Computing Dynamic Slices of Feature-Oriented Programs with Aspect-Oriented Extensions

Madhusmita Sahu and Durga Prasad Mohapatra
Department of Computer Science and Engineering
National Institute of Technology, Rourkela-769008
Rourkela, Odisha, India
E-mail: madhu_sahu@yahoo.com and durga@nitrkl.ac.in

Keywords: feature-oriented programming (FOP), aspect-oriented programming (AOP), composite feature-aspect dependence graph (CFADG), mixin layer, refinement chain

Received: September 10, 2018

This paper proposes a technique to compute dynamic slices of feature-oriented programs with aspect-oriented extensions. The technique uses a dependence based intermediate program representation called composite feature-aspect dependence graph (CFADG) to represent feature-oriented software that contain aspects. The CFADG of a feature-oriented program is based on the selected features that are composed to form a software product and the selected aspects to be weaved. The proposed dynamic slicing technique has been named feature-aspect node-marking dynamic slicing (FANMDS) algorithm. The proposed feature-aspect node marking dynamic slicing algorithm is based on marking and unmarking the executed nodes in the CFADG suitably during run-time. The advantage of the proposed approach is that no trace file is used to store the execution history. Also, the approach does not create any additional nodes during run-time.

Povzetek: Prispevek predstavlja izvirni pristop pri programiranju na osnovi sprejemljivk z aspektno orientiranimi podaljški. Gre za računanje dinamičnih odsekov omenjenih programov.

1 Introduction

Weiser [33] first introduced the concept of a program slice. Program slicing decomposes a program into different parts related to a particular computation. A *slicing criterion* is used to construct a program slice. A slicing criterion is a tuple, $< s, v >$, consisting of a statement s, in a program and a variable v, used or defined at that statement s. Program slicing technique is employed in many areas of software engineering including debugging, program understanding, testing, reverse engineering, etc.

Feature-oriented programming (FOP) is concerned with the separate definition of individual features and the composition of required features to build varieties of a particular software product. The functionalities of a software product are identified as features in FOP paradigm. FOP is used to implement software product lines and incremental designs. A family of software systems constitutes a software product line [20].

Motivation: Today, the variability of software products is crucial for successful software development. One mechanism to provide the required variability is through Software Product Lines, which is inspired by product lines used in the industry, like product lines used in the production of a car or a meal at some fast-food restaurant. Feature-Oriented Programming (FOP) approach is used to implement software product lines. Despite the advantages, feature-oriented programming (FOP) yields some problems in expressing features such as lack of expressing crosscutting modularity. During software evolution, a software should adapt the unanticipated requirements and circumstances. This leads to modifications and extensions that crosscut many existing implementation units. The problem of crosscutting modularity is solved by using aspect-oriented programming (AOP). Kiczales et al. [8] proposed AOP paradigm to separate and modularize the crosscutting concerns like exception handling, synchronization, logging, security, resource sharing, etc. The modularity of crosscutting concerns in FOP can be improved by integrating AOP paradigm into FOP paradigm. In dynamic slicing techniques, first an intermediate representation of the program is statically created in the form of a dependence graph. Then, the dynamic slice is computed by traversing the graph, starting from the point specified in the slicing criterion, using an algorithm. For the programs in general languages like C/C++, Java etc., a single dependence graph is created. There is no composition of features in these languages. FOP is used to develop a family of software products. In FOP (with AOP extensions), multiple dependence graphs are created depending upon the composition of features and aspects. For example, if there are four features and two aspects in a product line out of which two features and one aspect are mandatory, then there are eight possible combinations of features

and aspects. Each possible combination of features and aspects creates a different product. Thus, there are eight software products in the product line. Accordingly, there are eight dependence graphs, one graph for each product. Dynamic slice for each possible combination of features and aspects is computed using the corresponding dependence graph. The dynamic slice consists of statements from the composed program that is generated after composition of features and aspects. These statements are again mapped back to the program used for composition. This mapping is not required in general languages like C/C++, Java etc. Again, feature-oriented programs have some special characteristics such as mixins, mixin layers, refinements etc. which are not present in case of general languages like C/C++, Java etc. These characteristics of feature-oriented programs require inclusion of some new nodes/edges in the dependence graph. Similarly, these characteristics require introduction of some new steps/phases in the slicing algorithm (e.g., the handling mixins, the handling of mixin layers, etc.), which are not required in the case of general languages like C/C++, Java, etc. The existing dynamic slicing algorithms for aspect-oriented programs cannot be directly applied for slicing of feature-oriented programs with aspect-oriented extensions due to the specific features of feature-oriented programs such as the presence of mixin layers, refinements of classes, refinements of constructors etc. These characteristics of feature-oriented programs requires inclusion of some new nodes/edges in the dependence graph. Similarly, these characteristics require the introduction of some new steps/phases in the slicing algorithm. Although FOP is an extension of OOP, the existing dynamic slicing algorithms for C/C++, Java cannot be directly applied for slicing of feature-oriented programs due to the presence of aforementioned specific features. Since, program slicing has many applications including testing, software maintenance etc., there is an increasing demand for slicing of feature-oriented programs.

Objective: The main objectives of this work are to develop a suitable intermediate representation of feature-oriented programs with aspect-oriented extension and to propose an efficient slicing algorithm to compute dynamic slices for the above types of programs using the developed intermediate representation. A dependence graph is used to signify the intermediate representation. For a single feature-oriented program, more than one dependence graph can be obtained depending on the number of features to be composed and the number of aspects to be captured. We also aim at calculating the slice computation time for different compositions of features and different aspects captured.

Organization: The organization of rest of the paper is as follows. Section 2 provides a brief introduction to feature-oriented programming (FOP) and program slicing. Section 3 discusses the construction of *composite feature-aspect dependence graph* (CFADG), which is a dependence based intermediate representation of feature-oriented programs containing aspects. In Section 4, the details of our proposed algorithm named *feature-aspect node marking dynamic slicing* (FANMDS) algorithm, is discussed. This section also presents the space and time complexity of FANMDS algorithm. Section 5 furnishes a brief overview of the implementation of FANMDS algorithm along with experimental results. A brief comparison of the proposed work with some other related work is furnished in Section 6. Section 7 concludes the paper along with some possible future work.

2 Basic concepts

In this Section, we provide some basic concepts of feature-oriented programming and outline the features of Jak language, which is a feature-oriented programming language. We also discuss the problems of feature-oriented programming and solutions to these problems through aspect-oriented programming extensions.

2.1 Feature-oriented programming (FOP)

Prehofer [1] was the pioneer to coin the term feature-oriented programming (FOP). The key idea behind FOP is to build the software by composing *features*. Features are the basic building blocks that are used to satisfy user requirements on a software system. The *step-wise refinement* where features incrementally refine other features leads to a stack of features that are arranged in layers. One suitable technique to implement the features is through the use of *Mixin Layers*. A Mixin Layer is a static component that encapsulates fragments of several different classes (Mixins) to compose all fragments consistently. Several languages like Jak [2],[1], Fuji[2], FeatureHouse[3], FeatureRuby [40, 41], FeatureC++ [5, 3, 4] support the concept of features explicitly. FeatureIDE [6][4] is a tool that supports Feature-Oriented Software Development (FOSD) in many languages. We have taken Jak program as input in our proposed approach as it is supported by *Algebraic Hierarchical Equations for Application Design* (AHEAD) tool suite, which is a composer. AHEAD tool suite is a group of tools to work with Jak language. Other languages have their own composers, and those composers are not a group of tools. Jak (short for Jakarta) is a language that extends Java by incorporating feature-oriented mechanisms [2]. Jak-specific tools like *jampack* and *mixin* are invoked to compose Jak files. A Jak file is translated to its Java counterpart using another tool called *jak2java*. The different features supported by Jak language are *Super()* references, an extension of constructors, declaration of local identifiers, etc. The details of Jak language and its features can be found in [2].

[1] https://juliank.files.wordpress.com/2012/04/jlang1.pdf

[2] http://www.infosun.fim.uni-passsau.de/spl/apel/fuji

[3] http://www.fosd.de/fh

[4] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/deploy

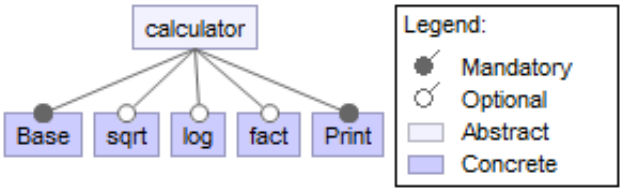


Figure 1: Features supported by Calculator Product Line (CPL)

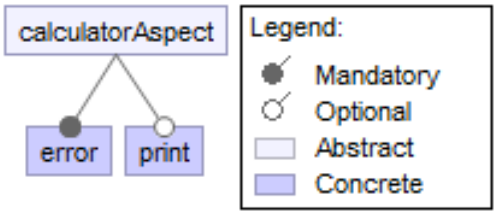


Figure 2: Aspects captured in Calculator Product Line (CPL)

Example 2.1. ***Calculator Product Line (CPL) [45]:*** *This program calculates the factorial, square root and logarithmic value with base 10 of a number. This program is referred to as Calculator Product Line (CPL).*

A *feature-tree* depicts various features supported by a product line in a hierarchical manner. The feature-tree for Example 2.1 is given in Figure 1. The various aspects that are captured for Example 2.1 are given in Figure 2. Figure 3 depicts the source code for each feature given in Figure 1 and each aspect given in Figure 2. Figure 4(a) – Figure 4(b) show the resultant files generated after the composition of all features.

2.2 Problems in feature-oriented programming (FOP)

Feature-oriented programming (FOP) suffers from many problems in modularization of crosscutting concerns [3, 4, 5]. The presence of these problems leads to degradation in modularity of program family members and also decrease in maintainability, customizability, and evolvability. Some of the problems of FOP are discussed below.

1. FOP is unable to express *dynamic crosscutting concerns* that affect the control flow of a program. It can only express *static crosscutting concerns* that affect the structure of the program. AOP languages can handle dynamic crosscutting concerns in an efficient manner through the use of pointcuts, advices etc.

2. FOP languages support only *heterogeneous crosscutting concerns* where different join points are provided with different pieces of codes. In contrast, AOP languages support *homogeneous crosscutting concerns* where different join points are provided with the same piece of code.

3. FOP suffers from excessive method extension problem when a feature crosscuts a large fraction of existing classes because of refinements. A lot of methods are to be overridden for each method on which a crosscut depends. This is because FOP is unable to modularize homogeneous crosscutting concerns. AOP uses wildcards in pointcuts to deal with this problem.

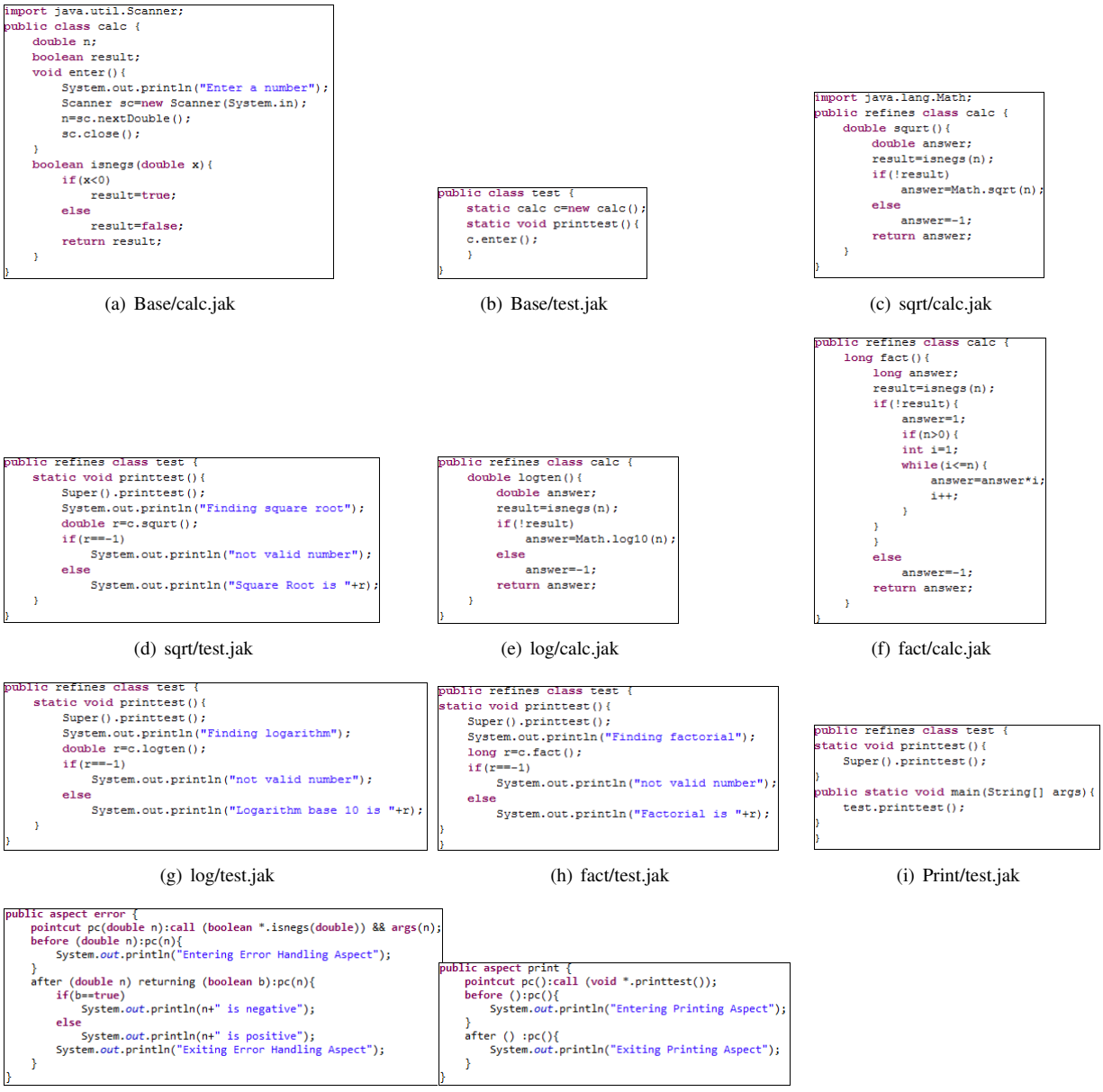
2.3 AOP extensions to FOP

AOP can be used to solve the above problems of FOP by integrating AOP language features like wildcards, pointcuts, and advices into FOP languages. The different approaches used for integrating AOP language features into FOP languages are *Multi Mixins*, *Aspectual Mixin Layers*, *Aspectual Mixins*. More details about these approaches can be found in [5, 3, 4]. The *Aspectual Mixin Layers* approach is a popular one amongst all the approaches since this approach overcomes all the aforementioned problems. Other approaches overcome some of the problems. We have used the approach of aspectual mixin layers in our work. We have separated the aspects from mixin layers for easy understanding of our approach. Our mixin layers contain only a set of classes. Aspects are designed as different layers.

2.4 Program slicing

Program slicing is a technique which is employed to analyze the behavior of a program on the basis of dependencies that exist between various statements. It takes out statements from a program related to a specific computation. The extracted statements constitute a slice. Thus, a slicing criterion is employed to compute a slice. A slicing criterion consists of a statement s (or location in the program) and a variable v (or set of variables), and it is represented as a tuple $< s, v >$. Program slicing technique can be either *static* or *dynamic* based on the input to the program. A program slicing technique is said to be *static* when it extracts all the statements from a program with respect to a slicing criterion *irrespective* of the input to the program. On the other hand, a program slicing technique is said to be *dynamic* when all the statements from a program are extracted with respect to a slicing criterion for a *specific* input to the program.

The difference between *static slicing* and *dynamic slicing* can be understood by taking an example. Let us consider the example C program given in Figure 5. The static slice with respect to slicing criterion $< 11, y >$ is depicted as the bold italic statements in Figure 6. It includes statements 1, 2, 3, 4, 6, 7, 9, and 11. The dynamic slice with respect to slicing criterion $< \{x = 10\}, 11, y >$ is depicted as the bold italic statements in Figure 7. It includes statements 1, 2, 3, 6, 9, and 11. For finding the slices of a program, first an intermediate representation of the program is constructed. Then, the slices are found out by using some algorithm on the intermediate representation. There are many slicing algorithms found in the literature



```
import java.util.Scanner;
public class calc {
    double n;
    boolean result;
    void enter(){
        System.out.println("Enter a number");
        Scanner sc=new Scanner(System.in);
        n=sc.nextDouble();
        sc.close();
    }
    boolean isnegs(double x){
        if(x<0)
            result=true;
        else
            result=false;
        return result;
    }
}
```

(a) Base/calc.jak

```
public class test {
    static calc c=new calc();
    static void printtest(){
    c.enter();
    }
}
```

(b) Base/test.jak

```
import java.lang.Math;
public refines class calc {
    double squrt(){
        double answer;
        result=isnegs(n);
        if(!result)
            answer=Math.sqrt(n);
        else
            answer=-1;
        return answer;
    }
}
```

(c) sqrt/calc.jak

```
public refines class test {
    static void printtest(){
        Super().printtest();
        System.out.println("Finding square root");
        double r=c.squrt();
        if(r==-1)
            System.out.println("not valid number");
        else
            System.out.println("Square Root is "+r);
    }
}
```

(d) sqrt/test.jak

```
public refines class calc {
    double logten(){
        double answer;
        result=isnegs(n);
        if(!result)
            answer=Math.log10(n);
        else
            answer=-1;
        return answer;
    }
}
```

(e) log/calc.jak

```
public refines class calc {
    long fact(){
        long answer;
        result=isnegs(n);
        if(!result){
            answer=1;
            if(n>0){
            int i=1;
            while(i<=n){
                answer=answer*i;
                i++;
            }
        }
        }
        else
            answer=-1;
        return answer;
    }
}
```

(f) fact/calc.jak

```
public refines class test {
    static void printtest(){
        Super().printtest();
        System.out.println("Finding logarithm");
        double r=c.logten();
        if(r==-1)
            System.out.println("not valid number");
        else
            System.out.println("Logarithm base 10 is "+r);
    }
}
```

(g) log/test.jak

```
public refines class test {
static void printtest(){
    Super().printtest();
    System.out.println("Finding factorial");
    long r=c.fact();
    if(r==-1)
        System.out.println("not valid number");
    else
        System.out.println("Factorial is "+r);
}
}
```

(h) fact/test.jak

```
public refines class test {
static void printtest(){
    Super().printtest();
}
public static void main(String[] args){
    test.printtest();
}
}
```

(i) Print/test.jak

```
public aspect error {
    pointcut pc(double n):call (boolean *.isnegs(double)) && args(n);
    before (double n):pc(n){
        System.out.println("Entering Error Handling Aspect");
    }
    after (double n) returning (boolean b):pc(n){
        if(b==true)
            System.out.println(n+" is negative");
        else
            System.out.println(n+" is positive");
        System.out.println("Exiting Error Handling Aspect");
    }
}
```

(j) error.aj

```
public aspect print {
    pointcut pc():call (void *.printtest());
    before ():pc(){
        System.out.println("Entering Printing Aspect");
    }
    after () :pc(){
        System.out.println("Exiting Printing Aspect");
    }
}
```

(k) print.aj

Figure 3: Jak program for Calculator Product Line (CPL) along with aspect code. Figure 3(a)–Figure 3(i) represent base code or non-aspect code written in Jak language and Figure 3(j)–Figure 3(k) represent aspect code written in AspectJ

```
   import java.util.Scanner;
   import java.lang.Math;
c1  abstract class calc$$Base {
     double n;
     boolean result;
m2   void enter(){
s3         System.out.println("Enter a number");
s4         Scanner sc=new Scanner(System.in);
s5         n=sc.nextDouble();
s6         sc.close();
     }
m7   boolean isnegs(double x){
s8         if(x<0)
s9               result=true;
           else
s10              result=false;
s11        return result;
     }
   }
c12 abstract class calc$$sqrt extends  calc$$Base  {
m13   double sqrt(){
           double answer;
s14        result=isnegs(n);
s15        if(!result)
s16              answer=Math.sqrt(n);
           else
s17              answer=-1;
s18        return answer;
     }
   }
c19 abstract class calc$$log extends  calc$$sqrt  {
m20   double logten(){
           double answer;
s21        result=isnegs(n);
s22        if(!result)
s23              answer=Math.log10(n);
           else
s24              answer=-1;
s25        return answer;
     }
   }
c26 public class calc extends  calc$$log  {
m27   long fact(){
           long answer;
s28        result=isnegs(n);
s29        if(!result){
s30              answer=1;
s31              if(n>0){
s32              int i=1;
s33              while(i<=n){
s34                    answer=answer*i;
s35                    i++;
                 }
                 }
           }
           else
s36              answer=-1;
s37        return answer;
     }
   }
```

(a) calc.java

```
c38 abstract class test$$Base {
s39  static calc c=new calc();
m40  static void printtest(){
s41  c.enter();
     }
   }
c42 abstract class test$$sqrt extends  test$$Base  {
m43  static void printtest(){
s44        test$$Base.printtest();
s45        System.out.println("Finding square root");
s46        double r=c.squrt();
s47        if(r==-1)
s48              System.out.println("not valid number");
           else
s49              System.out.println("Square Root is "+r);
     }
   }
c50 abstract class test$$log extends  test$$sqrt  {
m51  static void printtest(){
s52        test$$sqrt.printtest();
s53        System.out.println("Finding logarithm");
s54        double r=c.logten();
s55        if(r==-1)
s56              System.out.println("not valid number");
           else
s57              System.out.println("Logarithm base 10 is "+r);
     }
   }
c58 abstract class test$$fact extends  test$$log  {
m59  static void printtest(){
s60  test$$log.printtest();
s61  System.out.println("Finding factorial");
s62  long r=c.fact();
s63  if(r==-1)
s64        System.out.println("not valid number");
     else
s65        System.out.println("Factorial is "+r);
     }
   }
c66 public class test extends  test$$fact  {
m67  static void printtest(){
s68  test$$fact.printtest();
     }
m69  public static void main(String[] args){
s70  test.printtest();
     }
   }
```

(b) test.java

```
as71 public aspect error {
p72    pointcut pc(double n):call (boolean *.isnegs(double)) && args(n);
a73    before (double n):pc(n){
s74          System.out.println("Entering Error Handling Aspect");
       }
a75    after (double n) returning (boolean b):pc(n){
s76          if(b==true)
s77                System.out.println(n+" is negative");
           else
s78                System.out.println(n+" is positive");
s79          System.out.println("Exiting Error Handling Aspect");
       }
     }
```

(c) error.aj

```
as80 public aspect print {
p81    pointcut pc():call (void *.printtest());
a82    before ():pc(){
s83          System.out.println("Entering Printing Aspect");
       }
a84    after () :pc(){
s85          System.out.println("Exiting Printing Aspect");
       }
     }
```

(d) print.aj

Figure 4: Java codes generated (Figure 4(a)–Figure 4(b)) after the composition of all features depicted in Figure 1. Figure 4(c)–Figure 4(d) are the AspectJ codes for the aspects captured.

```
    #include<stdio.h>
1   main(){
    int x,y,z;
2   scanf("%d",&x);
3   if(x<0){
4   y=x+5;
5   z=x*2;
    }
6   else if(x==0){
7   y=x+20;
8   z=x*8;
    }
    else{
9   y=x+4;
10  z=x*3;
    }
11  printf("y= %d",y);
12  printf("z= %d",z);
  }
```

Figure 5: An example C program

```
    #include<stdio.h>
1   main(){
    int x,y,z;
2   scanf("%d",&x);
3   if(x<0){
4   y=x+5;
5   z=x*2;
    }
6   else if(x==0){
7   y=x+20;
8   z=x*8;
    }
    else{
9   y=x+4;
10  z=x*3;
    }
11  printf("y= %d",y);
12  printf("z= %d",z);
  }
```

Figure 6: Static slice with respect to slicing criterion $< 11, y >$

```
    #include<stdio.h>
1   main(){
    int x,y,z;
2   scanf("%d",&x);
3   if(x<0){
4   y=x+5;
5   z=x*2;
    }
6   else if(x==0){
7   y=x+20;
8   z=x*8;
    }
    else{
9   y=x+4;
10  z=x*3;
    }
11  printf("y= %d",y);
12  printf("z= %d",z);
  }
```

Figure 7: Dynamic slice with respect to slicing criterion $< \{x = 10\}, 11, y >$

[10, 11, 12, 14, 15, 16, 17, 19, 21, 25, 26, 37, 38, 42]. For the details of the intermediate program representations and different slicing algorithms, the readers may refer to [10, 11, 12, 14, 15, 16, 17, 19, 21, 25, 26, 37, 38, 42]. In the next section, we propose an intermediate program representation for feature-oriented programs, on which our slicing algorithm can be applied.

3 Composite feature-aspect dependence graph (CFADG): an intermediate representation for feature-oriented programs

We have proposed an intermediate representation for feature-oriented programs, called Composite Feature-Aspect Dependence Graph (CFADG). CFADG is an arc-classified digraph, $G = (N, E)$, where N is the set of vertices depicting the statements and E is the set of edges symbolizing the dependence relationships between the statements. The set E captures various dependencies that exist between the statements in various mixin layers and aspects in a feature-oriented program. CFADG is constructed based on the composition of different features and aspects captured. Thus, there will be different types of CFADGs according to the features composed and aspects captured. Figure 8 shows the CFADG for the composition given in

Figure 3. The square box with a1_in: n_in=n etc. specifies the actual and formal parameters. For example: a1_in: n_in=n specifies that n is an actual-in parameter. Similarly, a2_in: b_in=b specifies that b is an actual-in parameter, f1_in: x=n_in specifies that x is a formal-in parameter, f2_in: b=b_in specifies that b is a formal-in parameter. These notations are adopted from Horwitz et al. [47]. The construction of CFADG consists of the following steps:

- Constructing Procedure Dependence Graph (PDG) for each method in a mixin.
- Constructing Mixin Dependence Graph (MxDG) for each mixin.
- Constructing System Dependence Graph (SDG) for each mixin layer.
- Constructing Advice Dependence Graph (ADG) for each advice.
- Constructing Introduction Dependence Graph (IDG) for each introduction.
- Constructing Pointcut Dependence Graph (PtDG) for each pointcut.
- Constructing Aspect Dependence Graph (AsDG) for each aspect.
- Constructing Composite Feature Aspect Dependence Graph (CFADG) by combining all the SDGs and AsDGs.

Below, we briefly explain the steps for constructing the CFADG and the pseudocode.

(1) Construction of Procedure Dependence Graph (PDG)

A *procedure dependence graph* (PDG) depicts the control and data dependence relationships that exist between the statements in a program with only one function/method/procedure. The nodes in the graph correspond to the program statements and edges correspond to the dependence relationships between the statements.

(2) Construction of Mixin Dependence Graph (MxDG)

A *mixin dependence graph* (MxDG) is used to capture all dependencies within a mixin. A MxDG has a *mixin entry vertex* that connects the method entry vertex of each method in the mixin by a *mixin membership edge*. Each method entry in the MxDG is associated with *formal-in* and *formal-out* parameter nodes. The interactions among methods in a mixin occur by calling each other. This effect of method calls is symbolized by a *call* node in a MxDG. *Actual-in* and *actual-out* parameter nodes are created at each call node corresponding to formal-in and formal-out parameter nodes. The effect of *return* statements in a MxDG is represented by joining each *return* node to its corresponding *call* node through a *return dependence edge*.

(3) Construction of System Dependence Graph (SDG) for each Mixin Layer

A single mixin layer may contain more than one mixin. A mixin may derive another mixin through inheritance. The MxDG for the derived class is constructed. The mixin membership edges connect the mixin entry vertex of derived class to the method entry vertices of all those methods that are defined and inherited in the derived class. The SDG for a mixin layer is constructed by joining all the mixin dependence graphs for that mixin layer through parameter edges, call edges and summary edges.

(4) Construction of Advice Dependence Graph (ADG)

An *advice dependence graph* (ADG) represents an advice in an aspect. The statements or predicates in the advice are represented as vertices and dependencies amongst statements are represented as edges in an ADG. Each ADG is associated with a unique vertex called *advice start vertex* to signify entry into the advice.

(5) Construction of Introduction Dependence Graph (IDG)

An *introduction dependence graph* (IDG) represents an introduction in an aspect. If an introduction is a method or constructor, then its IDG is similar to PDG of a method. A unique vertex, called *introduction start vertex*, is used in IDG to signify the entry into the introduction.

(6) Construction of Pointcut Dependence Graph (PtDG)

Pointcuts in an aspect contain no body. Therefore, to represent pointcuts, only a *pointcut start vertex* is created to denote the entry into the pointcut.

(7) Construction of Aspect Dependence Graph (AsDG)

An *aspect dependence graph* (AsDG) is used to represent a single aspect. It consists of a collection of ADGs, IDGs, PtDGs that are connected by some special kinds of edges. Each AsDG is associated with a unique vertex called *aspect start vertex*, to represent entry into the aspect. An *aspect membership edge* is used to represent the membership relationships between an aspect and its members. This edge connects the aspect start vertex to each start vertex of an ADG, IDG or PtDG. Each pointcut start vertex is connected to its corresponding advice start vertex by call edges.

(8) Construction of Composite Feature-Aspect Dependence Graph (CFADG)

The CFADG is constructed by combining the SDGs for all mixin layers present in the composition and the AsDGs through special kinds of edges. The SDGs for all mixin layers in a composition are connected using refinement edges, mixin call edges, mixin data dependence edges, and mixin return dependence edges. The AsDGs are connected to all the SDGs through weaving edges and aspect data dependence edges. The mixin membership edges and aspect membership edges along with mixin start vertices and aspect start vertices are removed during construction of CFADG. The CFADG for the program given in Figure 3 is shown in Figure 8. A CFADG contains the following types of edges:

(a) **Control dependence edge:** A *control dependence edge* in a CFADG from a node n_1 to a node n_2 indicates that either node n_2 is under the scope of node n_1 or node n_1 controls the execution of node n_2 where node n_1 is a predicate node.
In Figure 8, edge $(m20, s21)$ is a control dependence edge.

(b) **Data dependence edge:** A *data dependence edge* in a CFADG from a node n_1 to a node n_2 indicates that node n_2 uses a variable that is assigned a value at node n_1 or n_1 creates an object o and o is used at n_2.
In Figure 8, edges $(s21, s22)$, $(s39, s41)$, $(p72, a73)$ are data dependence edges.

(c) **Mixin data dependence edge:** A *mixin data dependence edge* in a CFADG from a node n_1 to a node n_2 indicates that node n_2 in a mixin layer defines a variable which is used at node n_1 in another mixin layer.
In Figure 8, edges $(s5, s16)$ and $(s39, s54)$ are mixin data dependence edges.

(d) **Aspect data dependence edge:** An *aspect data dependence edge* in a CFADG from a node n_1 to a node n_2 indicates that node n_2 in an aspect uses the value of a variable and that variable is defined at node n_1 in a mixin.
In Figure 8, edge $(s5, a1\_in)$ is an aspect data dependence edge.

(e) **Call edge:** A *call edge* in CFADG from a node n_1 to a node n_2 indicates that node n_1 calls a method defined at node n_2. Both the nodes n_1 and n_2 are in same mixin layer.
In Figure 8, edge $(s41, m2)$ is a call edge.

(f) **Mixin call edge:** A *mixin call edge* in CFADG from a node n_1 to a node n_2 indicates that node n_1 in a mixin layer calls a method that is defined in a different mixin layer at node n_2.
In Figure 8, edge $(s28, m7)$ is a mixin call edge.

(g) **Return dependence edge:** A *return dependence edge* in a CFADG from node n_1 to node n_2 indicates that node n_1 in a mixin layer returns a value to node n_2 in the same mixin layer and node n_2 calls a method where node n_1 is present.
In Figure 8, edge $(s18, s46)$ is a return dependence edge.

(h) **Mixin return dependence edge:** A *mixin return dependence edge* in a CFADG from node n_1 to node n_2 indicates that node n_1 in one mixin layer returns a value to node n_2 in another mixin layer and node n_2 calls a method where node n_1 is present.
In Figure 8, edge $(s11, s21)$ is a mixin return dependence edge.

(i) **Parameter-in edge:** *Parameter-in edge* in CFADG is added from actual-in parameters to corresponding formal-in parameters to indicate the receipt of values from the calling method to the called method.
In Figure 8, edges $(s14 \rightarrow a1\_in, m7 \rightarrow f1\_in)$, $(s21 \rightarrow a1\_in, m7 \rightarrow f1\_in)$, and $(s28 \rightarrow a1\_in, m7 \rightarrow f1\_in)$ are parameter-in edges.

(j) **Parameter-out edge:** *Parameter-out edge* is added from formal-out parameters to corresponding actual-out parameters to indicate the return of values from the called method to the calling method. If an actual parameter is modified inside a method, then the modified value becomes an actual-out parameter and the original value becomes an actual-in parameter. The parameter used to hold the value of actual-in parameter in method definition becomes a formal-in parameter and the parameter used to hold the modified value becomes a formal-out parameter.
In Figure 8, there are no parameter-out edges, since, in our example, no parameter is modified inside a method.

(k) **Summary edge:** The *summary edge* is used to represent the transitive flow of dependence between an actual-in parameter node and an actual-out parameter node if the value of the actual-in parameter node affects the value of the corresponding actual-out vertex.
In Figure 8, edges $(s14 \rightarrow a1\_in, s14)$, $(s21 \rightarrow a1\_in, s21)$, and $(s28 \rightarrow a1\_in, s28)$ are summary edges.

(l) **Message dependence edge:** A message dependence edge from a node n_1 to another node n_2 in a dependency graph signifies that node n_1 represents a statement outputting some message without using any variable and node n_2 represents an input statement, a computation statement, a method call statement, or a predicate statement.
In Figure 8, there exists a message dependence edge $(s3, s4)$. Similarly, edges $(s45, s46)$, $(s53, s54)$, and $(s61, s62)$ are message dependence edges.

(m) **Refinement edge:** A *refinement edge* in a CFADG from a node n_1 to a node n_2 indicates that node n_1 in child mixin layer calls a method $k()$ by prefacing *Super()* call and $k()$ is defined at node n_2 in parent mixin layer.
In Figure 8, the edge $(s44, m40)$ is a refinement edge. Similarly, edges $(s68, m59)$, $(s60, m51)$, and $(s52, m43)$ are refinement edges.

(n) **Weaving edge:** A weaving edge from a node n_1 to node n_2 indicates that

- node n_1 is a method call node and node n_2 is a *before* advice node capturing the method called at n_1 and node n_2 executes before the method called by n_1 executes. OR
- node n_1 is the last statement in *before* advice and node n_2 is the method entry node of the method captured by the advice and node n_2 executes after node n_1 executes. OR
- node n_2 is an *after* advice node and node n_1 is the last statement in the method captured by node n_2 and node n_2 executes after node n_1 executes. OR
- node n_1 is the last statement in an *after* advice and node n_2 is the statement followed by method call node and the method is captured by the advice and node n_1 executes before node n_2 executes.

In Figure 8, edge $(s14, a73)$ is a weaving edge.

The brief pseudocode for constructing the CFADG for a feature-oriented program is given below, and the complete algorithm is given in Algorithm 8 in Appendix A.

CFADG construction Algorithm

(1) For each mixin layer

(a) For each mixin
i. Create mixin entry vertex
ii. For each method
A. Compute control and data dependences.
B. Construct PDG using control & data dependence edges.
iii. For each method call
A. Create actual parameter vertices.
iv. For each method definition
A. Create method entry vertex.
B. Create formal parameter vertices.
v. Construct MxDG by connecting all PDGs through method call edges, parameter edges and summary edges and connecting each method vertex to mixin start vertex through mixin membership edges.

(b) Construct SDG by connecting all MxDGs through method call edges, parameter edges.

(2) For each aspect

(a) Create aspect entry vertex.
(b) For each advice
i. Create advice start vertex.
ii. Compute control and data dependences.
iii. Construct ADG using control & data dependence edges.
(c) For each introduction
i. Create introduction start vertex.
ii. If introduction is a field then
Do not create any dependence graph.
Else if introduction is a method then
Construct IDG using control and data dependence edges.
(d) For each pointcut
i. Create pointcut start vertex.
ii. Construct PtDG.
(e) Construct AsDG by connecting advice start vertices, introduction start vertices, pointcut start vertices to aspect start vertex through aspect membership edges.

(3) Remove mixin membership edges, aspect membership edges, mixin start vertices, and aspect start vertices.

(4) Connect all SDGs through refinement edges, mixin call edges, mixin data dependence edges, and mixin return dependence edges.

(5) Connect all AsDGs to all SDGs through weaving and aspect data dependence edges.

4 Feature-aspect node-marking dynamic slicing (FANMDS) algorithm

In this section, we present our proposed algorithm for computing dynamic slices of feature-oriented programs using CFADG. We have named our algorithm *Feature-Aspect Node-Marking Dynamic Slicing* (FANMDS) algorithm as it is based on marking and unmarking the nodes of CFADG. Before presenting our FANMDS algorithm, we first introduce some definitions which will be used in our algorithm.

4.1 Definitions

Definition 1: Defn(v): Let v be a variable or an object in program P. A node u in the CFADG is said to be $Defn(v)$ node if u corresponds to a definition statement that defines a value to variable v or u represents a statement that creates object v.
In the CFADG given in Figure 8, nodes $s23$, and $s24$ represent *Defn(answer)* nodes in the method *logten()* in mixin *calc* in *log* mixin layer.
Definition 2: DefnSet(v): The set of all $Defn(v)$ nodes is referred to as $DefnSet(v)$.
In the CFADG given in Figure 8, $DefnSet(answer) = \{s23, s24\}$ in the method *logten()* in mixin *calc* in *log* mixin layer.
Definition 3: RecDefn(v): For each variable v, $RecDefn(v)$ represents the node corresponding to the most recent definition of v with respect to some point s in an execution.

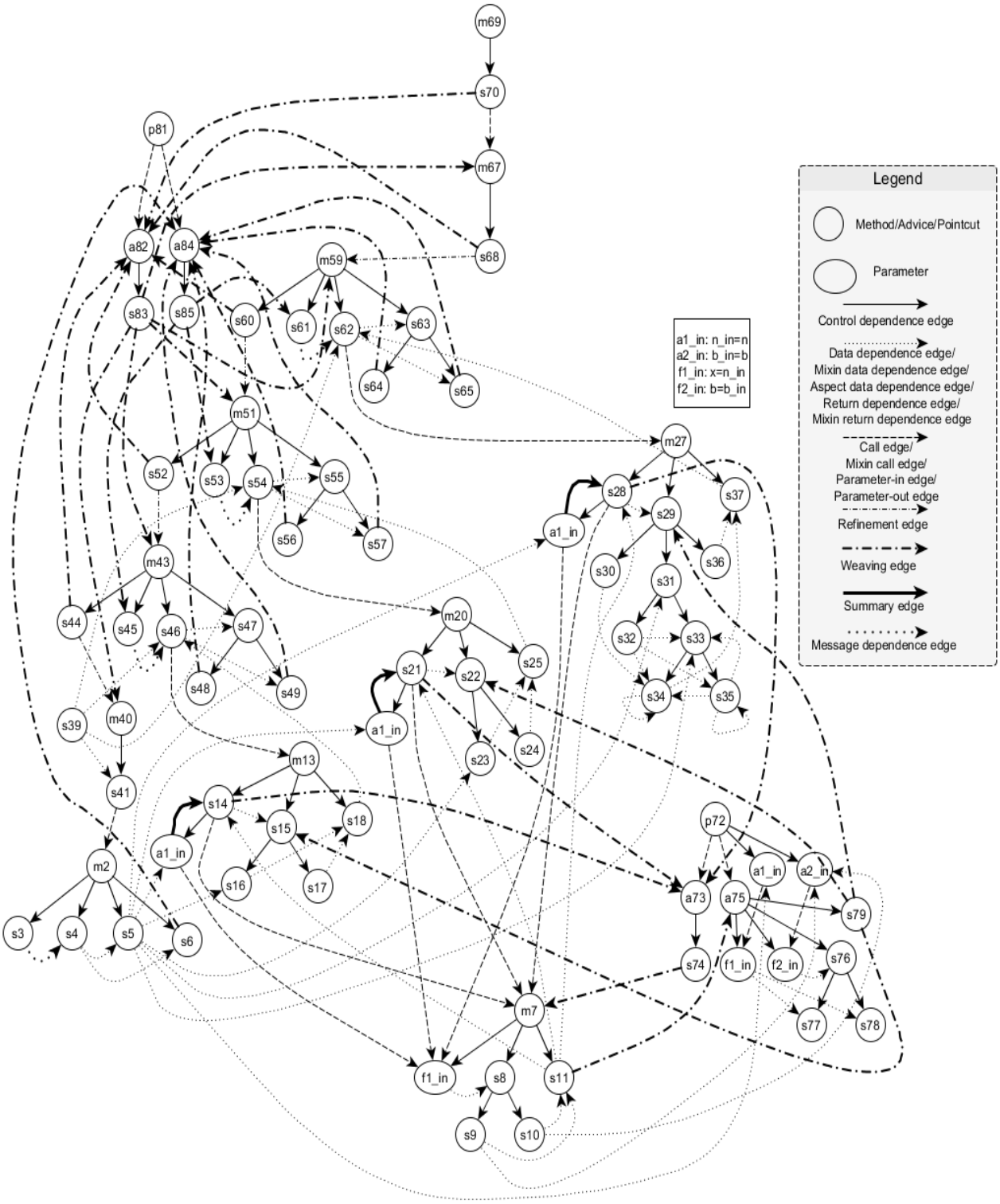


Figure 8: Composite Feature-Aspect Dependence Graph (CFADG) for the program given in Figure 4

In the CFADG of Figure 8, $RecDefn(i)$ is at statement $s32$ before while loop and it is at statement $s35$ during execution of while loop.

Definition 4: Usage(v): Let v be a variable or an object in program P. A node u in the CFADG is said to be $Usage(v)$ node if u represents a statement that uses the variable v or u represents a statement that uses the object v to call a method on that object or to assign the object v with another object.

In the CFADG given in Figure 8, nodes $s47$, and $s49$ represent $Usage(r)$ nodes. Similarly, node $s77$, and $s78$ are $Usage(n)$ nodes.

Definition 5: UsageSet(v): The set of all $Use(v)$ nodes is referred to as $UsageSet(v)$.

In the CFADG given in Figure 8, and $UsageSet(r) = \{s47, s49\}$, $UsageSet(n) = \{s77, s78\}$.

4.2 Overview of FANMDS algorithm

Before execution of a feature-oriented program FP, the features required for composition and the aspects to be captured are selected. Then, the selected features are composed and selected aspects are weaved. The CFADG is constructed statically only once based on the composition of selected features and weaving of selected aspects. The program is executed for a specified input. The executed nodes in CFADG are marked and unmarked during program execution depending upon the arise and cease of dependences respectively. When a statement executes a *Super()* node, it is marked by the algorithm. Also the corresponding method entry node, the associated actual and formal parameter nodes are marked. When there is an invocation of a method, the corresponding call node, the corresponding method entry node, the associated actual and formal parameter nodes are also marked. Whenever a pointcut is executed, the corresponding advice nodes are marked. When an advice is executed, the corresponding formal parameter nodes are marked. During execution, the dynamic slice of each executed statement is computed. After execution of each node and computation of dynamic slice at that node, the algorithm unmarks it.

Let $dyn\_slice(u)$ denote the dynamic slice with respect to the most recent execution of node u. Let (e_1, u), $(e_2, u), \ldots, (e_k, u)$ be all the marked predecessor nodes of u in the CFADG after execution of node u. The dynamic slice with respect to the present execution of node u is computed as
$dyn\_slice(u) = \{u, e_1, e_2, \ldots, e_k\} \cup dyn\_slice(e_1) \cup dyn\_slice(e_2) \cup \ldots \cup dyn\_slice(e_k)$.

Our FANMDS algorithm computes the dynamic slice with respect to the specified slicing criterion by simply looking up the corresponding dyn_slice computed during run-time. Below, we present the pseudocode of our FANMDS algorithm in brief. Algorithm 9 in Appendix B presents our FANMDS algorithm in detail.

Feature-Aspect Node-Marking Dynamic Slicing (FANMDS) Algorithm

(1) CFADG Construction: Construct the CFADG for the given feature-oriented program with aspect-oriented extensions, statically only once.

(2) Initialization: Do the followings before each execution of FP.

(a) Unmark all nodes of CFADG.

(b) Set $dyn\_slice(u) = \phi$ for every node u.

(c) Set $RecDefn(v) = NULL$ for every variable v of the program FP.

(3) Run time updations: Execute the program for the given set of input values and carry out the followings after each statement s of the program FP is executed. Let node u in CFADG corresponds to the statement s in the program FP.

(a) For every variable v used at node u,
Update $dyn\_slice(u) = \{u, e_1, e_2, \ldots, e_k\} \cup dyn\_slice(e_1) \cup dyn\_slice(e_2) \cup \ldots \cup dyn\_slice(x_k)$ where $e_1, e_2, \ldots, e_k$ are the marked predecessor nodes of u in CFADG.

(b) If u is $defn(v)$ node, then

i. Unmark the node $RecDefn(v)$.

ii. Update $RecDefn(v) = u$.

(c) Mark node u.

(d) If u is a *method call* node or `new` operator node or *polymorphic* node or *mixin call* node, then

i. Mark node u.

ii. Mark the associated *actual-in* and *actual-out* nodes corresponding to the present execution of u.

iii. Mark the corresponding *method entry* node for the present execution of u.

iv. Mark the associated *formal-in* and *formal-out* parameter nodes.

(e) If u is a *Super()* method node

i. Mark node u.

ii. Mark the associated *actual-in* and *actual-out* nodes corresponding to the present execution of u.

iii. Mark the corresponding *method entry* node present in the parent mixin layer for the present execution of u.

iv. Mark the *formal-in* and *formal-out* parameter nodes associated with the *method entry* node.

(f) If u is a *pointcut* node

i. Mark node u.

ii. Mark the corresponding *advice* nodes for present execution of u.

(g) If u is an *advice* node

i. Mark node u.

ii. Mark the *formal-in* and *formal-out* parameter nodes associated with the *advice* node.

(h) If u is an *introduction* node such that u is a method

i. Mark node u.

ii. Mark the *formal-in* and *formal-out* parameter nodes.

(i) If u is an *introduction* node such that u is a field

i. Mark node u.

ii. Mark the node that defines a value to u for the current execution of u.

iii. Mark the node that uses the value of u for the current execution of u.

(4) Slice Look Up

(a) For a given slicing command
$< u, v >$, do

i. Look up $dyn\_slice(u)$ for variable v for the content of the slice.

ii. Map the Java statements included in the computed dynamic slice to the corresponding composed Jak statements to get the final dynamic slice

iii. Display the resulting slice.

(b) If the program has not terminated, go to Step 3.

Working of the Algorithm

The working of FANMDS algorithm is illustrated through an example. Consider the feature-oriented program given in Figure 3 and the selected features for composition and aspects given in Figure 1 and Figure 2 respectively. After the composition of the selected features, the files that are generated are depicted in Figure 4. The corresponding CFADG is shown in Figure 8. During the initialization step, our algorithm first unmarks all the nodes of the CFADG and sets $dyn\_slice(u) = \phi$ for every node u of the CFADG. Now, for the input data $n = 5$, the program will execute the statements $m69$, $s70$, $p81$, $a82$, $s83$, $m67$, $s68$, $a82$, $s83$, $m59$, $s60$, $a82$, $s83$, $m51$, $s52$, $a82$, $s83$, $m43$, $s44$, $a82$, $s83$, $s39$, $m40$, $s41$, $m2$, $s3$, $s4$, $s5$, $s6$, $a84$, $s85$, $s45$, $s46$, $m13$, $s14$, $p72$, $a73$, $s74$, $m7$, $s8$, $s10$, $s11$, $a75$, $s76$, $a78$, $s79$, $s15$, $s16$, $s18$, $s47$, $s49$, $a84$, $s85$, $s53$, $s54$, $m20$, $s21$, $a73$, $s74$, $m7$, $s8$, $s10$, $s11$, $a75$, $s76$, $s78$, $s79$, $s22$, $s23$, $s25$, $s55$, $s57$, $a84$, $s85$, $s61$, $s62$, $m27$, $s28$, $a73$, $s74$, $m7$, $s8$, $s10$, $s11$, $a75$, $s76$, $s78$, $s79$, $s29$, $s30$, $s31$, $s32$, $s33$, $s34$, $s35$, $s37$, $s63$, $s65$, $a84$, $s85$, $a84$, $s85$ in order. So, our FANMDS algorithm marks these nodes. Our algorithm also marks the associated actual parameter vertices at the calling method and the formal parameter vertices at the called method.

Now, the dynamic slice is to be computed with respect to variable n at statement $s78$, i.e., with respect to slicing criterion $< \{n = 5\}, s78, n >$ by traversing the CFADG in backward manner. According to the FANMDS algorithm, the dynamic slice with respect to variable n at statement $s78$ is given by the expression
$dyn\_slice(s78) = \{s78, s76, a75 \rightarrow f1\_in\} \cup dyn\_slice(s76)$
$\cup\, dyn\_slice(a75 \rightarrow f1\_in)$.
By evaluating the above expression in a recursive manner, we get the final dynamic slice consisting of the statements corresponding to the nodes $m2$, $s3$, $s4$, $s5$, $m7$, $s8$, $s10$, $s11$, $m13$, $s14$, $m20$, $s21$, $m27$, $s28$, $s39$, $m40$, $s41$, $m43$, $s44$, $s45$, $s46$, $s47$, $s49$, $m51$, $s52$, $s53$, $s54$, $s55$, $s57$, $m59$, $s60$, $s61$, $s62$, $m67$, $s68$, $m69$, $s70$, $p72$, $a73$, $s74$, $a75$, $s76$, $s78$, $p81$, $a82$, $s83$, $a84$, $s85$. These are indicated as bold vertices in Figure 9 and the corresponding statements are indicated in rectangular boxes in Figure 10. Similarly, dynamic slice with respect to any slicing criterion can be computed using FANMDS algorithm.

5 Implementation

This section briefly describes the implementation of FANMDS algorithm. A dynamic slicing tool has been developed to implement the algorithm which has been named *feature-aspect dynamic slicing tool* (FADST). Figure 11 depicts the architectural design of the slicing tool FADST. The working of our slicing tool is depicted in Figure 12, through a flow chart.

In Figure 11, the executable components are depicted in rectangular boxes and the passive components are depicted in ellipses. First, the features required to compose and aspects to be captured are selected. The selected features, the selected aspects, and the slicing criterion consisting of input, line number, and variable are provided to FADST through the *Graphical User Interface* (GUI) component. The *Dynamic Slicer* component interacts with the GUI component and produces the required result as output back to GUI. The *AHEAD composer* [2] composes the selected features to generate a set of Java programs. These Java programs and the selected aspects are fed to *AspectJ composer*. AspectJ composer weaves the aspects at the appropriate join points, and the result is a composed AspectJ program. The *lexical analyzer* component reads the composed AspectJ program and generates tokens from these programs. Upon encountering a useful token, the lexical analyzer component returns the token along with its type to the *parser and semantic analyzer* component. The *parser and semantic analyzer* component takes the token and analyzes it using the grammatical rules designed for the input programs.

The *code instrumentor* component instruments the composed AspectJ programs. The classes are instrumented with line numbers prefixed with c, the aspects are instrumented with line numbers prefixed with as, the methods are instrumented with line numbers prefixed with m, the pointcuts are instrumented with line numbers prefixed with p, the advices are instrumented with line numbers prefixed with a, and the statements containing assignments, computations, predicates are instrumented with line numbers prefixed with s.

The *CFADG constructor* component constructs the CFADG using the required program analysis information such as type of statement, sets of variables defined or used at a statement etc. The *dynamic slicer* component implements the FANMDS algorithm. We have used Java language for our implementation. A compiler writing tool, *ANTLR* (Another Tool for Language Recognition)[5], has been used for *lexical analyzer*, *parser and semantic analyzer* components of FADST.

An adjacency matrix `adj[][]` has been used for storing the CFADG with respect to a selected composition of features of the given feature-oriented program.

Arrays are used to store the sets *Defn(v)*, *Usage(v)*, *RecDefn(v)*, and *dyn_slice(u)*.

[5] `www.antlr.org`

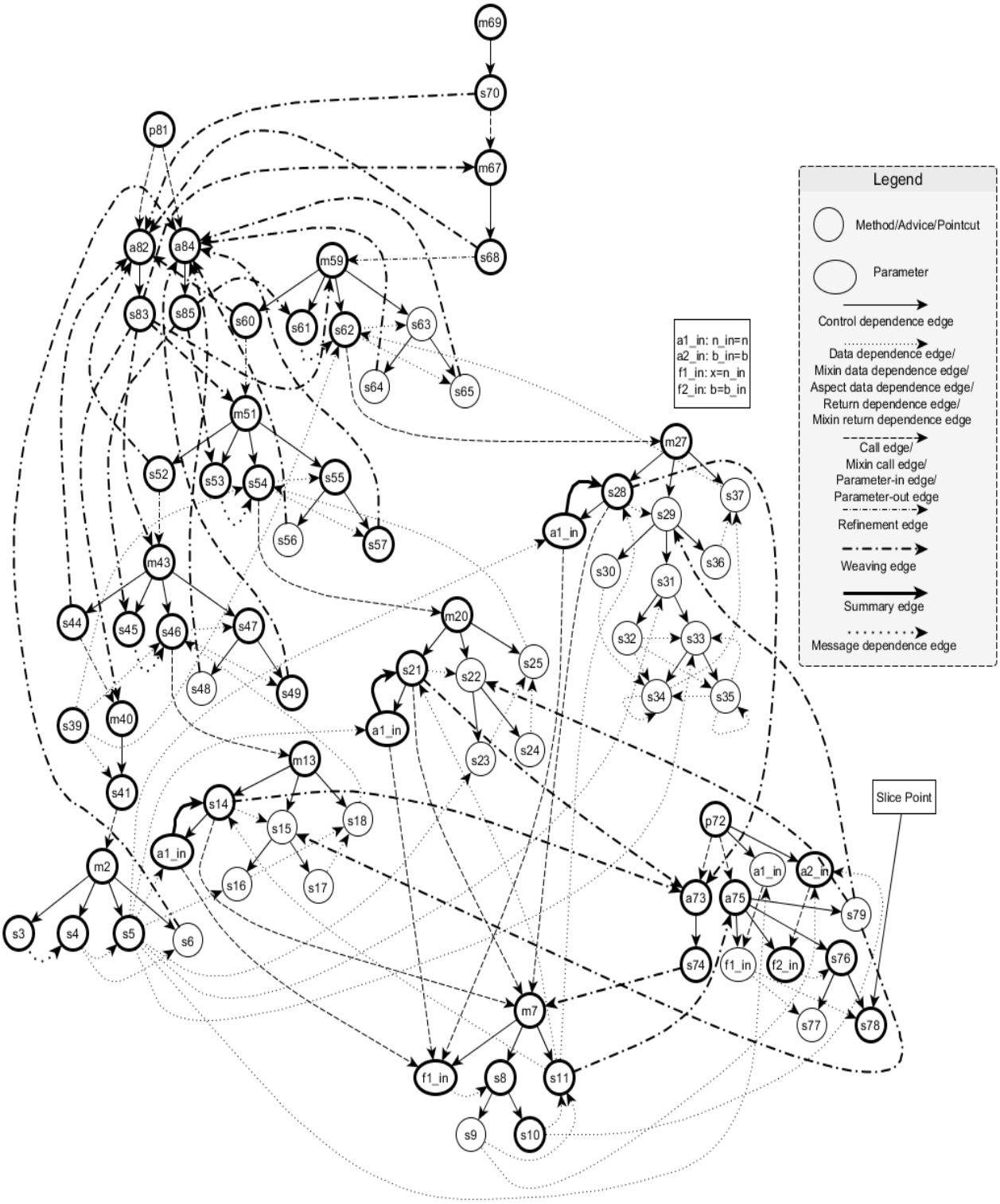


Figure 9: CFADG showing statements included in dynamic slice as bold nodes

```
import java.util.Scanner;
import java.lang.Math;
SoUrCe RooT Base "../features/Base/calc.jak";
abstract class calc$$Base {
      double n;
      boolean result;
      void enter(){
            System.out.println("Enter a number");
            Scanner sc=new Scanner(System.in);
            n=sc.nextDouble();
            sc.close();
      }
      boolean isnegs(double x){
            if(x<0)
                  result=true;
            else
                  result=false;
            return result;
      }
}
SoUrCe  sqrt "../features/sqrt/calc.jak";
abstract class calc$$sqrt extends  calc$$Base  {
      double squrt(){
            double answer;
            result=isnegs(n);
            if(!result)
                  answer=Math.sqrt(n);
            else
                  answer=-1;
            return answer;
      }
}
SoUrCe  log "../features/log/calc.jak";
abstract class calc$$log extends  calc$$sqrt  {
      double logten(){
            double answer;
            result=isnegs(n);
            if(!result)
                  answer=Math.log10(n);
            else
                  answer=-1;
            return answer;
      }
}
SoUrCe  fact "../features/fact/calc.jak";
public class calc extends  calc$$log  {
      long fact(){
            long answer;
            result=isnegs(n);
            if(!result){
                  answer=1;
                  if(n>0){
                  int i=1;
                  while(i<=n){
                        answer=answer*i;
                        i++;
                  }
            }
            }
            else
                  answer=-1;
            return answer;
      }
}
```

(a) calc.jak

```
SoUrCe RooT Base "../features/Base/test.jak";
abstract class test$$Base {
      static calc c=new calc();
      static void printtest(){
      c.enter();
      }
}
SoUrCe  sqrt "../features/sqrt/test.jak";
abstract class test$$sqrt extends  test$$Base  {
      static void printtest(){
            Super().printtest();
            System.out.println("Finding square root");
            double r=c.squrt();
            if(r==-1)
                  System.out.println("not valid number");
            else
                  System.out.println("Square Root is "+r);
      }
}
SoUrCe  log "../features/log/test.jak";
abstract class test$$log extends  test$$sqrt  {
      static void printtest(){
            Super().printtest();
            System.out.println("Finding logarithm");
            double r=c.logten();
            if(r==-1)
                  System.out.println("not valid number");
            else
                  System.out.println("Logarithm base 10 is "+r);
      }
}
SoUrCe  fact "../features/fact/test.jak";
abstract class test$$fact extends  test$$log  {
static void printtest(){
      Super().printtest();
      System.out.println("Finding factorial");
      long r=c.fact();
      if(r==-1)
            System.out.println("not valid number");
      else
            System.out.println("Factorial is "+r);
}
}
SoUrCe  Print "../features/Print/test.jak";
public class test extends  test$$fact  {
static void printtest(){
      Super().printtest();
}
public static void main(String[] args){
      test.printtest();
}
}
```

(b) test.jak

```
as71 public aspect error {
p72    pointcut pc(double n):call (boolean *.isnegs(double)) && args(n);
a73    before (double n):pc(n){
s74          System.out.println("Entering Error Handling Aspect");
      }
a75    after (double n) returning (boolean b):pc(n){
s76          if(b==true)
s77                System.out.println(n+" is negative");
           else
s78                System.out.println(n+" is positive");
s79          System.out.println("Exiting Error Handling Aspect");
      }
   }
```

(c) error.aj

```
as80 public aspect print {
p81    pointcut pc():call (void *.printtest());
a82    before ():pc(){
s83          System.out.println("Entering Printing Aspect");
      }
a84    after () :pc(){
s85          System.out.println("Exiting Printing Aspect");
      }
   }
```

(d) print.aj

Figure 10: Dynamic slice with respect to slicing criterion $< \{n = 5\}, s78, n >$ depicted as statements in rectangular boxes

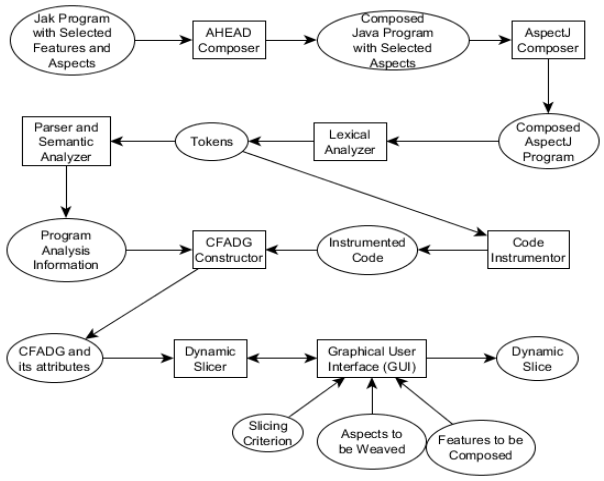


Figure 11: Architecture of the slicing tool

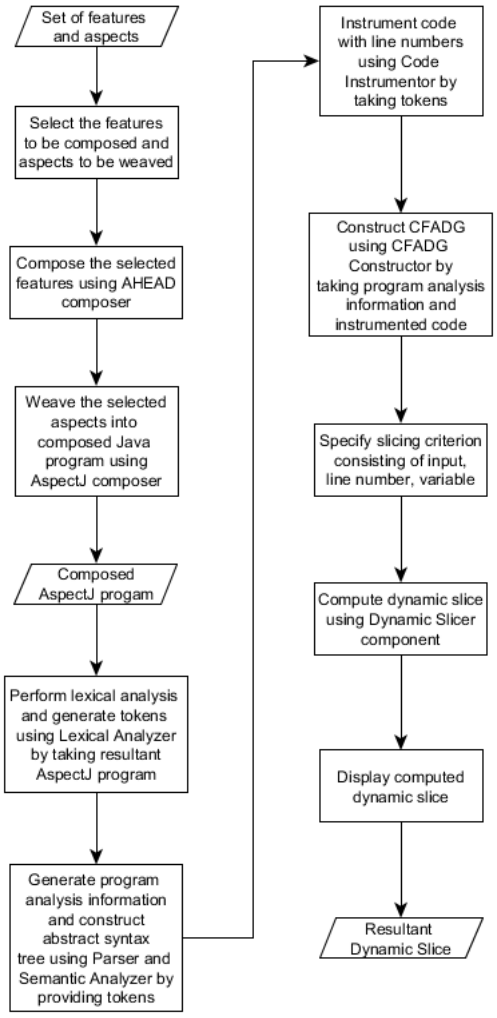


Figure 12: Flowchart for working of the slicing tool given in Figure 11

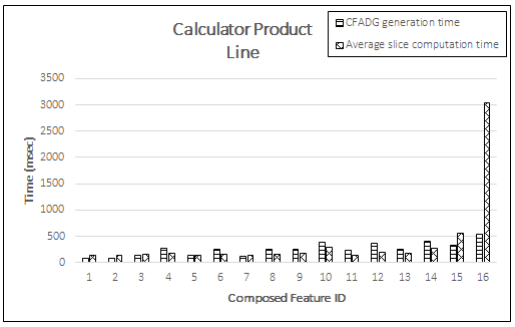


Figure 13: CFADG generation time and Average slice computation time for Calculator Product Line

5.1 Case studies and experimental results

We have applied our algorithm to some product lines [6,7]. We have also taken some open-source Java programs[8] [9] [10]. We have developed few product lines by identifying various features and converting these available Java programs into corresponding Jak programs. It may be noted that Jak is one of the feature-oriented programming languages. We have also taken the models of few product lines (such as calculator product line, stack product line, graph product line) from the work of different researchers [45, 44, 43, 46] and developed the corresponding Jak programs. These may be considered as representative feature-oriented programs with aspect-oriented extensions. In all the product lines, we have identified the aspects that are scattered throughout the program. The product lines we have taken as our case studies have various features and aspects which can be used for composing a variety of software product lines. We have taken fifteen product lines as our case studies. The characteristics of our software product lines are depicted in Table 1. These programs are executed for different compositions of features with different aspects weaved for different inputs. Also, the algorithm has been tested for different slicing criteria for different compositions of features and different inputs.

The CFADG generation time and average slice computation time for various compositions of features in different product lines are depicted in Figures 13–27.

It can be inferred from Figures 13–27 that different compositions of features result in different slice computation times. The aspects weaved at more number of join points take more time than the aspects weaved at less number of join points. For example, in *Calculator Product Line* (CPL), the number of join points where the aspect *Print* is weaved is more than that of aspect *Error*. That's why the slice computation time for the program where *Print* aspect

[6] http://spl2go.cs.ovgu.de/projects

[7] http://www.infosun.fim.uni-passau.de/spl/apel/fh

[8] http://www.sanfoundry.com/java-program-implement-avl-tree/

[9] http://www.geeksforgeeks.org/avl-tree-set-2-deletion/

[10] https://ankurm.com/implementing-singly-linked-list-in-ja

Table 1: List of Case Study Software Product Lines

Sl. No.	Name of Product Line	Description	Total No. of Features Supported		Total No. of Aspects weaved		Total No. of lines in composed AspectJ file
			Mandatory	Optional	Mandatory	Optional	
1	Calculator Product Line (CPL)	Calculates the factorial, square root and logarithmic value with base 10 of a number.	2 (Base, Print)	3 (sqrt, log, fact)	1 (Error)	1 (Print)	85
2	Stack Product Line (SPL)	Models variations of stacks.	1 (Stack)	3 (Counter, Lock, Undo)	1 (Size)	1 (Top)	130
3	Graph Product Line	Models variability for different types of graphs such as colored, weighted etc.	1 (Base)	4 (Weight, Color, Recursive, PrintHeader)	1 (Print)	1 (AddNode)	150
4	AVL Tree Product Line (AVLTPL)	Simulates the various operations on an AVL tree.	2 (Base, Display)	4 (Insert, Delete, Count, Search)	1 (ComputeHeight	1 (Rotate)	315
5	Single Linked List Product Line (SLLPL)	Simulates the various operations on a single linked list.	2 (Base, Display)	7 (InsertBegin, InsertEnd, InsertAfter, DeleteBegin, DeleteEnd, DeleteAfter, Count)	2 (Insert)	0	195
6	DesktopSearcher	Program for indexing and content based searching in files	7	9	3	4	2516
7	TankWar	A Game	12	19	6	7	3746
8	GPL	Graph and algorithm library	12	24	5	12	801
9	MobileMedia	MobileMedia is a Software Product Line (SPL) that manipulates photo, music, and video on mobile devices. It is a multimedia management for phones	10	37	5	10	4669
10	Digraph	A library for representing and manipulating directed graph structures. Beside basic graphs, it supports various operations such as removal, traversal, and transposition, implemented as optional features.	1	3	1	2	1733
11	Elevator	Simulates various operations of an Elevator	3	3	2	1	873
12	Vistex	VisTex is a product line that features graphical manipulation of graphs and their textual representation. It is designed to be easily extendible for specific graph-based applications such as UML.	5	11	2	3	1890
13	Violet	Graphical model editor	15	73	6	9	9203
14	Notepad	Graphical text editor	6	7	3	3	1672
15	PkJab	Instant messaging client for Jabber.	3	5	2	2	3963

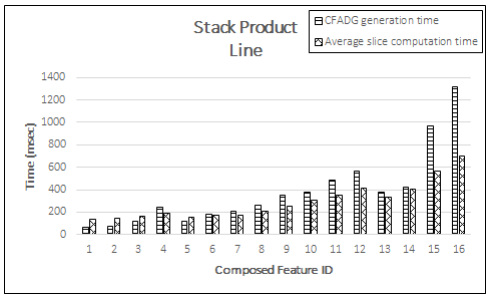


Figure 14: CFADG generation time and Average slice computation time for Stack Product Line

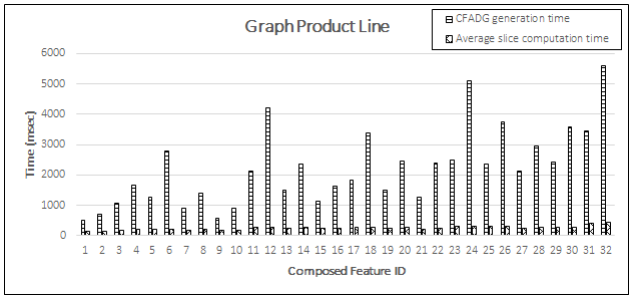


Figure 15: CFADG generation time and Average slice computation time for Graph Product Line

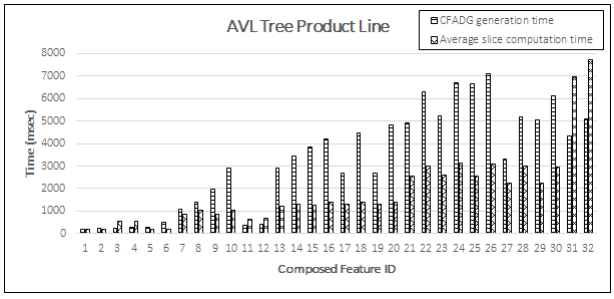


Figure 16: CFADG generation time and Average slice computation time for AVL Tree Product Line

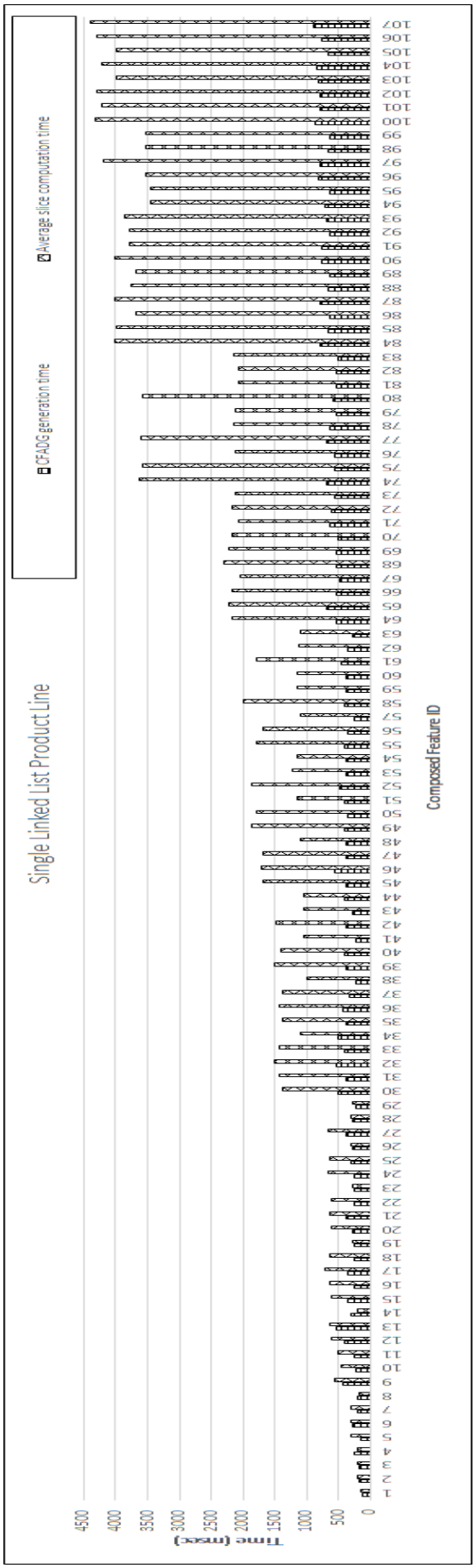


Figure 17: CFADG generation time and Average slice computation time for Single Linked List Product Line

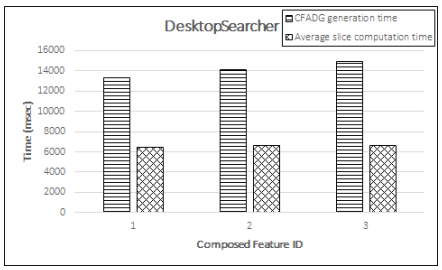


Figure 18: CFADG generation time and Average slice computation time for DesktopSearcher

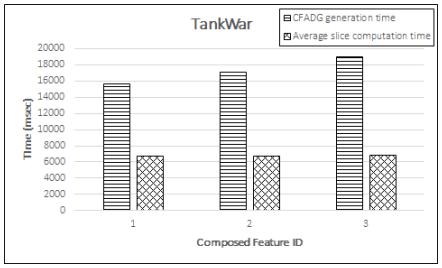


Figure 19: CFADG generation time and Average slice computation time for TankWar

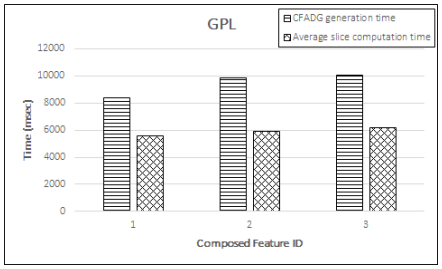


Figure 20: CFADG generation time and Average slice computation time for GPL

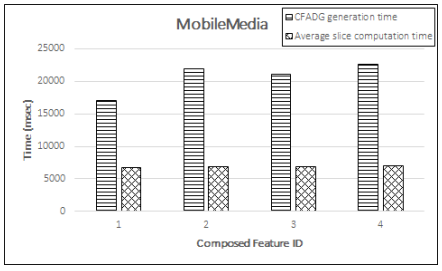


Figure 21: CFADG generation time and Average slice computation time for MobileMedia

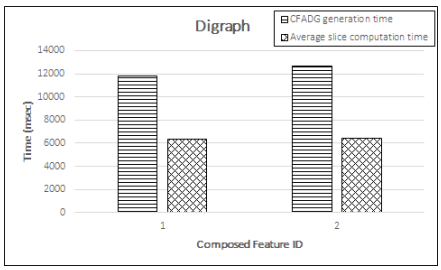


Figure 22: CFADG generation time and Average slice computation time for Digraph

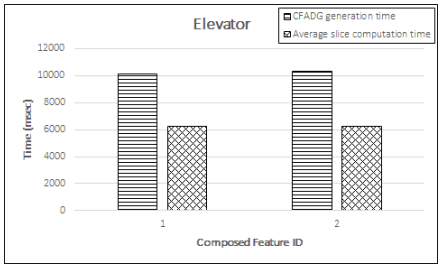


Figure 23: CFADG generation time and Average slice computation time for Elevator

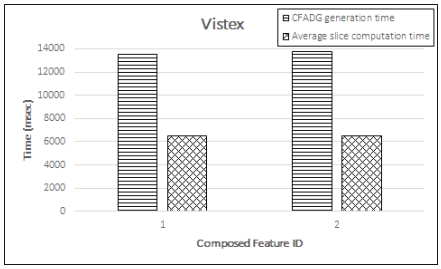


Figure 24: CFADG generation time and Average slice computation time for Vistex

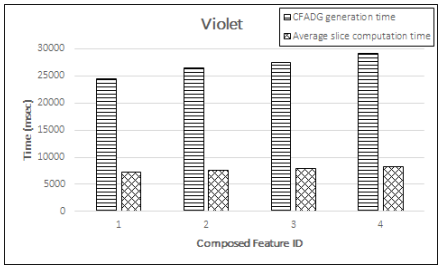


Figure 25: CFADG generation time and Average slice computation time for Violet

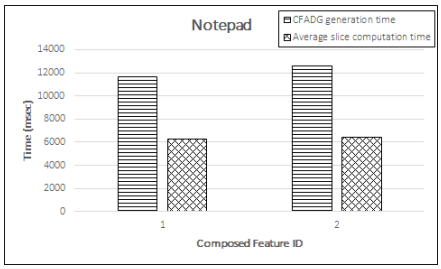


Figure 26: CFADG generation time and Average slice computation time for Notepad

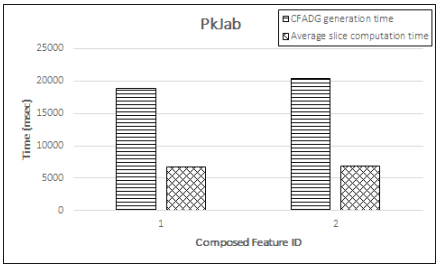


Figure 27: CFADG generation time and Average slice computation time for PkJab

is weaved is more than that of the program where *Error* aspect is weaved. The features containing more number of loops take more time. The composed features containing less number of executable statements take less time compared to those containing more number of executable statements.

6 Comparison with related work

Several works have been carried out on slicing of procedure-oriented programs [34, 32, 33, 30, 47], object-oriented programs [11, 21, 39, 22, 15], aspect-oriented programs [37, 9, 10, 16, 18, 23]. But very, few work have been carried out on slicing of feature-oriented programs [35].

Zhao [9] was the first to develop a two-phase slicing algorithm to compute static slices of aspect-oriented programs. Later, Zhao et al. [10] developed an efficient algorithm for constructing system dependence graph for aspect-oriented programs. Ray et al. [16] developed an algorithm to compute dynamic slices of aspect-oriented programs by constructing *Aspect System Dependence Graph* (AOSG). They had introduced a new logical node called *C-node* to capture communication dependencies among the non-aspect code and aspect code. They had also introduced a new arc called *aspect-membership arc* to connect the dependence graphs of the non-aspect code and aspect code. They had not shown the actual parameters in the pointcuts. Singh et al. [18] proposed a method to compute slices depending upon the slice point location in the program. Their computed slice was an executable slice. Munjal et al. [23] automated the generation of system dependence graphs (SDG) for aspect-oriented programs by analysing the bytecode of aspect-oriented programs. Then, they proposed a three-phase slicing algorithm to compute static slices using the intermediate graph for a given aspect-oriented program. All the above works [9, 15, 16, 18, 23] have not considered feature-oriented programs.

Apel et al. [3] presented a novel language for FOP in C++ namely FeatureC++. They also mentioned few problems of FOP languages. Apel et al. [4] demonstrated FeatureC++ along with its adaptation to Aspect-Oriented Programming (AOP) concepts. They discussed the use of FeatureC++ in solving various problems related to incremental software development using AOP concepts. They also discussed the weaknesses of FOP for modularization of crosscutting concerns. Apel et al. [5] discussed the limitations of crosscutting modularity and the missing support of C++. They also focused on solutions for ease evolvability of software. Batory [2] presented basic concepts of FOP and a subset of the tools of the *Algebraic Hierarchical Equations for Application Design* (AHEAD) tool suite. Apel et al. [7] presented an overview of feature-oriented software development (FOSD) process. They had identified various key issues in different phases of FOSD. Thum et al. [6] developed an open source framework for FOSD namely FeatureIDE that supported all phases of FOSD along with support for feature-oriented programming languages, and delta-oriented programming languages, aspect-oriented programming languages. Pereira et al. [20] discussed the findings of SPL management tools from a Systematic Literature Review (SLR). These works [7, 5, 3, 4, 2, 20, 6] discussed only the programming and development aspects of FOP and did not consider the slicing aspects. We have presented a technique for dynamic slicing of feature-oriented programs with aspect-oriented extensions using Jak as the FOP language.

Very few work have been carried out on slicing of feature-oriented programs [35]. Sahu et al. [35] suggested a technique to compute dynamic slices of feature-oriented programs. Their technique first composed the selected features of feature-oriented programs. Then, they used an execution trace file and a dependence-based program representation namely *dynamic feature-oriented dependence graph* (DFDG). The dynamic slice was computed by traversing DFDG in breadth-first or depth-first manner and mapping the traversed vertices to the program statements. They had missed some of the dependences such as mixn call edge, refinement edge, and mixin return dependence edge, etc. that might arise in feature-oriented programs. The drawback of their approach is the use of execution trace file which may lead to more slice computation time. They had not considered the aspect-oriented extensions of feature-oriented programs. In our approach, we have not used any execution trace file. Usually, the execution trace file is used to store the execution history of each executed statement for a given input. Much time is required to store and retrieve the executed statements. The statements are then used for calculation of dynamic slice for each statement. Thus, extra time is required to perform I/O operations on an execution trace

file. We do not use any execution trace file. During execution of the program for a given input, the dynamic slice for each statement is computed by marking and unmarking process. Thus, there is no requirement of any execution trace file for storing the executed statements. So, our proposed approach does not take any extra time to read from or write into the execution trace file, thereby reducing the slice extraction time. Also, we have considered the aspects that are scattered throughout the code. Our algorithm does not create any new node in the intermediate representation CFADG during runtime. This results in faster computation of slices.

7 Conclusion and future work

We have presented an approach to compute dynamic slices of feature-oriented programs with aspect-oriented extensions. The features required for composition are first selected and composed using *Algebraic Hierarchical Equations for Application Design (AHEAD)* composer. Then, the aspects are weaved into the generated composed Java program using *AspectJ* composer to produce the resultant AspectJ program. The intermediate dependence based representation of the program containing Jak code and AspectJ code is constructed and it is called *Composite Feature-Aspect Dependence Graph (CFADG)*. The program is executed for an input. During execution, the nodes of CFADG are marked and unmarked according to our *feature-aspect node marking dynamic slicing* (FANMDS) algorithm. We have developed a tool to implement our FANMDS algorithm and named it FADST. Our tool FADST computes the dynamic slices and the average slice extraction times for various compositions of features and aspects weaved for various product lines. Currently, our tool is able to handle various compositions for few product lines with few aspects captured. Also, current evaluation only uses primitive feature-oriented programs. In future, we will extend our tool to handle more number of product lines with more number of compositions.

Our algorithm may easily be extended to compute dynamic slices of other feature-oriented languages like FeatureC++, FeatureRuby, FeatureHouse, Fuji, etc. Also, the extension of the algorithm can be used to compute conditioned slices, amorphous slices for feature-oriented programs with various aspects captured. We will also find out the differences in the performance of different aspects.

References

[1] Christian Prehofer (1997) Feature-Oriented Programming: A Fresh Look at Objects, *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP)*, Springer, Berlin, Heidelberg, pp. 419–443. https://doi.org/10.1007/bfb0053389

[2] Don Batory (2006) A Tutorial on Feature–Oriented Programming and the AHEAD Tool Suite, *Proceedings of the 2005 International Conference on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, Springer–Verlag, Berlin, Heidelberg, pp. 3–35. https://doi.org/10.1007/11877028_1

[3] Sven Apel and Thomas Leich and Marko Rosenmuller and Gunter Saake (2005) FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++, Tech. rep., Department of Computer Science, Otto-von-Guericke University, Magdeburg, Germany.

[4] Sven Apel and Thomas Leich and Marko Rosenmuller and Gunter Saake (2005) FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE'05)*, Springer, pp. 125–140. https://doi.org/10.1007/11561347_10

[5] Sven Apel and Thomas Leich and Marko Rosenmuller and Gunter Saake (2005) Combining Feature-Oriented and Aspect-Oriented Programming to Support Software Evolution, *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE)*, School of Computer Science, University of Magdeburg, July, pp. 3–16.

[6] Thomas Thum and Christian Kastner and Fabian Benduhn and Jens Meinicke and Gunter Saake and Thomas Leich (2014) FeatureIDE: An extensible framework for feature-oriented software development, *Science of Computer Programming*, 79, pp. 70–85. https://doi.org/10.1016/j.scico.2012.06.002

[7] Sven Apel and Christian Kastner (2009) An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5), pp. 49–84, July–August. https://doi.org/10.5381/jot.2009.8.5.c5

[8] Gregor Kiczales and John Irwin and John Lamping and Jean Marc Loingtier and Cristiana Videira Lopes and Chris Maeda and Anurag Mendhekar (1997) Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June, pp. 220–242. https://doi.org/10.1007/bfb0053381

[9] Jianjun Zhao (2002) Slicing Aspect-Oriented Software, *Proceedings of 10th International Workshop on Program Comprehension*, pp. 251–260, June. https://doi.org/10.1109/wpc.2002.1021346

[10] Jianjun Zhao and Martin Rinard (2003) System Dependence Graph Construction for Aspect-Oriented Programs. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, USA, March.

[11] Loren Larsen and Mary Jean Harrold (1996) Slicing Object-Oriented Software, *Proceedings of 18th International Conference on Software Engineering*, pp. 495–505, March. https://doi.org/10.1109/icse.1996.493444

[12] Timon Ter Braak (2006) Extending Program Slicing in Aspect–Oriented Programming With Inter–Type Declarations, *5th TSConIT Program*, June.

[13] Aspect Oriented Programming. www.wikipedia.org.

[14] Hiralal Agrawal and Joseph R. Horgan (1990) Dynamic Program Slicing, ACM SIGPLAN Notices, *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation PLDI'90*, 25(6), pp. 246–256, June. https://doi.org/10.1145/93542.93576

[15] Durga Prasad Mohapatra and Rajib Mall and Rajeev Kumar (2004) An Edge Marking Technique for Dynamic Slicing of Object-Oriented Programs, *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*. https://doi.org/10.1109/cmpsac.2004.1342806

[16] Abhisek Ray and Siba Mishra and Durga Prasad Mohapatra (2012) A Novel Approach for Computing Dynamic Slices of Aspect-Oriented Programs, *International Journal of Computer Information Systems*, 5(3),pp. 6–12, September.

[17] Abhisek Ray and Siba Mishra and Durga Prasad Mohapatra (2013) An Approach for Computing Dynamic Slice of Concurrent Aspect-Oriented Programs *International Journal of Software Engineering and Its Applications*, 7(1), pp. 13–32, January.

[18] Jagannath Singh and Durga Prasad Mohapatra (2013) A Unique Aspect-Oriented Program Slicing Technique, *Proceedings of International Conference on Advances in Computing, Communications and Informatics (ICACCI'13)*, pp. 159–164. https://doi.org/10.1109/icacci.2013.6637164

[19] Jagannath Singh and Dishant Munjal and Durga Prasad Mohapatra (2014) Context Sensitive Dynamic slicing of Concurrent Aspect-Oriented Programs, *Proceedings of 21st Asia-Pacific Software Engineering Conference (APSEC'14)*, pp. 167–174. https://doi.org/10.1109/apsec.2014.35

[20] Juliana Alves Pereira and Kattiana Constantino and Eduardo Figueiredo (2015) A Systematic Literature Review of Software Product Line Management Tools, *Proceeddings of 14th International Conference on Software Reuse (ICSR'15)*, Berlin Heidelberg, pp. 73–89. https://doi.org/10.1007/978-3-319-14130-5_6

[21] Durga Prasad Mohapatra and Rajeev Kumar and Rajib Mall and D. S. Kumar and Mayank Bhasin (2006) Distributed dynamic slicing of Java programs, *Journal of Systems and Software*, 79(12), pp. 1661–1678. https://doi.org/10.1016/j.jss.2006.01.009

[22] Durga Prasad Mohapatra and Rajib Mall and Rajeev Kumar (2005) Computing dynamic slices of concurrent object-oriented programs, *Information & Software Technology*, 47(12), pp. 805–817. https://doi.org/10.1016/j.infsof.2005.02.002

[23] Dishant Munjal and Jagannath Singh and Subhrakanta Panda and Durga Prasad Mohapatra (2015) Automated Slicing of Aspect-Oriented Programs using Bytecode Analysis, *Proceedings of IEEE 39th Annual International Computers, Software & Applications Conference (COMPSAC 2015)*, pp. 191–199. https://doi.org/10.1109/compsac.2015.98

[24] Madhusmita Sahu and Durga Prasad Mohapatra (2007) A Node-Marking Technique for Dynamic Slicing of Aspect-Oriented Programs, *Proceedings of 10th International Conference on Information Technology (ICIT 2007)*, pp. 155–160. https://doi.org/10.1109/icit.2007.70

[25] Diganta Goswami and Rajib Mall (1999) Fast Slicing of Concurrent Programs, *Proceedings of 6th International Conference on High Performance Computing (HiPC 1999)*, pp. 38–42. https://doi.org/10.1007/978-3-540-46642-0_6

[26] Diganta Goswami and Rajib Mall (2000) Dynamic Slicing of Concurrent Programs, *Proceedings of 7th International Conference on High Performance Computing (HiPC 2000)*, pp. 15–26. https://doi.org/10.1007/3-540-44467-x_2

[27] Jaiprakash T. Lallchandani and Rajib Mall (2011) A Dynamic Slicing Technique for UML Architectural Models, *IEEE Transactions on Software Engineering*, 37(6), pp. 737–771. https://doi.org/10.1109/tse.2010.112

[28] Philip Samuel and Rajib Mall (2009) Slicing-based test case generation from UML activity diagrams, *ACM SIGSOFT Software Engineering Notes*, 34(6), pp. 1–14. https://doi.org/10.1145/1640162.1666579

[29] Jaiprakash T. Lallchandani and Rajib Mall (2010) Integrated state-based dynamic slicing technique for UML models, *IET Software*, 4(1), pp. 55–78. https://doi.org/10.1049/iet-sen.2009.0080

[30] G. B. Mund and Rajib Mall (2006) An efficient interprocedural dynamic slicing method, *The Journal of Systems and Software*, 79, pp. 791–806. https://doi.org/10.1016/j.jss.2005.07.024

[31] Diganta Goswami and Rajib Mall (2004) A parallel algorithm for static slicing of concurrent programs, *Concurrency – Practice and Experience*, 16(8), pp. 751–769. https://doi.org/10.1002/cpe.789

[32] G. B. Mund and R. Mall and S. Sarkar (2003) Computation of intraprocedural dynamic program slices, *Information and Software Technology*, 45 (8), pp. 499–512. https://doi.org/10.1016/S0950-5849(03)00029-6

[33] G. B. Mund and R. Mall and S. Sarkar (2002) An efficient dynamic program slicing technique, *Information and Software Technology*, 44 (2), pp. 123–132. https://doi.org/10.1016/s0950-5849(01)00224-5

[34] Diganta Goswami and Rajib Mall (2002) An efficient method for computing dynamic program slices, *Information Processing Letters*, 81(2), pp. 111–117. https://doi.org/10.1016/s0020-0190(01)00202-2

[35] Madhusmita Sahu and Durga Prasad Mohapatra (2016) Dynamic Slicing of Feature-Oriented Programs, *Proceedings of 3rd International Conference on Advanced Computing, Networking and Informatics (ICACNI 2015)*, pp. 381–388. https://doi.org/10.1007/978-81-322-2529-4_40

[36] Mark Weiser (1981) Program Slicing, *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pp. 439–449. IEEE Computer Society, March.

[37] Durga Prasad Mohapatra and Madhusmita Sahu and Rajib Mall and Rajeev Kumar (2008) Dynamic Slicing of Aspect–Oriented Programs, *Informatica*, 32(3), pp. 261–274.

[38] Jaiprakash T. Lallchandani and Rajib Mall (2005) Computation of Dynamic Slices for Object-Oriented Concurrent Programs, *Proceedings of Asia Pacific Software Engineering Conference (APSEC 2005)*, pp. 341–350. https://doi.org/10.1109/apsec.2005.51

[39] Jianjum Zhao (1998) Dynamic Slicing of Object–Oriented Programs, Technical report, Information Processing Society of Japan, pp. 1–7, May.

[40] Sebastian Gunther and Sagar Sunkle (2009) Feature-Oriented Programming with Ruby. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD'09)*, pp. 11–18, October. https://doi.org/10.1145/1629716.1629721

[41] Sebastian Gunther and Sagar Sunkle (2012) rbFeatures: Feature-oriented programming with Ruby, *Science of Computer Programming*, 77(3), pp. 152–173, March. https://doi.org/10.1016/j.scico.2010.12.007

[42] Bogdan Korel and Satish Yalamanchili (1994) Forward Computation Of Dynamic Program Slices, *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis(ISSTA'94)*, pp. 66–79, August. https://doi.org/10.1145/186258.186514

[43] Sven Apel and Thomas Leich and Gunter Saake(2008) Aspectual Feature Modules, *IEEE Transactions On Software Engineering*, 34(2),pp. 162–180, March/April. https://doi.org/10.1109/tse.2007.70770

[44] Jia Liu and Don Batory and Srinivas Nedunuri (2005) Modeling Interactions in Feature-Oriented Software Designs, *Proceedings of International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI 2005)*, pp. 178–197.

[45] Ian Adams and Sigmon Myers (2009) FOP and AOP: Benefits, Pitfalls and Potential for Interaction, pp. 1–7.

[46] Sagar Sunkle and Marko Rosenmuller and Norbert Siegmund and Syed Saif ur Rahman and Gunter Saake and Sven Apel (2008) Features as First-class Entities-Toward a Better Representation of Features. *Proceedings of Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pp. 27–34, October.

[47] Susan Horwitz and Thomas Reps and David Binkley (1990) Inter-Procedural Slicing Using Dependence Graphs, *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26–60, January. https://doi.org/10.1145/77606.77608

8 Appendices

A Construction of CFADG

Algorithm 8 Construction of CFADG

Input: The feature-oriented program containing aspects with selected required features and weaved aspects.
Output: The composite feature-aspect dependence graph (CFADG).

procedure CONSTRUCTPDG()
  for start of a method **do**
    Create *method entry node*.
  for each executable statement in the program **do**
    Create a node.
  for all nodes created **do**
    if node n_2 is under the scope of node n_1 **then**
      Add *control dependence edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
    if node n_1 controls the execution of node n_2 **then**
      Add *control dependence edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
    if node n_2 uses the value of a variable that is defined at node n_1 **then**
      Add *data dependence edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
procedure CONSTRUCTMXDG()
  for all methods in a mixin **do**
    Call *ConstructPDG()*.
  for entry of a mixin **do**
    Create *mixin entry node*.
  for each parameter present in the method call **do**
    Create an *actual-in* parameter node.
  for each parameter present in the method definition **do**
    Create a *formal-in* parameter node.
  for each parameter in the method call that is modified inside the method **do**
    Create an *actual-out* parameter node.
  for each actual-out parameter node **do**
    Create corresponding *formal-out* parameter node.
  for all nodes created **do**
    if node x corresponds to mixin entry node and node y is a method entry node **then**
      Add *mixin membership edge* from x to y, $x \rightarrow y$.
    if node n_1 returns a value to the calling method at node n_2 within a mixin layer **then**
      Add *return dependence edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
    if node n_1 calls a method that is defined at node n_2 within a mixin layer **then**
      Add *call edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
    if node n_1 calls a method that is defined at node n_2 within a mixin layer by passing parameters **then**
      Add *call edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
      Add *parameter-in edge* from actual-in parameter node to corresponding formal-in parameter node.
      Add *parameter-out edge* from formal-out parameter node to corresponding actual-out parameter node.
    if node n_1 is an actual-in parameter node and node n_2 is an actual-out parameter node such that the value at node n_1 affects the value at node n_2 **then**
      Add *summary edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
procedure CONSTRUCTSDG()
  for all mixins within a mixin layer **do**
    Call *ConstructMxDG*.
  for all nodes created **do**
    if node x is a *polymorphic method call* **then**
      Create *polymorphic choice vertex*.
      if node y is a polymorphic choice vertex **then**
        Add a *call edge* from x to y, $x \rightarrow y$
    if node x is a *new* operator node and node y is the corresponding constructor node **then**
      Add *call edge* from n_1 to n_2, $n_1 \rightarrow n_2$.
      Add *parameter-in edge* from actual-in parameter node to corresponding formal-in parameter node.
      Add *parameter-out edge* from formal-out parameter node to corresponding actual-out parameter node.

```
    Remove mixin membership edges.
    Remove mixin entry nodes.
procedure CONSTRUCTADG()
    for start of an advice do
        Create advice start vertex.
    if advice contains parameters then
        Create formal-in and formal-out parameter nodes.
    for all nodes created do
        if node n2 is under the scope of node n1 then
            Add control dependence edge from n1 to n2, n1 → n2.
        if node n1 controls the execution of node n2 then
            Add control dependence edge from n1 to n2, n1 → n2.
        if node n2 uses the value of a variable that is defined at node n1 then
            Add data dependence edge from n1 to n2, n1 → n2.
procedure CONSTRUCTIDG()
    for entry of introduction do
        Create introduction start vertex.
    if introduction is a method or constructor then
        Call ConstructPDG.
    if introduction is a field then
        Do nothing.
procedure CONSTRUCTPTDG()
    for entry of pointcut do
        Create pointcut start vertex.
    if pointcut contains parameters then
        Create actual-in and actual-out parameter nodes.
procedure CONSTRUCTASDG()
    for entry of an aspect do
        Create aspect start vertex.
    for all advices in an aspect do
        Call ConstructADG().
    for all pointcuts in an aspect do
        Call ConstructPtDG().
    for all introductions in an aspect do
        Call ConstructIDG().
    for all nodes created do
        if node x is aspect start vertex then
            if node y is advice start vertex then
                Create aspect membership edge from x to y, x → y.
            if node y is pointcut start vertex then
                Create aspect membership edge from x to y, x → y.
            if node y is introduction start vertex then
                Create aspect membership edge from x to y, x → y.
        if node x is pointcut start node and node y is advice start node then
            Create data dependence edge from x to y, x → y.
            Add parameter-in edge from actual-in parameter node to corresponding formal-in parameter node.
            Add parameter-out edge from formal-out parameter node to corresponding actual-out parameter node.
procedure CONSTRUCTCFADG()
    for each mixin layer do
        Call ConstructSDG().
    for each aspect do
        Call ConstructAsDG.
    for all nodes created do
        if node n2 in one mixin layer uses the value of a variable that is defined at node n1 in different mixin layer then
            Add mixin data dependence edge from n1 to n2, n1 → n2.
        if node n2 in an aspect uses the value of a variable that is defined at node n1 in a mixin then
            Add aspect data dependence edge from n1 to n2, n1 → n2.
        if node n1 in one mixin layer returns a value to the calling method at node n2 in different mixin layer then
            Add mixin return dependence edge from n1 to n2, n1 → n2.
        if node n1 in one mixin layer calls a method that is defined at node n2 in different mixin layer then
            Add mixin call edge from n1 to n2, n1 → n2.
            Add parameter-in and parameter-out edges.
        if node n1 calls a method that is defined at node n2 using Super() method then
            Add refinement edge from n1 to n2, n1 → n2.
        if node n1 is an output statement followed by node n2 and node n2 is an input, a computation, a predicate, or a method call statement then
            Add message dependence edge from n1 to n2, n1 → n2.
        if node n1 is a method call node and node n2 is a before advice node capturing the method called at n1 then
            Add weaving edge from n1 to n2, n1 → n2.
        if node n1 is the last statement in a before advice and node n2 is the method entry node of the method captured by the advice then
            Add weaving edge from n1 to n2, n1 → n2.
        if node y is an after advice node and node n1 is the last statement in the method captured by node n2 then
            Add weaving edge from n1 to n2, n1 → n2.
        if node n1 is the last statement in an after advice and node n2 is the statement followed by method call node and the method is captured by the advice then
            Add weaving edge from n1 to n2, n1 → n2.
    Remove aspect membership edges.
    Remove aspect entry vertices.
```

B Feature-aspect node-marking dynamic slicing (FANMDS) algorithm

Algorithm 9 Feature-Aspect Node Marking Dynamic Slicing (FANMDS) Algorithm

INPUT: Composite Feature-Aspect Dependence Graph (CFADG) of the program FP, Slicing criterion $< i, s, v >$.
OUTPUT: List of nodes contained in required dynamic slice.

$Marked = \phi$ ▷ *Initially, unmark all nodes of CFADG.*
Set $dyn\_slice(u) = \phi$ ▷ *u is a node in CFADG.*
Set $RecDefn(v) = NULL$ ▷ *v is a variable.*
Execute the program FP for input i.
while FP does not terminate **do**
  Update $dyn\_slice(u) = \{u, e_1, e_2, \ldots, e_k\} \cup dyn\_slice(e_1) \cup dyn\_slice(e_2) \cup \ldots \cup dyn\_slice(e_k)$
  $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
  if u is a $Defn(v)$ node **then**
    $Marked = Marked \setminus \{RecDefn(v)\}$. ▷ *Unmark the node RecDefn(v).*
    $RecDefn(v) = u$. ▷ *Update RecDefn(v).*
  if u is a *method call* node for a method M **then**
    $panode_M = f(M, panode)$.
    $Me_M = g(M, Me)$.
    $pfnode_M = h(M, pfnode)$.
    $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
    $Marked = Marked \cup panode_M$. ▷ *Mark associated actual parameter nodes.*
    $Marked = Marked \cup \{Me_M\}$. ▷ *Mark corresponding method entry node.*
    $Marked = Marked \cup pfnode_M$. ▷ *Mark associated formal parameter nodes.*
  if u is a `new` *operator* node for a constructor M **then**
    $panode_M = f(M, panode)$.
    $Me_M = g(M, Me)$.
    $pfnode_M = h(M, pfnode)$.
    $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
    $Marked = Marked \cup panode_M$. ▷ *Mark associated actual parameter nodes.*
    $Marked = Marked \cup \{Me_M\}$. ▷ *Mark corresponding method entry node.*
    $Marked = Marked \cup pfnode_M$. ▷ *Mark associated formal parameter nodes.*
  if u is a *polymorphic* node for a virtual method M **then**
    $panode_M = f(M, panode)$.
    $Me_M = g(M, Me)$.
    $pfnode_M = h(M, pfnode)$.
    $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
    $Marked = Marked \cup panode_M$. ▷ *Mark associated actual parameter nodes.*
    $Marked = Marked \cup \{Me_M\}$. ▷ *Mark corresponding method entry node.*
    $Marked = Marked \cup pfnode_M$. ▷ *Mark associated formal parameter nodes.*
  if u is a *mixin call* node for a method M **then**
    $panode_M = f(M, panode)$.
    $Me_M = g(M, Me)$.
    $pfnode_M = h(M, pfnode)$.
    $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
    $Marked = Marked \cup panode_M$. ▷ *Mark associated actual parameter nodes.*
    $Marked = Marked \cup \{Me_M\}$. ▷ *Mark corresponding method entry node.*
    $Marked = Marked \cup pfnode_M$. ▷ *Mark associated formal parameter nodes.*
  if u is a `Super()` *method call* node for a method M **then**
    $Me_M = h(M, Me)$.
    $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
    $Marked = Marked \cup \{Me_M\}$. ▷ *Mark corresponding method entry node.*
  if u is a *pointcut* node **then**
    $badv_P = x(P, badv)$.
    $aadv_P = y(P, aadv)$.
    $panode_P = f(P, panode)$.
    $pfnode_M = g(M, pfnode)$.
    $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
    $Marked = Marked \cup panode_M$. ▷ *Mark corresponding actual parameter nodes.*
    $Marked = Marked \cup pfnode_M$. ▷ *Mark corresponding formal parameter nodes.*
    $Marked = Marked \cup badv_P$. ▷ *Mark the corresponding before advice entry node.*
    $Marked = Marked \cup aadv_P$. ▷ *Mark the corresponding after advice entry node.*
  if u is an *advice* entry node for an advice A corresponding to pointcut P **then**
    $bbadv_A = z(A, badv_P)$.
    $baadv_A = z(A, aadv_P)$.
    $Marked = Marked \setminus bbadv_A$.
    $Marked = Marked \setminus baadv_A$. ▷ *Unmark all nodes in body of advice corresponding to previous execution of u.*
    $pfnode_M = g(M, pfnode)$.
    $Marked = Marked \setminus pfnode_M$. ▷ *Unmark all the formal parameter nodes associated with u corresponding to previous execution of u.*

64: **if** u is an *introduction* node such that u is a method **then**
65: $Mb_M = k(M, Mb)$.
66: $Marked = Marked \setminus Mb_M$. ▷ *Unmark all the nodes in the method body corresponding to previous execution of u.*
67: $pfnode_M = g(M, pfnode)$.
68: $Marked = Marked \setminus pfnode_M$. ▷ *Unmark all the formal parameter nodes associated with u corresponding to previous execution of u.*
69: **if** v is method call node corresponding to previous execution of u **then**
70: $Marked = Marked \setminus v$. ▷ *Unmark the method call node corresponding to previous execution of u.*
71: $panode_v = f(v, panode)$.
72: $Marked = Marked \setminus panode_v$. ▷ *Unmark the associated actual parameter nodes for a method call node corresponding to previous execution of u.*
73: $pfnode_M = h(M, pfnode)$.
74: $Marked = Marked \cup \{u\}$. ▷ *Mark node u.*
75: $Marked = Marked \cup pfnode_M$. ▷ *Mark associated formal parameter nodes.*
76: **if** u is an *introduction* node such that u is a field **then**
77: $Marked = Marked \cup \{Defn(v)\}$. ▷ *Mark Defn(v) node.*
78: $Marked = Marked \cup \{Usage(v)\}$. ▷ *Mark Usage(v) node.*
79: **if** u is a *method entry* node for a method M **then**
80: $Mb_M = k(M, Mb)$.
81: $Marked = Marked \setminus Mb_M$. ▷ *Unmark all the nodes in the method body corresponding to previous execution of u.*
82: $pfnode_M = g(M, pfnode)$.
83: $Marked = Marked \setminus pfnode_M$. ▷ *Unmark all the formal parameter nodes associated with u corresponding to previous execution of u.*
84: **if** v is method call node corresponding to previous execution of u **then**
85: $Marked = Marked \setminus v$. ▷ *Unmark the method call node corresponding to previous execution of u.*
86: $panode_v = f(v, panode)$.
87: $Marked = Marked \setminus panode_v$. ▷ *Unmark the associated actual parameter nodes for a method call node corresponding to previous execution of u.*
88: **if** u is a *mixin entry* node for a method M **then**
89: $Mb_M = k(M, Mb)$.
90: $Marked = Marked \setminus Mb_M$. ▷ *Unmark all the nodes in the method body corresponding to previous execution of u.*
91: $pfnode_M = g(M, pfnode)$.
92: $Marked = Marked \setminus pfnode_M$. ▷ *Unmark all the formal parameter nodes associated with u corresponding to previous execution of u.*
93: **if** v is method call node corresponding to previous execution of u **then**
94: $Marked = Marked \setminus v$. ▷ *Unmark the method call node corresponding to previous execution of u.*
95: $panode_v = f(v, panode)$.
96: $Marked = Marked \setminus panode_v$. ▷ *Unmark the associated actual parameter nodes for a method call node corresponding to previous execution of u.*
97: **if** u is `new` *operator entry* node for a constructor M **then**
98: $Mb_M = k(M, Mb)$.
99: $Marked = Marked \setminus Mb_M$. ▷ *Unmark all the nodes in the method body corresponding to previous execution of u.*
100: $pfnode_M = g(M, pfnode)$.
101: $Marked = Marked \setminus pfnode_M$. ▷ *Unmark all the formal parameter nodes associated with u corresponding to previous execution of u.*
102: **if** v is method call node corresponding to previous execution of u **then**
103: $Marked = Marked \setminus v$. ▷ *Unmark the method call node corresponding to previous execution of u.*
104: $panode_v = f(v, panode)$.
105: $Marked = Marked \setminus panode_v$. ▷ *Unmark the associated actual parameter nodes for a method call node corresponding to previous execution of u.*
106: **for** a given slicing command $< i, s, v >$ **do**
107: Look up $dyn\_slice(u)$ for variable v.
108: Display $dyn\_slice(u)$.
109: Map nodes in $dyn\_slice(u)$ to corresponding statements in composed Java program.
110: Map statements included in $dyn\_slice(u)$ in composed Java program to corresponding statements in composed Jak program.
111: Display statements included in $dyn\_slice(u)$ from composed Jak program.