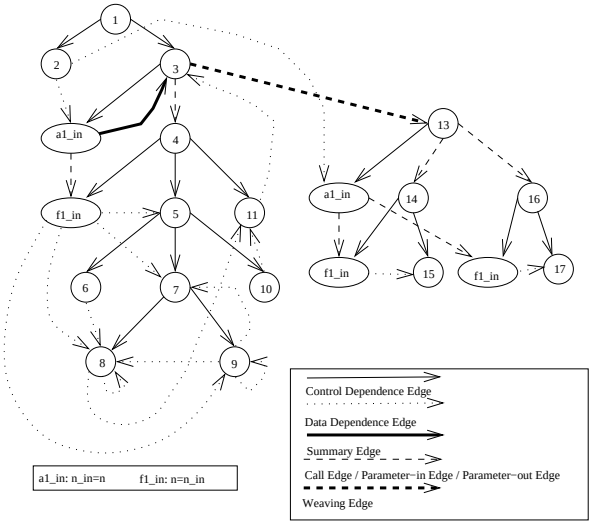

Figure 10: SDG of the program given in Figure 2 as proposed by Zhao and Rinard [23]

slices of concurrent AOPs and distributed AOPs running on different machines connected through a network. The algorithm can also be extended to compute *conditioned slices* with respect to a given condition. Although we have presented the approach for AspectJ, this approach can be easily extended to other aspect-oriented languages such as AspectWerkz, AML, RIDL etc. Our tool can be used to develop efficient debuggers and test drivers for large scale aspect-oriented programs.

References

[1] Antlr. www.antlr.org.

[2] Aspect-Oriented Programming. www.wikipedia.org.

[3] AspectJ. www.eclipse.org/aspectj.

[4] Introduction to AOP. www.media.wiley.com.

[5] Mendhekar A., Kiczales G., and Lamping J. RG: A Case-Study for Aspect-Oriented Programming. Technical report, Xerox Palo Alto Research Center, February 1997.

[6] Dufour B., Goard C., Hendren L., Verbrugge C., Moor O. D., and Sittampalam G. Measuring the Dynamic Behaviour of AspectJ Programs. Technical report, McGill University, School of Computer Science, Sable Research Group and Oxford University, Computing Laboratory, Programming Tools Group, March 2004.

[7] Korel B. and Laski J. Dynamic Program Slicing. *Information Processing Letters*, 29(3):155–163, 1988.

[8] Mund G. B., Mall R., and Sarkar S. Computation of Interprocedural Dynamic Program Slices. *Journal of Information and Software Technology*, 45(8):499–512, June 2003.

[9] Xu B. and Chen Z. Dynamic Slicing Object-Oriented Programs for Debugging. In *Proceedings of 2nd IEEE International Workshop on Source Code Analysis amd Manipulation (SCAM'02)*, pages 115–122, 2002.

[10] Murphy G. C., Walker R. J., and Baniassad E. L. A. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, July-Aug 1999.

[11] Liang D. and Harrold M. J. Slicing Objects Using System Dependence Graph. In *Proceedings of the International Conference on Software Maintenance, IEEE*, pages 358–367, November 1998.

[12] Tip F. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[13] Cugola G., Ghezzi C., and Monga M. Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming. In *Proceedings of Workshop on System Distribution: Algorithm, Architecture and Language, WSDAAL*. Italy, 1999.

[14] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Griswold W. G. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*. Budapest,Hungary, 18-22 June 2001.

[15] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., and Grisworld W. G. Report on An Overview of Aspectj. Notes by Tai Hu, CMSC631 Fall 2002.

[16] Kiczales G., Irwin J., Lamping J., Loingtier J. M., Lopes C. V., Maeda C., and Mendhekar A. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[17] Kiczales G. and Mezini M. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering, ICSE'05*, May 2005.

[18] Agarwal H. and Horgan J. R. Dynamic Program Slicing. In *ACM SIGPLAN Notices, Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation PLDIŠ90*, volume 25 of *6*, pages 246–256, June 1990.

[19] Agrawal H. and Horgan J. R. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation, SIGPLAN Notices, Analysis and Verification*, volume 25, pages 246–256, 1990.

[20] Irwin J., Loingtier J. M., Gilbert J. R., Kiczales G., Lamping J., Mendhekar A., and Shpeisman T. Aspect-Oriented Programming of Sparse Matrix Code. In *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), Marina del Rey, CA*. Springer-Verlag, December 1997.

[21] Zhao J. Dynamic Slicing of Object-Oriented Programs. Technical report, Information Processing Society of Japan, May 1998.

[22] Zhao J. Slicing Aspect-Oriented Software. In *Proceedings of 10th International Workshop on Program Comprehension*, pages 251–260, June 2002.

[23] Zhao J. and Rinard M. System Dependence Graph Construction for Aspect-Oriented Programs. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, USA, March 2003.

[24] Larsen L. and Harrold M. J. Slicing Object-Oriented Software. In *Proceedings of 18th International Conference on Software Engineering*, pages 495–505, March 1996.

[25] Sahu M. and Mohapatra D. P. A Node Marking Technique for Dynamic Slicing of Aspect-Oriented Programs. In *Proceedings of the 10th International Conference on Information Technology (ICIT'07)*, pages 155–160, 2007.

[26] Weiser M. Programmers Use Slices When Debugging. *Communications of the ACM*, 25(7):446–452, July 1982.

[27] Mohapatra D. P. *Dynamic Slicing of Object-Oriented Programs*. PhD thesis, Indian Institute of Technology, Kharagpur, May 2005.

[28] Mohapatra D. P., Mall R., and Kumar R. Dynamic Slicing of Concurrent Object-Oriented Programs. In *Proceedings of International Conference on Information Technology: Progresses and Challenges (ITPC)*, pages 283–290. Kathamandu, May 2003.

[29] Mohapatra D. P., Mall R., and Kumar R. An Edge Marking Technique for Dynamic Slicing of Object-Oriented Programs. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, 2004.

[30] Mohapatra D. P., Mall R., and Kumar R. A Node-Marking Technique for Dynamic Slicing of Object-Oriented Programs. In *Proceedings of Conference on Software Design and Architecture (SODA'04)*, 2004.

[31] Mohapatra D. P., Mall R., and Kumar R. An Overview of Slicing Techniques for Object-Oriented Programs. *Informatica*, 30:253–277, 2006.

[32] Levine J. R. Mason T. and Brown D. *Lex and Yacc*. O'REILLY, 3rd edition, 2002.

[33] Horwitz S., Reps T., and Binkley D. Inter-Procedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[34] Braak T. T. Extending Program Slicing in Aspect-Oriented Programming with Inter-Type Declarations. 5th TSConIT Program, June 2006.

[35] Ishio T., Kusumoto S., and Inoue K. Program Slicing Tool for Effective Software Evolution Using Aspect-Oriented Technique. In *Proceedings of Sixth International Workshop on Principles of Software Evolution*, pages 3–12. IEEE Press, September 2003.

[36] Song Y. T. and Huynh D. T. Forward Dynamic Object-Oriented Program Slicing. In *Proceedings of IEEE Symposium on Application Specific Systems and Software Engineering and Technology (ASSET'99)*, pages 230–237, Richardson, TX, USA, 03/24/1999-03/27/1999 1999.

[37] Wang T. and Roychoudhury A. Using Compressed Bytecode Traces for Slicing Java Programs. In *Proceedings of 26th International Conference on Software Engineering (ICSE'04)*, pages 512–521, 23-28 May 2004.

[38] Binkley D. W. and Gallagher K. B. Program Slicing. *Advances in Computers*, 43, 1996. Academic Press, San Diego, CA.

[39] Chen Z. and Xu B. Slicing Object-Oriented Java Programs. *ACM SIGPLAN Notices*, 36(4), April 2001.