**Vrije Universiteit Brussel - Belgium**
**Faculty of Sciences**
**In Collaboration with Ecole des Mines de Nantes - France**
**2007**

# CONTROL-FLOW INTERACTION IN ASPECT-ORIENTED PROGRAMMING

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Pablo Daniel Quiroga

Promoter: Prof. Viviane Jonckers (Vrije Universiteit Brussel)
Advisors: Bruno De Fraine and Wim Vanderperren (Vrije Universiteit Brussel)

**Abstract**

In order to manage the increasing complexity of current software systems, it is necessary to maintain a clear separation of the involved concerns. At the level of the implementation, aspect-oriented programming (AOP) offers modularization constructs (*aspects*) to separate concerns that crosscut traditional modularization boundaries. Unfortunately, when multiple aspects are combined, they can interact in an unexpected manner. Even though some support exist for managing these interactions among aspects, most techniques are only applicable when the aspects share a common join point. Since other interaction can be very relevant as well, this dissertation proposes a technique for managing broader, control flow interactions among aspects. In this technique we employ existing static analyses to produce control flow graphs and call graphs of the woven result. In addition, we propose a set of predicates, as in predicate logic, that represent relevant situations in the control flow, as can be observed in the aforementioned graphs. The aspects can then be documented with control flow *policies* expressed as logical formulae that employ these predicates. The policies specify certain control flow relations that must hold between different aspects, or between the aspects and the base system. Finally, we provide a light-weight object-oriented logic engine that allows the evaluation of the logic formulae. As such, the policies can be verified automatically. This provides a complete tool to the developer for the management of the control flow interactions in a software system.

# Acknowledgements

I would like to thank all the people at the Ecole des Mines de Nantes and at the Vrije Universiteit Brussel for their support during all this master. Specially, to my promotors Dr. Viviane Jonckers, Dr. Theo D'Hont and Dr. Jacques Noyé for giving me the opportunity to do this thesis.

I am particularly grateful for the invaluable help and dedication of Bruno, Wim and Coen.

A special thanks to my parents, Héctor and Graciela, and my sister Analia.

Last but not least, I would like to thank Maria for her love and her patience during all this master.

# Contents

**1 Introduction** **1**
1.1 Problem Statement . . . . . 2
1.1.1 Basic architecture example . . . . . 2
1.1.2 Base case with an aspect . . . . . 3
1.1.3 Case with two aspects depending each other . . . . . 4
1.1.4 Case with two aspects excluding each other . . . . . 4
1.2 Objectives . . . . . 5
1.3 Outline . . . . . 6

**2 Context** **7**
2.1 Aspect-oriented programming . . . . . 7
2.1.1 AspectJ . . . . . 9
2.1.2 Other approaches . . . . . 19
2.2 Feature interaction . . . . . 21
2.2.1 Detection . . . . . 22
2.2.2 Resolution . . . . . 24
2.3 Summary . . . . . 26

**3 Analysis with Soot tool** **27**
3.1 Overview of Soot . . . . . 27
3.1.1 Functionality . . . . . 27
3.1.2 How to run Soot . . . . . 28
3.2 Elements of the Analysis . . . . . 30
3.2.1 Control flow graphs . . . . . 30
3.2.2 Call graphs . . . . . 31
3.2.3 Points-to analysis . . . . . 33
3.3 Combination of CFGs and CGs . . . . . 35
3.3.1 Technique 1: Inlining . . . . . 35
3.3.2 Technique 2: Connecting . . . . . 35
3.3.3 Technique 3: Connecting + labelling . . . . . 36
3.3.4 Practical Realization . . . . . 36
3.4 Summary . . . . . 37

**4 Formal Documentation** **39**
4.1 Policy language . . . . . 39
4.1.1 Predefined predicates . . . . . 39
4.1.2 Derived predicates . . . . . 40
4.2 Problems Revisited . . . . . 41
4.3 Implementation of a light-weight logic engine . . . . . 41
4.3.1 Proposed engine . . . . . 42

4.3.2 General concepts . . . 42
4.3.3 Goals . . . 42
4.3.4 Predicates . . . 43
4.4 Summary . . . 44

**5 Conclusions** **45**
5.1 Evaluation of the Technique . . . 45
5.1.1 Static analysis . . . 45
5.1.2 Formal documentation . . . 45
5.1.3 Logic engine . . . 46
5.2 Contributions . . . 46
5.3 Future work . . . 46

**Bibliografhy** **47**

# List of Figures

1.1 Interaction example . . . 2
1.2 Three-tier architecture . . . 3
1.3 Applying authorization aspect . . . 4
1.4 Applying dependency of aspects . . . 5
1.5 Applying exclusion of aspects . . . 6

2.1 Source-level weaving . . . 8
2.2 Telephone calls class diagram . . . 9
2.3 Schematic overview of JAsCo . . . 20
2.4 Architecture of an AOP kernel . . . 21
2.5 Overview approach . . . 23
2.6 Ordering and nesting scenarios . . . 25

3.1 Soot overview . . . 28
3.2 Soot run example . . . 29
3.3 Soot options . . . 29
3.4 Control flow graph created by Soot . . . 31
3.5 Call graph created by Soot (abridged) . . . 32
3.6 Unrealizable path . . . 35
3.7 Valid paths . . . 36
3.8 Combination between CFGs and CGs . . . 38

4.1 Example argument. (path={methodA,methodB}) . . . 40
4.2 An overview of a logic engine . . . 42

# List of Tables

2.1 User's calls with precedence . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
2.2 User's calls without precedence . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17

4.1 Logical connectives . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

# Listings


2.1 Customer class . . . . . 10
2.2 Call class . . . . . 11
2.3 Connection class . . . . . 12
2.4 Local and LongDistance connections classes . . . . . 12
2.5 Timer class . . . . . 13
2.6 TimerLog aspect . . . . . 13
2.7 Timing aspect . . . . . 13
2.8 Billing aspect . . . . . 14
2.9 AbstractSimulation class . . . . . 15
2.10 BillingSimulation class . . . . . 16
2.11 Tagable interface . . . . . 17
2.12 TagAspect aspect . . . . . 17
2.13 Compile time warnings and errors . . . . . 18
2.14 Advice declaration . . . . . 18
2.15 Exception softening . . . . . 18
2.16 Pointcut example . . . . . 24
3.1 Control flow graph creation . . . . . 30
3.2 Application system . . . . . 31
3.3 Call graph construction . . . . . 31
3.4 Call graph use . . . . . 32
3.5 Combination between CFGs and CGs . . . . . 37
4.1 AspectJ's pointcut language (method patterns) . . . . . 40

# 1 Introduction

Since object-oriented programming (OOP) appeared in the software development, we have a different view on how software has to be developed. Nowadays, the developer can work with his systems as groups of entities and with the interaction between those entities as well. The more important thing here is that the developer can deal with larger and more complicated systems. In addition, he can develop them in less time than ever before [O'R04]. Nevertheless, when we use OOP on large projects, we can have some problems with maintaining their code because it remains difficult to separate some concerns into modules. An attempt to do a minor change in the program design may require several updates to a large number of unrelated modules.

In order to solve the problems that some concerns are not cleanly modularized using traditional abstraction mechanisms such as class hierarchies, aspect-oriented programming (AOP) provides a better separation of concerns. Typical examples of such concerns are tracing, synchronization and transaction management. An aspect-oriented language that uses the popular pointcut/advice mechanism, defines a join point model (JPM). A JPM defines join points that are points in a running program where additional behavior can be usefully joined. This additional behavior is known as advice and basically an advice can run before, after and around (i.e. instead of) join points. Additionally, the AOP language defines a so-called pointcut language to specify (or quantify) join points. Pointcuts determine whether a given join point matches.

As we know, AspectJ [KHH+01] is an aspect-oriented extension to Java. In AspectJ, an aspect selects a set of join points in the target application where advices alter the regular execution. The aspect logic is then automatically woven into the target application. Due to that, when multiple, perhaps independently developed, aspects are specified, it could produce unexpected behavior in the system. This problem has been identified as the *feature interaction problem*. The most commonly considered interaction among aspects is when they specify behavior at the same join point. For that kind of interaction we have detection and resolution facilities. On one way, it is possible to detect when two aspects advise the same join point. This join point could be a method call or a method execution. On the other hand, for resolution facilities we specify that when aspect A is triggered, aspect B cannot be triggered. This situation could be tackled with the JAsCo [SVJ03] approach applying a *combination strategy* as a solution.

However, interactions can occur without the aspects sharing a joint point, so we have to consider more than just the shared join points. In this thesis, we propose to verify relations like the example with aspect A and B from above, but for aspects that are applied on a certain (static) control flow. For instance, we suppose there are two advices called A1 and A2 (see figure 1.1). The advice A1 has a pointcut that matches with a method call at the method m2. In addition, the advice A2 has another pointcut that matches with the same method m2 but in the method execution. So, now if we call method m2 in a certain method m1, we note that the m2 method execution is in the control flow of the m2 method call. This is because a method

call wraps a method execution. However, the state-of-the-art aspect-oriented approaches offer limited support for explicitly managing the interactions among aspects, when they do not share a join point. This is unfortunate in this case, because the advices do not share a join point based on the (arguably arbitrary) distinction between call and execution join points of a method invocation. In other words, the likelihood of interaction between the advices is probably not altered because of the fact that one aspect advises the call join point while the other advises the execution join point. Yet in mainstream approaches, we lose all support for managing any interactions because of this distinction. As we will show in the problem statement, support for the detection of control-flow interactions is useful beyond the case of relating call and execution join points.

For this reason, we propose a detection technique to verify control-flow interactions between aspects in general. This technique consists of three main parts. The first part uses a tool to statically analyze the woven result of the system. In this way, we can derive all the possible control flow paths with the involved aspects. In the second part, a formal documentation of aspects is defined. This documentation allows us to define, as *policies*, certain control-flow relations among aspects and between an aspect and the base program. We use well-known operators from propositional logic and first-order logic for the definition of these policies. Finally, we define an algorithm verify policies using the control-flow paths constructed by the first part of the approach. As such, we are able to detect violations of the policies in the actual control flow.

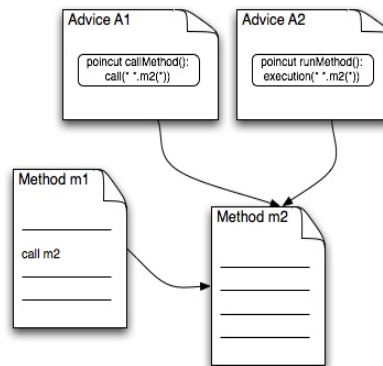


Figure 1.1: Interaction example

## 1.1 Problem Statement

To motivate our approach for managing control-flow interactions among aspects, we will use examples in the context of the well-know three-tier architecture, as employed in a large number of contemporary middleware solutions (e.g. JBoss, Spring, etc.).

### 1.1.1 Basic architecture example

The three-tier architecture is used when an effective client/server design is needed that provides increased performance, flexibility, maintainability, reusability, and scalability. In the top-most level of application there is the user interface. This level is called the *Presentation tier*. The main function of the interface is to translate tasks and results to something the user can understand. Afterwards, there is a *Logic tier*. This layer coordinates the application, processes commands,

makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two other layers. Lastly, the *Data tier* contains the information that it is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then comes back to the user.

The figure 1.2 shows an example of a three-tier architecture.

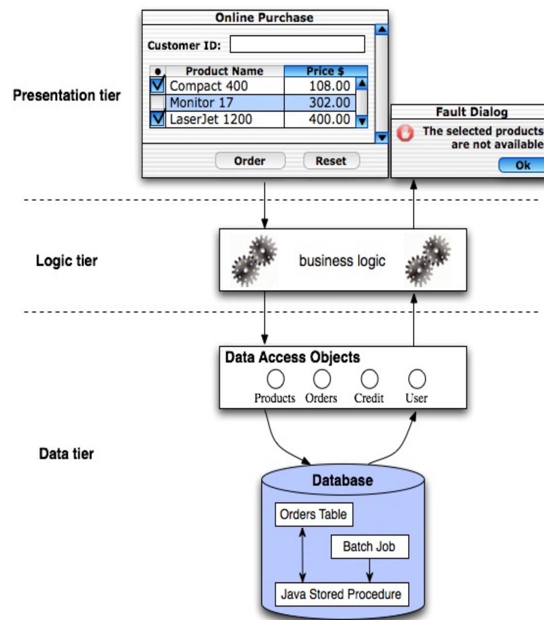


Figure 1.2: Three-tier architecture

In order to demonstrate the control-flow in the three-tier architecture, a simple purchasing module will be presented. In this demonstration, a product catalog is displayed to the user (Graphical User Interface - GUI). The user can browse through the catalog and purchase products of interest. When the user chooses some products, the logic tier receives the user's choice and it takes decisions such as add all product prices together. Afterwards, the logic tier passes the user's selection to the data access objects. So, these objects can execute queries at the database. After the queries are executed, the control-flow goes back to the objects. Once the objects get the result from the database, the control-flow goes to the logic tier again. Therefore, the logic tier can take new decisions about, for instance, which information is shown. Finally, the control-flow of the example ends when the user interface shows the answer of the purchase with a message dialog.

### 1.1.2 Base case with an aspect

Once the functionality of the three-tier architecture was explained we continue with the main motivation of our approach. In the first place, we present a simple case in which the control-flow is affected by an aspect. To do this, we use the same example of purchasing previously explained. In order to begin the example, we suppose that we must administer the resources that the system offers, and as a consequence allow or not the access them. Therefore, an authorization aspect is required. In the figure 1.3 we can appreciate that the authorization aspect is applied in the data tier. This is because the resources in this layer are represented by objects such as the products of the purchase. Finally, if an authorization aspect is applied to the purchase

system, we will be able to guarantee a more sure access to the resources. In this way, no user could accede to a resource that is not allowed to him.

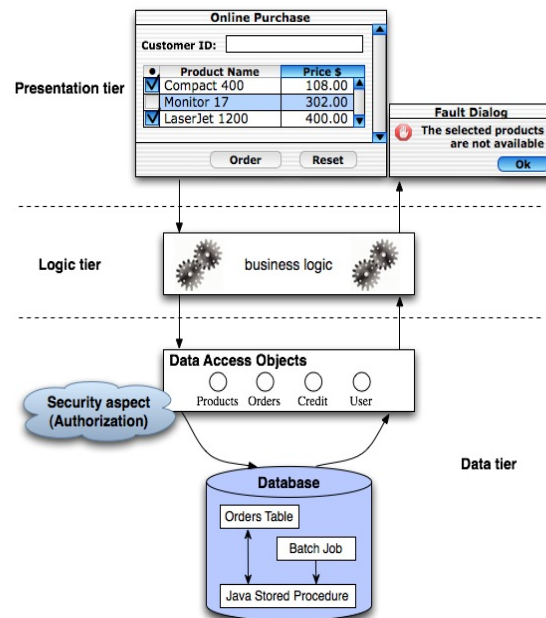


Figure 1.3: Applying authorization aspect

### 1.1.3 Case with two aspects depending each other

Another situation that often appears in programming is when several aspects are needed to apply in a system. This situation can cause problems when these aspects are implemented by different developers. Therefore, it can affect the behavior of the system, since the aspects could be related to each other. If this happens, the simultaneous execution of these aspects would have to be controlled so that the system continues its normal execution.

In order to illustrate the situation mentioned above, we consider an authorization aspect that determines whether the user is authorized or not to access to the resources. Another aspect that we use is an authentication aspect. Most of the time these aspects are related because we need to know if the user is a legitimate user to gain access to one of the resources. So, the authentication aspect should be applied at the logic tier, because the verification of the user has to be done before the user obtains permission to access the resource. With the authentication aspect we can prevent malicious access to a resource. The figure 1.4 shows the combination of both authorization aspect and authentication aspect.

We can graphically notice that we must invoke the authentication aspect before the authorization aspect is invoked. In addition, we can see that the aspects are applied at the same control flow at the purchase example, and they have a dependency on each other as well.

### 1.1.4 Case with two aspects excluding each other

Last but not least, we present another situation where two aspects exclude each other. In this case, we want to do caching at the highest possible level (Presentation tier) because we want to gain the most performance. For instance, the fastness of the purchase by *caching* the user data can be improved. This allows us to accelerate the access to the resources such as an acquired

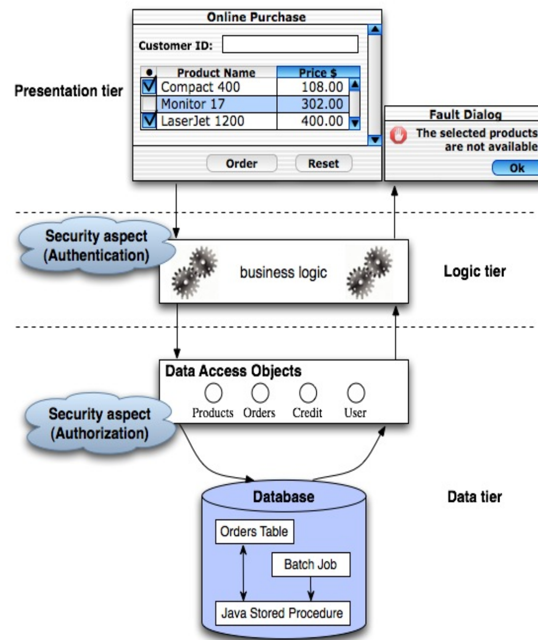


Figure 1.4: Applying dependency of aspects

product in a previous purchase or the visualization of the information of its account. The figure 1.5 shows the caching aspect interacting with both authentication aspect and authorization aspect.

The caching aspect not only improves performance but also helps scalability, by reducing the average request time of the purchase. We can cache frequently used data applicable to all users. For instance, if we make caching of a product's specification, several user could access more faster to the same product. This caching can cause that the access to the product is not verified by the authorization aspect, because the product that the user asked was in the cache. Therefore, the caching aspect would be excluding the execution from the authorization aspect.

Finally, all things considered can help us to observe some possible interactions, that can occur when we apply several aspects on a certain control flow. Those interactions are the motivation of the present work and we focus to detect them.

## 1.2 Objectives

Our main objective in this thesis is to provide support for the management of control-flow interactions among the aspects deployed in a system. We propose a technique that divides the objective in three main points that are detailed in the following way:

$a$. Static analysis of the woven result to reproduce possible control flow. This analysis is useful to know both all the possible control flows and what aspects are applied inside those control flows.

$b$. Formal documentation of aspects to specify what aspects, in the control flow, should be applied. We define each specification like a policy. For instance, one policy could specify that an authentication aspect should be executed before authorization aspect does. In addition, this should happens in the control flow of a certain method.

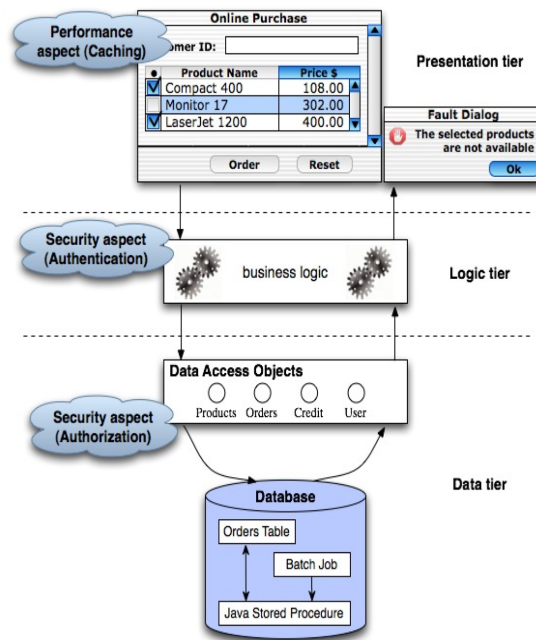


Figure 1.5: Applying exclusion of aspects

*c*. An algorithm to detect violations of these policies on a certain control flow.

## 1.3 Outline

In this section, we present how the next chapters are structured. Firstly, we introduce the main concept of AOP and the state-of-the-art of feature interaction in chapter 2. In chapter 3 we analyze, with Soot tool, how to tackle the first objective defined in section 1.2. Then, in chapter 4 we specify the formal documentation that represents the policies to be applied. In addition, we show the algorithm to detect violations to those policies and examples of both. Finally, in chapter 5 we write down the conclusions of this thesis, and after that we discuss some future work that remain.

# 2 Context

In the present chapter, we introduce the main concept of aspect-oriented programming. In addition, we deal with some approaches of aspect languages. After all, we include a review of the current state-of-the-art of feature interaction.

## 2.1 Aspect-oriented programming

Before introducing the main concept of aspect-oriented programming, we comment why aspects are relevant to the software systems. The development of a software system basically comprises the design and implementation of the basic functionality and other system concerns such as synchronization, distribution, error handling, caching, and security management. While functional and object oriented decomposition are well suited for the design and implementation of the basic functionality, these techniques are not enough to tackle the other aspects. When using these techniques, the concerns are spread over the system and tangled with the code for the basic functionality. So, separation of concern is not well supported for such concerns. In addition, the quality of a software system decreases.

Aspect oriented programming (AOP) [EFB01] aims at supporting the separation of concern for the above mentioned aspects. AOP is a continuous development of the object oriented paradigm, and as such it supports aspect oriented decomposition in addition to object oriented decomposition and functional decomposition. Separation of concerns allows design and code to be structured to reflect the way developers want to think about the system. In addition, it builds on existing technologies and provides additional mechanisms that make it possible to affect the implementation of systems.

Before entering greater detail on AOP, we introduce some terminology to help us understand the concepts [O'R04].

1. Crosscutting concerns: Aspects of a program which affect (crosscut) other concerns and often cannot be cleanly decomposed from the rest of the system in both the design and implementation.

2. Scattering: Term used when some concerns of an application cannot be cleanly modularized because they are scattered all over the different modules of the system.

3. Tangling: Is the situation where multiple concerns are addressed by the same source code construct, for instance, when one method in a class takes care of managing database connections, logging the action to a file on disk, showing the results in a window on the screen, etc.. In short, tangling makes the source code difficult to develop, understand and evolve.

4. Aspect: A modular implementation of a crosscutting concern using some aspect mechanism (e.g. pointcut/advice, introductions, etc.).

AOP does not replace existing programming paradigms and languages; instead, it works with them to improve their expressiveness and functionality. For instance, weaving is required to execute AO program on ordinary execution platform. Therefore, both the code and aspects are combined into a final executable form using an aspect weaver. As a result, a single aspect can contribute to the implementation of a number of methods, modules, or objects. As we can see, the original code does not need to know about any functionality the aspect has added.

There are different ways to do weaving. First, source-level weaving can be implemented using preprocessors that require access to program source files like in figure 2.1. However, Java's well-defined binary form enables bytecode weavers to work with any Java program in *class-file* form. So, bytecode weavers can be deployed during the build process or, if the weave model is per-class, during class loading.

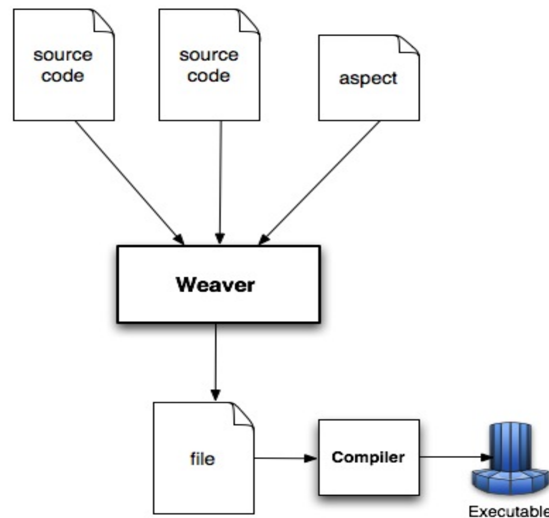


Figure 2.1: Source-level weaving

In AOP [MW99], we can distinguish two main approaches to aspect language design: domain specific aspect languages and general purpose aspect languages. By domain specific we refer to the different concerns. Domain specific aspect languages (DSALs) support one or more of these concerns, but cannot support other ones than the ones they were designed for. DSALs are usually more abstract than the base language and they express the domain specific concepts in a higher level representation. As such, they provide the developer with the usual benefits of a high-level programming language: e.g. higher productivity and better understandibility and maintainability. *Adaptive programming* concerns one example of domain-specific aspect languages, namely those that handle the description of object traversals.

General purpose aspect languages (GPALs) [MW99] are designed to be used for every kind of concern. Therefore, they cannot impose restrictions on the base language. They mainly support the separate definition of concerns by providing aspects. They usually have the same level of abstraction as the base language and also the same set of instructions, as it must be possible to express arbitrary code in the aspects. In addition, GPALs are subdivided in three kind of approach: *pointcuts and advices* approaches (e.g.: AspectJ, JAsCo, CaesarJ and Reflex), *composition filters* (e.g.: Compose*) approaches and *subject-oriented* approaches (e.g.: HyperJ).

In order to see a good example of how AOP works, we introduce the most well-known language called AspectJ. As we said before, this approach is a general purpose aspect language.

After this, we present some other approaches to see the main concepts of them.

### 2.1.1 AspectJ

AspectJ [KHH+01] is an aspect-oriented extension to Java. It introduces AOP programming to Java by adding constructs to support dynamic and static crosscutting. Dynamic crosscutting modifies the behavior of the modules, while static crosscutting modifies the structure of the modules. Dynamic crosscutting in AspectJ is based on a set of constructs. In the first instance, join points are well-defined points in the execution of the program; pointcuts are a means of referring to collections of join points and certain values at those join points. Next, advice are method-like constructs used to define additional behavior at join points; and aspects are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations. AspectJ exposes the join points in a system through pointcuts. Pointcut [FECA05] expressions are built up using AspectJ's primitive pointcut designators. There are three kind of designators. The first set of pointcut designators matches based on the kind of join point (e.g.: method call or access field). The second set of designators matches join points based on context at the join point. For example, when we are executing an instance of a certain class. Finally, the third set of designators matches based on scope, which means that the join point resulting from code within a certain package. The advice constructs provide a way to express actions at the desired join points. On the other hand, static crosscutting, which can be used alone or in support of dynamic crosscutting, includes the constructs of member introduction, type hierarchy modification, compile-time declarations, and exception softening [Lad03].

In order to show a concrete example of dynamic crosscutting in AspectJ, we introduce the telecom example [Tel03]. There, people can make telephone calls with different connection types (local and long-distance). The simulation can be executed at different ways. So, we can use only the calls with the basic functionality needed for making phone calls (call, accept, hang up etc.). Another simulation could be applying a timing aspect, which keeps track of a connection's duration and cumulates a customer's connection durations. But, a useful simulation could be with a billing aspect, that it adds functionality to calculate charges for phone calls of each customer based on connection type and duration. In this way, we are using both aspects at once.

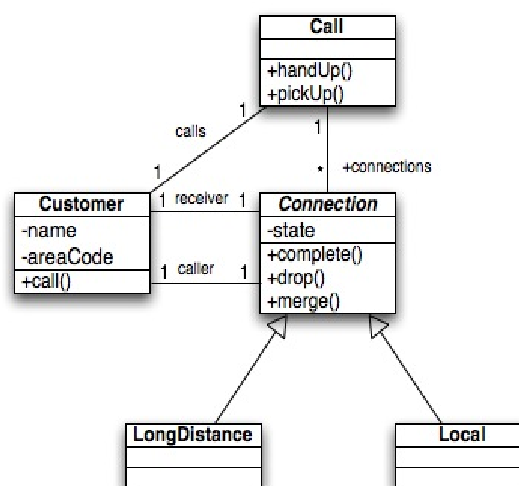


Figure 2.2: Telephone calls class diagram

The figure 2.2 shows how the classes are related without aspects. The basic objects of the

telecom simulation comprise the classes *Customer*, *Call* and the *Connection* abstract class with its two concrete subclasses *Local* and *LongDistance*. Customers have a name and a numeric area code. They also have methods for managing calls. Simple calls are made between one customer (the caller) and another (the receiver). A Connection object is used to connect them. Conference calls between more than two customers will involve more than one connection. A customer may be involved in many calls at one time. Therefore, a Customer has methods *call*, *pickup*, *hangup* and *merge* for managing calls. The code in listing 2.1 shows this class.

Listing 2.1: Customer class

```
public class Customer {
      private String name;
      private int areacode;
      private Vector calls = new Vector();

      protected void removeCall(Call c){
          calls.removeElement(c);
      }
      protected void addCall(Call c){
          calls.addElement(c);
      }
      public Customer(String name, int areacode) {
          this.name = name;
          this.areacode = areacode;
      }
      public String toString() {
          return name + "(" + areacode + ")";
      }
      public int getAreacode(){
          return areacode;
      }
      public boolean localTo(Customer other){
          return areacode == other.areacode;
      }
      public Call call(Customer receiver) {
          Call call = new Call(this, receiver);
          addCall(call);
          return call;
      }
      public void pickup(Call call) {
          call.pickup();
          addCall(call);
      }
      public void hangup(Call call) {
          call.hangup(this);
          removeCall(call);
      }
      public void merge(Call call1, Call call2){
          call1.merge(call2);
          removeCall(call2);
      }
  }
```

Calls are created with a caller and receiver who are customers. If the caller and receiver have the same area code then the call can be established with a Local connection, otherwise a LongDistance connection is required. A call comprises a number of connections between customers. Initially there is only the connection between the caller and receiver but additional

connections can be added if calls are merged to form conference calls. In brief, the Call class is shown in the listing 2.2.

Listing 2.2: Call class

```
public class Call {
    private Customer caller, receiver;
    private Vector connections = new Vector();

    public Call(Customer caller, Customer receiver) {
        this.caller = caller;
        this.receiver = receiver;
        Connection c;
        if (receiver.localTo(caller)) {
           c = new Local(caller, receiver);
        } else {
           c = new LongDistance(caller, receiver);
        }
        connections.addElement(c);
    }
    public void pickup() {
        Connection connection = (Connection)connections.lastElement();
        connection.complete();
    }
    public boolean isConnected(){
        return ((Connection)connections.lastElement()).getState()
            == Connection.COMPLETE;
    }
    public void hangup(Customer c) {
        for(Enumeration e = connections.elements();
        e.hasMoreElements();){
          ((Connection)e.nextElement()).drop();
        }
    }
    public boolean includes(Customer c){
        boolean result = false;
        for(Enumeration e = connections.elements();
        e.hasMoreElements();){
           result = result ||
           ((Connection)e.nextElement()).connects(c);
        }
        return result;
    }
    public void merge(Call other){
        for(Enumeration e = other.connections.elements();
        e.hasMoreElements();){
        Connection conn = (Connection)e.nextElement();
        other.connections.removeElement(conn);
        connections.addElement(conn);
        }
    }
}
```

The Connection class models the physical details of establishing a connection among customers. It does this with a simple state machine (connections are initially PENDING, then COMPLETED and finally DROPPED). Messages are printed to the console so that the state of connections can be observed. Connection is an abstract class with two concrete subclasses:

Local and LongDistance as mentioned above. The Connection class is shown in the listing 2.3.

Listing 2.3: Connection class

```
public abstract class Connection {

      public static final int PENDING = 0;
      public static final int COMPLETE = 1;
      public static final int DROPPED = 2;

      Customer caller, receiver;
      private int state = PENDING;

      Connection(Customer a, Customer b) {
          this.caller = a;
          this.receiver = b;
      }
      public int getState(){
          return state;
      }
      public Customer getCaller() { return caller; }

      public Customer getReceiver() { return receiver; }

      void complete() {
          state = COMPLETE;
          System.out.println("connection completed");
      }
      void drop() {
          state = DROPPED;
          System.out.println("connection dropped");
      }
      public boolean connects(Customer c){
          return (caller == c || receiver == c);
      }
  }
```

The two kinds of connections supported by simulation are Local and LongDistance connections. These connections are represented by the code in the listing 2.4.

Listing 2.4: Local and LongDistance connections classes

```
public class Local extends Connection {
      Local(Customer a, Customer b) {
          super(a, b);
          System.out.println("[new local connection from " +
             a + " to " + b + "]");
      }
  }

public class LongDistance extends Connection {
      LongDistance(Customer a, Customer b) {
          super(a, b);
          System.out.println("[new long distance connection from " +
              a + " to " + b + "]");
      }
  }
```

After all this basic classes, we continue with the aspects in detail. For instance, the Timing aspect keeps track of total connection time for each Customer by starting and stopping a timer associated with each connection. It uses some helper classes like the Timer class shown in the listing 2.5.

Listing 2.5: Timer class

```
1 public class Timer {
2       long startTime, stopTime;
3 
4       public void start() {
5           startTime = System.currentTimeMillis();
6           stopTime = startTime;
7       }
8       public void stop() {
9           stopTime = System.currentTimeMillis();
10      }
11      public long getTime() {
12          return stopTime - startTime;
13      }
14   }
```

A Timer object simply records the current time when it is started and stopped, and returns their difference when asked for the elapsed time. So, after the helper class, the TimerLog aspect, as shown in the listing 2.6, can be used to cause the start and stop times to be printed to standard output.

Listing 2.6: TimerLog aspect

```
1 public aspect TimerLog {
2     after(Timer t): target(t) && call(* Timer.start())  {
3       System.err.println("Timer started: " + t.startTime);
4     }
5     after(Timer t): target(t) && call(* Timer.stop()) {
6       System.err.println("Timer stopped: " + t.stopTime);
7     }
8 }
```

The Timing aspect, in the listing 2.7, declares an inter-type field totalConnectTime for Customer to store the accumulated connection time per Customer. It also declares that each Connection object has a timer. Two pieces of after advice ensure that the timer is started when a connection is completed and and stopped when it is dropped. The pointcut *endTiming* is defined so that it can be used by the Billing aspect.

Listing 2.7: Timing aspect

```
1 public aspect Timing {
2     public long Customer.totalConnectTime = 0;
3 
4     public long getTotalConnectTime(Customer cust) {
5         return cust.totalConnectTime;
6     }
7     private Timer Connection.timer = new Timer();
8     public Timer getTimer(Connection conn) { return conn.timer; }
9 
10    after (Connection c): target(c) &&
11                          call(void Connection.complete()) {
12        getTimer(c).start();
```

```
13     }
14
15     pointcut endTiming(Connection c): target(c) &&
16                                       call(void Connection.drop());
17
18     after(Connection c): endTiming(c) {
19         getTimer(c).stop();
20         c.getCaller().totalConnectTime += getTimer(c).getTime();
21         c.getReceiver().totalConnectTime += getTimer(c).getTime();
22     }
23 }
```

As we see, in the line 11 of the figure 2.7, a pointcut can provide contextual information about join points to their consumers. A pointcut can declare a formal parameter list of context it provides, and the contextual values are extracted based on name binding. For instance, in this line we explicitly want to use a Connection class, but a simple matching join point could be: `after (): call(void *.complete())`. In this example, we only show a simple use of matches certain methods. However, we can specify different pointcuts to matches with other methods. For example, if we need to control the time between a call and its pickup; so, we must add new pointcuts like: `pointcut startCall(): call(void *.call())` and `pointcut acceptedCall(): call(void *.pickup())`, and of course, the corresponding advice that calculates the time of this situation.

Therefore, the Billing aspect (listing 2.8), declares that each Connection has a *payer* inter-type field to indicate who initiated the call, and therefore who is responsible to pay for it. It also declares the inter-type method *callRate* of Connection so that local and long distance calls can be charged differently. The call charge must be calculated after the timer is stopped; the after advice on pointcut *Timing.endTiming* does this, and Billing is declared to be more precedent than Timing to make sure that this advice runs after Timing's advice on the same join point. Finally, it declares inter-type methods and fields for Customer to handle the *totalCharge*.

Listing 2.8: Billing aspect

```
1  public aspect Billing {
2      declare precedence: Billing, Timing;
3
4      public static final long LOCAL_RATE = 3;
5      public static final long LONG_DISTANCE_RATE = 10;
6
7      public Customer Connection.payer;
8      public Customer getPayer(Connection conn) { return conn.payer;}
9
10     after(Customer cust) returning (Connection conn):
11         args(cust, ..) && call(Connection+.new(..)) {
12         conn.payer = cust;
13     }
14
15     public abstract long Connection.callRate();
16
17     public long LongDistance.callRate() { return LONG_DISTANCE_RATE;}
18     public long Local.callRate() { return LOCAL_RATE; }
19
20     after(Connection conn): Timing.endTiming(conn) {
21         long time = Timing.aspectOf().getTimer(conn).getTime();
22         long rate = conn.callRate();
23         long cost = rate * time;
24         getPayer(conn).addCharge(cost);
```

```
    }

    public long Customer.totalCharge = 0;
    public long getTotalCharge(Customer cust) {
        return cust.totalCharge;}

    public void Customer.addCharge(long charge){
        totalCharge += charge;
    }
}
```

All in all, with both the Timing aspect and Billing aspect, we can run the simulation. As we said, we present a simulation to find out how much time each customer spent on the telephone and how big their bill is. That information is also stored in the classes, but they are accessed through static methods of the aspects, since the state they refer to is private to the aspect. Before all else, we show an abstract class that it has the *run method* to print out the status of the customer. The *AbstractSimulation* class is shown in the listing 2.9.

Listing 2.9: AbstractSimulation class

```
public abstract class AbstractSimulation {
    public static AbstractSimulation simulation;

    public void run() {
        Customer jim = new Customer("Jim", 650);
        Customer mik = new Customer("Mik", 650);
        Customer crista = new Customer("Crista", 415);

        say("Jim calls Mik...");
        Call c1 = jim.call(mik);
        wait(1.0);
        say("Mik accepts...");
        mik.pickup(c1);
        wait(2.0);
        say("Jim hangs up...");
        jim.hangup(c1);
        report(jim);
        report(mik);
        report(crista);

        say("Mik calls Crista...");
        Call c2 = mik.call(crista);
        say("Crista accepts...");
        crista.pickup(c2);
        wait(1.5);
        say("Crista hangs up...");
        crista.hangup(c2);
        report(jim);
        report(mik);
        report(crista);
    }
    abstract protected void report(Customer c);
    protected static void wait(double seconds) {
        Object dummy = new Object();
        synchronized (dummy) {
            try {dummy.wait((long)(seconds*100)); }
            catch (Exception e) {}
```

```
38          }
39      }
40      protected static void say(String s){
41          System.out.println(s);
42      }
43  }
```

Finally, the concrete *BillingSimulation* class shown in the listing 2.10 implements the report method that it is used in run method.

Listing 2.10: BillingSimulation class

```
1   public class BillingSimulation extends AbstractSimulation {
2
3       public static void main(String[] args){
4          System.out.println("\n... Billing simulation 2 ...\n");
5          simulation = new BillingSimulation();
6          simulation.run();
7       }
8       protected void report(Customer c){
9           Timing t = Timing.aspectOf();
10          Billing b = Billing.aspectOf();
11          System.out.println(c + " has been connected for "
12                                     + t.getTotalConnectTime(c)
13                                     + " seconds and has a bill of "
14                                     + b.getTotalCharge(c));
15      }
16  }
```

As a result, the simulation is executed and it represents two phone calls with its time and billing values. The output of the execution is shown in the following way.

```
... Billing simulation ...
Jim calls Mik...
[new local connection from Jim(650) to Mik(650)]
Mik accepts...
connection completed
Timer started: 1184585606643
Jim hangs up...
connection dropped
Timer stopped: 1184585606845
Jim(650) has been connected for 202 seconds and has a bill of 606
Mik(650) has been connected for 202 seconds and has a bill of 0
Crista(415) has been connected for 0 seconds and has a bill of 0
Mik calls Crista...
[new long distance connection from Mik(650) to Crista(415)]
Crista accepts...
connection completed
Timer started: 1184585606846
Crista hangs up...
connection dropped
Jim(650) has been connected for 202 seconds and has a bill of 606
Mik(650) has been connected for 352 seconds and has a bill of 1500
Timer stopped: 1184585606996
Crista(415) has been connected for 150 seconds and has a bill of 0
```

To see this output in a clear way, we perform the table 2.1 that it has users, calls, the time that it is related with each user with its call and the billing for that call. This table shows the correct interaction among aspects because the precedence is defined in an appropriate way.

Table 2.1: User's calls with precedence

| Call | User | Time (sec.) | Billing |
|---|---|---|---|
| 1 | Jim | 202 | 606 |
| 1 | Mik | 202 | 0 |
| 1 | Crista | 0 | 0 |
| 2 | Jim | 202 | 606 |
| 2 | Mik | 352 | 1500 |
| 2 | Crista | 150 | 0 |

Nevertheless, if we do not declare the precedence in the Billing aspect then the advice of this aspect is executed before Timing's advice. Because, it is defined before in the source code. For this reason, the billing aspect calculates the payment in an incorrect time. So, the table 2.2 summarizes the results.

Table 2.2: User's calls without precedence

| Call | User | Time (sec.) | Billing |
|---|---|---|---|
| 1 | Jim | 202 | 0 |
| 1 | Mik | 202 | 0 |
| 1 | Crista | 0 | 0 |
| 2 | Jim | 202 | 0 |
| 2 | Mik | 353 | 0 |
| 2 | Crista | 151 | 0 |

On the other hand, as we said before AspectJ supports static crosscutting as well. First of all, AspectJ allows the declaration of members. This means that it adds data members and methods to classes. For instance, the listing 2.11 shows an interface with two methods but without variables.

Listing 2.11: Tagable interface

```
1 public interface Tagable {
2     void setTag(String tag);
3     String getTag();
4 }
```

After, we define an aspect which implements the methods of this interface and it adds a variable called *tag*. This aspect is able to use the getTag and setTag methods. For example, we use the setTag method in the line 12 of the listing 2.12, and the getTag method in the line 15.

Listing 2.12: TagAspect aspect

```
1 public aspect TagAspect {
2     private String Tagable.tag;
3
4     public String Tagable.getTag() {
5       return tag;
6     }
```

```
7    public void Tagable.setTag(String tag) {
8      this.tag = tag;
9    }
10   declare parents : Employee implements Tagable;
11   before(Tagable t) : target(t) && !within(TagAspect) {
12     t.setTag(thisJoinPoint.toString());
13   }
14   after(Tagable t) : target(t) && !within(TagAspect){
15     System.out.println(t.getTag());
16   }
17 }
```

In AspectJ, we can declare existing classes to implement an interface or extend a superclass. Aspects can be made dependent only on a base type or interface. This makes aspects more reusable. For instance, an Employee class is defined to implement the Tagable class in the line 10 of the listing 2.12.

Another way to make static crosscutting in AspectJ is specifying custom compile time warnings and errors using pointcuts. These declarations could ensure that system components follow certain programming practices. For instance, enforce that public access to instance variables is prohibited or avoid incorrect usages of an API. In a word, we can add the declarations shown in the listing 2.13 to the aspect.

Listing 2.13: Compile time warnings and errors

```
1 declare error : set(public * *) || get(public * *)
2               : "Nonpublic access is not allowed.";
3
4 declare warning : get(* System.out) || get(* System.err)
5                 : "Consider using Logger.log() instead.";
```

Finally, AspectJ converts a checked exception into a runtime exception. Also called *exception softening*. Sometimes it can be inconvenient to have to deal with checked exceptions. Involves a proliferation of try/catch blocks and throws clauses. For instance, if we have the advice declaration as in the listing 2.14, and the *openOutputFile* method throws an IOException.

Listing 2.14: Advice declaration

```
1 before() : doingIO() {
2     openOutputFile();
3 }
```

The solution could be add the declaration shown in the listing 2.15.

Listing 2.15: Exception softening

```
1 declare soft : java.lang.IOException : call(* *.openOutputFile());
```

After that, we can apply the advice anywhere without handling IOException.

In conclusion, AspectJ is a simple and practical aspect-oriented extension to Java. It provides a general purpose aspect-oriented programming language. AspectJ also offers a set of facilities and advantages as:

- Provide few new constructs like aspect, pointcut, after, before, etc..
- Enable two kinds of crosscutting like dynamic and static.
- Cleanly well-modularized implementations of crosscutting concerns.

- Understandable structure of a crosscutting concern, when written as an AspectJ aspect.
- Reduce redundancy.

### 2.1.2 Other approaches

As we mentioned above, we introduce some other approaches that are involved with aspect-oriented programming. We explain the main concepts of these approaches without focusing too much on the details.

- AspectS: approach to general-purpose aspect-oriented programming in the Squeak/Smalltalk environment [Hir03]. Based on concepts of AspectJ it extends the Smalltalk metaobject protocol to accommodate the aspect modularity mechanism. AspectS supports coordinated meta-level programming, addressing the tangled code phenomenon by providing aspect related modules. It shows great flexibility by not relying on code transformations (Smalltalk's syntax nor its virtual machine) but making use of metaobject composition instead. In AspectS, aspects are implemented with classes, so their instances act as regular objects. An aspect is applied to objects in the image by sending an install message to an aspect instance. The effects of an aspect to the system are reverted by simply sending an uninstall message to the same aspect instance that cause the system transformation. In AspectS, a join point denotes targets for the weaving process to apply computational changes to the underlying base system stated in advice objects. Join points of a point cut can be enumerated statically, or, due to the very open and reflective nature of the Smalltalk environment, collected dynamically by querying the system. AspectS does not introduce a dedicated pointcut language but takes advantage of the expressiveness of Smalltalk itself. Finally, advice objects associate code fragments with pointcuts and their respective join points descriptors that describe targets for the weaver to place these fragments into the system. This code fragments are represented by blocks, which means instances of *BlockContext*. AspectS also allows to execute crosscutting behavior using before and after methods (AsBeforeAfterAdvice), exceptions (AsHandlerAdvice) and around method (AsAroundAdvice). There another advice called AsAdviceQualifier that allows the description of dynamic attributes of a pointcut related to an advice.

  Finally, AspectS takes the advantage of Smalltalk like simplicity, elegancy, and its open architecture.

- CaesarJ: is an aspect-oriented language [AGMO06] which unifies aspects, classes and packages in a single construct that helps to solve a set of different problems of both aspect-oriented and component-oriented programming. CaesarJ combines the aspect-oriented constructs, pointcut and advice, with advanced object-oriented modularization mechanisms. From an aspect-oriented point of view, this combination of features is particularly well-suited to make large-scale aspects reusable, we can say, it enables aspect components. From a component-oriented view, on the other hand, CaesarJ is addressing the problem of integrating independent components into an application without modifying the component to be integrated or the application.

  CaesarJ has also virtual classes and propagating mixing composition that provide a means for abstraction, refinement and polymorphism of multi-class components, but they are not sufficient for integration of independently developed components with different modular structure. The problem of crosscutting integration of structure and behavior can be solved by the mechanisms for join-point interception and dynamic object extensions in form of wrappers. The unification of aspects and collaborations facilitates development of

reusable well-modularized aspects. Finally, in CaesarJ, treating an aspect as a class enables its free instantiation and flexible control over its scope of application.

- JAsCo: is an aspect-oriented extension to the Java language [JAs05]. This approach is primarily based upon two existing aspect-oriented software development (AOSD) approaches: AspectJ and Aspectual Components. Therefore, JAsCo seems to CaesarJ since it wants to enable aspect components, but CaesarJ separates the interface declaration from the actual implementation.

  JAsCo combines the expressive pointcut declarations of AspectJ with the aspect independency idea of Aspectual Components. JAsCo does however restrict the joinpoints that are possible to the public interface of the components, meaning public methods and fired events. The JAsCo language is kept as close as possible to the regular Java syntax and concepts. Only a minimal number of new keywords and constructs are introduced. The JAsCo language introduces two new concepts: aspect beans and connectors. An aspect bean is a regular Java bean that is able to declare one or more logically related hooks as a special kind of inner classes. Hooks are generic and reusable entities and can be considered as a combination of an abstract pointcut and advice. Because aspect beans are described independently from a specific context, they can be reused and applied upon a variety of components. Connectors have two main purposes: instantiating the abstract aspect beans onto a concrete context and thereby binding the abstract pointcuts with concrete pointcuts. In addition, a connector allows specifying precedence and combination strategies between the aspects and components. An schematic overview of JAsCo y represented by figure 2.3.

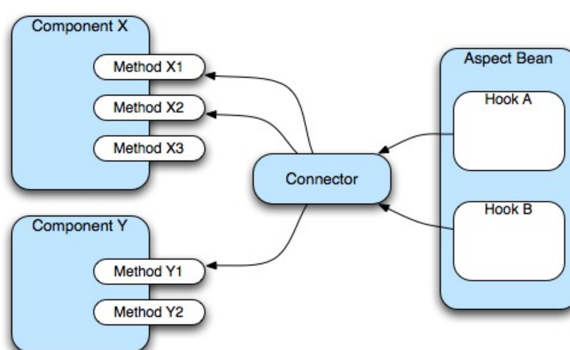


Figure 2.3: Schematic overview of JAsCo

- Reflex [TN05]: this approach aims at providing a versatile AOP kernel that can support different aspect languages simultaneously. Reflex supports core semantics, through proper structural and behavioral models. These models are resembled to *introductions* and *pointcut/advices* in AspectJ respectively. The kernel mediates among different approaches, detecting interactions among aspects and providing expressive means for their resolution. In order to reach that objective, the kernel has an architecture that consists of three layers like in figure 2.4, in which a transformation layer is in charge of basic weaving, supporting both structural and behavioral modifications of the base program; a composition layer make detection and resolution of interactions; and a language layer deal with the modular definition of aspect languages.

  In Reflex, an aspect consists of a cut and an action. A cut determines where an aspect applies, while an action specifies the effect of the aspect. Once again, we can say that

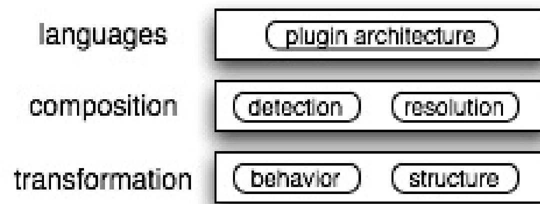


Figure 2.4: Architecture of an AOP kernel

a *cut* is resembled to a *pointcut* in AspectJ. In addition, an *action* seems to an *advice* in AspectJ. Reflex has an explicit link binding a cut to an action. This binding is represented by different types of links. One type of link binds a structural cut to an action, which can be either structural or behavioral (a structural link at load time). The second ones, binds a behavioral cut to an action (a behavioral link at runtime).

The AOP kernel provides means to modularly define aspect languages, either general-purpose or domain-specific. Due to that programmers can implement aspects with a good abstraction level. Accordingly, in Reflex, an aspect language can be implemented by a translator to kernel configuration, called a plugin. A plugin takes as input an aspect program written in a given language and outputs, either on-line or off-line, the adequate Reflex configuration: links, metaobject classes, selectors, etc., together with calls to the kernel API. The SOM (Sequential Object Monitors) and a subset of AspectJ were the aspect languages developed for the Reflex AOP kernel.

## 2.2 Feature interaction

Nowadays, the *feature interaction* term is a known concept. It is not only used in computer sciences but also in telecommunications. Actually, there is a definition of feature interaction problem in [PSC+02] on telecommunications domain. It was defined as the *unwanted interface between features running together in a software system*. In this domain, there are a lot of examples of features such as forwarding calls, placing callers on hold, or blocking calls [CV93]. Thinking about efficient and practical techniques for managing these interactions is really important in order to provide new telecommunications features.

There are different ways to view the feature interaction problem. Because, we can have several purposes that depend of the domain. In telecommunications [CV93], for instance, one view is *the software life-cycle view* where feature interactions are classified according to which phase of the software life cycle can best manage the interaction. The second view is called *the network configuration view*. In this view, feature interactions are classified depending on the configuration of network components and subscribers. Afterwards, in *the casual view*, feature interactions are classified based on the conditions that cause interactions. Then, a view, in which feature interactions are classified according to a layered architecture such as the open systems interconnection (OSI) stack is *the layered view*. Lastly, in *the organizational/operational view*, feature interactions are classified according to who is responsible for ensuring that they do not disrupt in the public network. All these mentioned views could be used by different approaches and also combine them.

There are several approaches to solving the feature-interaction problem in telecommunications. To a better understanding of the problem a division can be offered. In [CV93], the division consists in three classes: avoidance, detection and resolution. Avoidance looks at ways to prevent undesired feature interactions (e.g.: inadequate design

process). Detection assumes that feature interactions will be present, and determines methods for identifying and locating them. So, Resolution assumes that feature interactions will be present and detected, and looks at mechanisms for minimizing their potential adverse effects.

Once we saw the definition of feature interaction problem in telecommunications, we introduce the aspect-oriented programming ones. In AOP, the feature interaction problem appears when multiple aspects are specified and they could produce unexpected behavior in the system. So, after this, we are able to show which approaches are related to the feature interaction problem in AOP, and what they provide. Above all, we have to define a structure as well as in telecommunications. As a result, both detection and resolution sections are created. As we note, there is no avoidance section. This is because we are not dealing at the high level in the software development cycle. This thesis is targeted at the implementation level.

### 2.2.1 Detection

In order to present some approaches that deal with the detection of feature interactions, we look what approaches are, nowadays, dealing with that detection. As a result, we find some approaches that can be support this detection and we introduce each of them.

- In [DFS02] approach, they propose a three-phase model to feature interactions. This model consist of: programming, conflict analysis and conflict resolution. The *programming* phase means that the aspects which are part of an application are written independently, possibly by different programmers. The *conflict analysis* phase deal with an automatic tool to detect interactions among aspects and returns informative results to the programmer. This is the phase that we want to see in detail in this section. Lastly, the *conflict resolution* phase shows that the programmer resolves the interactions using a dedicated composition language. The result of this phase can be checked once again as in *conflict analysis* phase. The solution to this model is based on a generic framework for AOP, which is characterized by a very expressive crosscut language, static conflict analysis and linguistic support for conflict resolution. In the framework there is a weaving as a dynamic monitor, observing the execution of the program and inserting instructions according to execution states. It defines an observable execution and join points. In addition, it defines the aspect language, for instance rules like C $\triangleright$ I, which *C* is a crosscut and *I* is a program that is executed whenever the crosscut *C* matches the current join point.

  This approach proposes two different analysis detecting aspect interactions as follows.

  - *Strong independence* does not depend on the program to be woven. The aspect are independent for all programs. The advantage of this property is that it does not have to be checked after each program modification. In this analysis they use three definition and then an algorithm checks the strong independence based on the laws for aspects that are specified in [DFS02].
  - *Independence with respect to a program* takes into account the possible sequences of join points generated by the program to be woven. The advantage of this property compared to strong independence is that it is a weaker condition to enforce.

  They note that the first analysis is a sufficient but not a necessary condition. If two crosscuts *C* and *C'* match the same join point but their corresponding *I* and *I'* commute (i.e., executing *I* then *I'* is equivalent to executing *I'* then *I*) then the woven execution remains deterministic.

- As we mentioned in the section 2.1.2, Reflex [TN05] support automatic detection of aspect interactions limiting spurious conflicts. In addition, in that section we saw an

overview of the approach but now we introduce the proposed detection technique. Reflex defines two types of interactions such as static and dynamic interaction. They say that two behavioral links interact *statically* if the intersection of their hook-sets is not empty. On the other hand, they say that two behavioral links interact dynamically if they interact statically and they are both active at the same time. Since link ordering is resolved statically and activation conditions can be changed dynamically, Reflex adopts a defensive approach: any static interaction is reported, and must be considered by the developer, so that a dynamic interaction is never under-specified. As we know, Reflex has two types of links such as behavioral and structural link. The behavioral link interact was explained above and we continue with the other ones. So, they say that two structural links interact if the intersection of their class sets is not empty. They do not discriminate between static and dynamic interaction, because structural links are applied directly at load time.

Upon interactions, Reflex notifies an interaction listener. The default interaction listener simply issues warnings upon under-specification, informing the user that specification should be completed.

- The [DBA06] approach presents a language-independent technique to detect semantics conflicts among aspects that are applied at the same join point. The term *semantic* means the *behavior* of a component (aspect), rather than its syntax or structure. A *semantic conflict* is emerging behavior that conflicts with the originally intended behavior of one or more of the involved components. This approach introduces a formalization that enables us to express behavior and conflict detection rules. It is based on a resource-operation model, also called resource model, to represent the relevant semantics of advice, and detect conflicts among them.

  Now, we show in figure 2.5 the semantic analysis process and the relationships to the base system and advice that they propose in [DBA06].

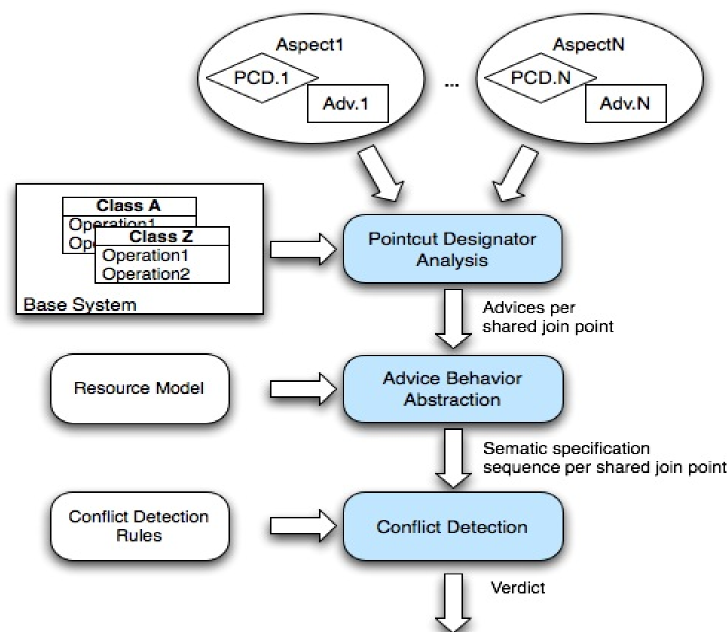


Figure 2.5: Overview approach

The figure 2.5 shows some aspects that contain advices and Pointcut Designators (PCD) at the top level. After that, we see that a base system and aspects are inputs of the *Pointcut Designatos Analysis* phase. At this level, all PCDs are evaluated with respect to the

base system and returns a set o join points with advices that are applied at the same join point. In the next level, the *abstraction* phase is executed. This phase has another input represented by the resource model. During the *abstraction* phase, the sequence of advices are transformed into sequences of resource-operation tuples per shared join point. To clarify the term resource-operation, they mention that a resource is in essence an *abstract data type*. On the other hand, an operation could be a *read* operation when, for instance, a logging advice accesses the arguments.

Finally, the *Conflict Detection* phase has two inputs: the operation sequences for resource per shared join point and the conflict detection rules. A conflict detection rule is a requirement on a resource. This is specified as a matching expression on the sequences of operations per resource. This rule can either be an assertion pattern, a combination of operations that must occur on a resource, or as a conflict pattern, a combination of operations that must not occur. So, this phase passes a verdict, i.e. whether a conflict is present or not, for each shared join point and for each combined sequence of operation per resource.

- AspectJ approach provides some visualizations to detect interactions as well. For instance, if we have a pointcut such as the listing 2.16.

  Listing 2.16: Pointcut example

```
1 pointcut relevantPoints(): call(* *.get*(..));
```

  After this, we specify two different advises that are applied at the same join point. As a result, AspectJ detects this interaction and something is printed out on the screen at the corresponding methods are executed. In addition, each advice has an arrow saying that it applies with a method, and what method as well.

### 2.2.2 Resolution

In this section, we present the resolution of feature interactions that were found in some approaches. Some of them are the same that we found in the detection of feature interactions. For this reason, we do not explain in detail the approaches again.

- As we said in section 2.2.1, the [DFS02] approach can deal with resolution of feature interactions. To provide this, the approach proposes some linguistic support. One is to specify how to compose inserts at the aspect level and the other ones is allows the user to control visibility of inserts by restricting the scope of aspects. The occurrences of rules of the form $C \rhd (I_1 \bowtie I_2)$ indicate potential interactions. They propose parallel operators of the form $\|_f$ to indicate that whenever a conflict occurs in the composition $A \|_f A'$, the corresponding inserts must be composed using *f*. Indeed, when inserts commute in a conflict, the inserts can be executed in any order ($I_1$ commute $I_2$) to allow the analyzer to produce an arbitrary sequence of $I_1$ and $I_2$. In addition, in order to control visibility, they propose a notion of scope for aspects. For instance, the command: *scope id Idset A* declares an aspect *A* with name *id* which can match only join points coming from an aspect whose name belongs to Idset.

- In this section, we present how the [TN05] approach applies resolution when there are aspect interactions. Above all, Reflex has the possibility to apply ordering and nesting. For instance, if links are mutually exclusive, specifying their ordering is not necessary. Otherwise, ordering must be specified by behavioral links. The interaction between two before-after aspects can be resolved in two ways: either one always applies prior to the

other (both before and after), or one "surrounds" the other, although AspectJ only supports wrapping. These alternatives can be expressed using composition operators dealing with sequencing and wrapping. Considering aspects that can act around an execution point (such as a caching aspect), the notion of *nesting* as in AspectJ appears: a nested advice is only executed if its parent around advice invokes proceed. In Reflex, link composition *rules* are specified using composition *operators*. The rule *seq*($l_1$,$l_2$) uses the *seq* operator to state that $l_1$ must be applied before $l_2$, both before and after the considered operation occurrence. The rule *wrap*($l_1$,$l_2$) means that $l_2$ must be applied within $l_1$, as clarified hereafter.

User composition operators are defined in terms of lower-level kernel operators. There are two kernel operators, *ord* and *nest* which express respectively ordering and nesting of link elements. In figure 2.6 we can see the sequencing and wrapping defined as follows: seq($l_1$,$l_2$) = ord($b_1$, $b_2$), ord($r_1$,$r_2$), ord($a_1$,$a_2$) and wrap($l_1$,$l_2$) = ord($b_1$,$b_2$), ord($a_2$,$a_1$), nest($r_1$,$b_1$), nest($r_1$,$r_2$), nest($r_1$,$a_2$)

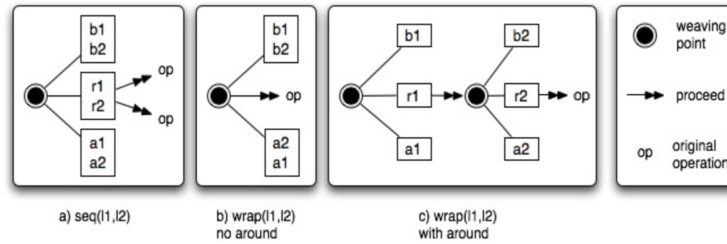


Figure 2.6: Ordering and nesting scenarios

Reflex makes it possible to define a handful of user operators for composition on top of the kernel operators. For instance, Seq and Wrap are binary operators that implement the *seq* and *wrap* operators.

Finally, when detecting link interactions, Reflex generates a hook skeleton based on the specified composition rules, similarly to figure 2.6. The hook skeleton is then used for driving the hook generation process: taking into account how link elements have to be inserted, with the appropriate calls to metaobjects. In order to support nesting of aspects with proceed, Reflex adopts a strategy similar to that of AspectJ, based on the generation of closures. So, in order to be able to do proceed, a metaobject is given an execution point closure (EPC) object, which has a proceed method, as well as methods for changing the actual arguments and receiver of the replaced operation. Accordingly, for each interaction scenario with nesting, Reflex generates closures embedding the composition resolution of the following nesting level, so that calling proceed on the EPC object results in the execution of the links at the nesting level below. This is done down to the deepest level where proceed results in the execution of the replaced operation. The top-level weaving points on figure 2.6 represent hooks, while nested weaving points represent closures.

- JAsCo provides a partial solution for the feature interaction problem. So, we suppose that multiple aspects are applied upon the same join point of an application, and in some way we need to order the execution of their behaviors. JAsCo partly addresses this open issue in AOSD by the specification of the advice executions in the connector. All advices of hooks that are triggered, are executed in the sequence specified in the connecter. If no explicit sequence is specified, they are executed in the order the hooks are instantiated. The precedence of the advices is enforced at run-time.

In comparison to AspectJ, JAsCo allows a more fine-grained control on the order in which aspects should be executed. For instance, some combinations of aspects require to specify that when aspect A is triggered, aspect B cannot be triggered. To this situation, JAsCo uses combination strategy that works like a filter on the list of hooks that are applicable at a certain point in the execution. Each combination strategy needs to implement the *validateCombinations* method, which filters the list of applicable hooks and possible modifies the behavior of individual hooks. Combination strategies can also be employed to change some properties of the hooks depending on dynamic conditions.

## 2.3 Summary

In this chapter we presented the main concepts of AOP and some approaches that are most well-know. We show a structured division of those approaches for a better understanding. Afterwards, we introduced the state-of-the-art of feature interaction. In that section we separated the feature interaction problem in two main sections such as detection and resolution. That is because not all approaches provide both detection and resolution. In general, all approaches encountered are focused at the interaction with aspects applied at the same join point.

# 3 Analysis with Soot tool

In this stage of the thesis, we already saw the problem statement that explains the main motivation of what we want to solve, and the context of feature interactions. So, at this time, we are able to focus in the first step of the objectives that we mentioned in the section 1.2. Therefore, we wish to do a static analysis of the woven result to reproduce possible control flow. This analysis is useful to know both all the possible control flows and what aspects are applied inside those control flows.

In order to tackle the mentioned analysis, we use a well-known tool called Soot [SOO07]. Therefore, in this chapter, we explain the main concepts of this tool and how it can help us with our objectives.

## 3.1 Overview of Soot

Soot [LLH04] is a byte-code analysis and transformation framework. Originally, Soot was a framework to experiment with analysis and optimizations of Java bytecode, and to provide a common infrastructure to compare results. However, nowadays, Soot allows analysis results to be encoded as class file annotations for use by other tools.

### 3.1.1 Functionality

One of the main advantage of Soot is that provides four different *Intermediate Representations* (IR) for analysis purposes. Of course, they provide different levels of abstraction when analyzing. These representations are described as follows:

- *Baf* is a streamlined stack-based representation of bytecode. Used to inspect Java bytecode as stack code. Baf is useful for bytecode based analyses, optimizations and transformations.

- *Jimple* is the principal representation in Soot. The Jimple representation is a typed, 3-address, statement based intermediate representation, which is suitable for analysis and transformations. In addition, Jimple has less instructions than Java bytecode. Jimple representations can be created directly in Soot or based on Java source code and Java bytecode.

- *Shimple* is a Static Single Assignment (SSA) form version of the Jimple representation. SSA form guarantees that each local variable has a single static point of definition which significantly simplifies a number of analyses. Shimple is almost identical to Jimple with the main exception of this single static point of definition. Therefore Shimple can be treated almost in the same way as Jimple.

- *Grimp* is similar to Jimple, but allows trees of expressions together with a representation of the new operator. In this respect Grimp is closer to resembling Java source code than Jimple is and so is easier to read and hence the best intermediate representation for inspecting disassembled code by a human reader. In addition, Grimp is a good starting point for decompilation.

In addition, Soot builds the following data structures during its analysis:

- *Scene* class represents the complete environment the analysis takes place in. Through it, you can set e.g., the application classes (The classes supplied to Soot for analysis), the main class (the one that contains the main method) and access information regarding interprocedural analysis (e.g., points-to information and call graphs).
- *SootClass* represents a single class loaded into Soot or created using Soot. SootMethod represents a single method of a class.
- *SootField* represents a member field of a class.
- *Body* represents a method body and comes in different flavors, corresponding to different IRs (e.g., JimpleBody).

These data structures are implemented using object-oriented techniques, and are designed as suitable and extensible abstractions.

In order to show an overview of the Soot framework [VRGH+00], we illustrate the complete workflow in the figure 3.1. As we see, many different compilers can be used to generate the class files. In addition, the framework takes the original class files as input, and produces optimized class files as output. These optimized class files can be used as input of Java interpreters, Just-In-Time (JIT) compilers, adaptive execution engines and Ahead-of-Time compilers.

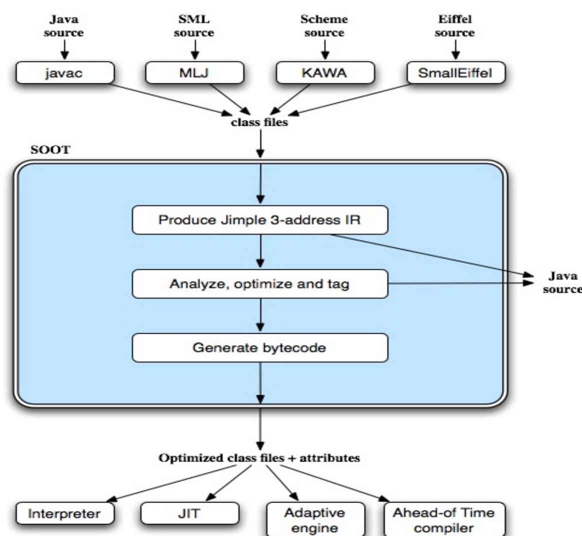


Figure 3.1: Soot overview

### 3.1.2 How to run Soot

Soot can be used as a stand-alone tool and executed either using command line or the Eclipse plugin. But, we choose to use the command line because it provides a more independent way

to use this tool (although it is a more complex). Soot can be invoked from the command line as follows:

*java [javaOptions] soot.Main [sootOptions] classname*

where *[javaOptions]* are general options for the Java Virtual Machine that executes Soot. Next, *[sootOptions]* represent the various options Soot accepts and *classname* is the class to analyze. In addition, Soot distinguishes among three kinds of classes: argument classes, application classes and library classes. First, the argument classes are the classes we specify to Soot. Secondly, the application classes are the classes to be analyzed or transformed by Soot, and turned into output. Finally, library classes are those classes that are refereed to by application classes but are not application classes. They are used in the analyses and transformations but are not themselves transformed or outputted.

As we mentioned above, there is another way to execute Soot using the Eclipse plugin like in the figure 3.2. In this plugin we must take the decision of what options represent the same meaning of the command line arguments.

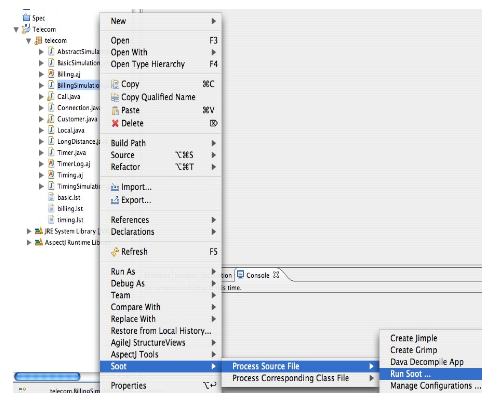


Figure 3.2: Soot run example

As we note, we just select which class we want to analyze and then choose every option to the analyses and transformations. For instance, if we choose the option *w* in the command line then we must click in the option *Whole-program mode* in the Eclipse plugin. The figure 3.3 shows a frame with the options.

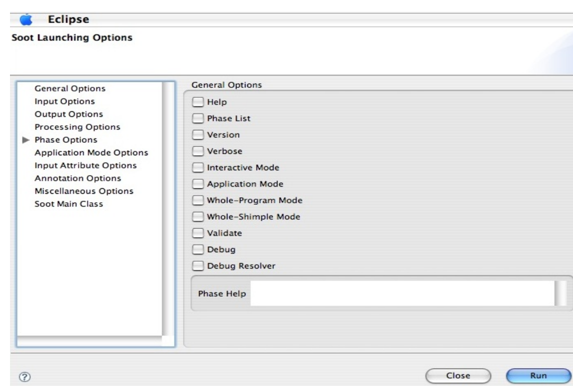


Figure 3.3: Soot options

Soot provides several modules to realize analyses and transformations, but we just take the advantage of the control flow graphs, call graphs and points-to analysis. These three important

modules provide to us nice results that we can combine them to solve our problem statement. In the following sections we explain each of them and finally we combine them.

## 3.2 Elements of the Analysis

### 3.2.1 Control flow graphs

Soot provides several different Control Flow Graphs (CFGs). The base class for these kinds of graphs is *UnitGraph*, an abstract class that provides facilities to build CFGs. There are three implementations of this abstract class: *BriefUnitGraph*, *ExceptionalUnitGraph* and *TrapUnitGraph*.

- *BriefUnitGraph* is very simple in the sense that it does not have edges representing control flow due to exceptions being thrown.
- *ExceptionalUnitGraph* includes edges from throw clauses to their handler (catch block, referred to in Soot as Trap), that is if the trap is local to the method body. Additionally, this graph takes into account exceptions that might be implicitly thrown by the virtual machine (VM) (e.g., ArrayIndexOutOfBoundsException). For every unit that might throw an implicit exception, there will be an edge from each of that units predecessors to the respective trap handler's first unit. Normally, this is the CFG used when performing control flow analyses.
- *TrapUnitGraph* like ExceptionalUnitGraph, takes into account exceptions that might be thrown. There are three major differences:
    1. Edges are added from every trapped unit (i.e., within a try block) to the trap handler.
    2. There are no edges from predecessors of units that may throw an implicit exception to the trap handler (unless they are also trapped).
    3. There is always an edge from a unit that may throw an implicit exception to the trap handler.

For reasons of simplicity, we use the *BriefUnitGraph* implementation in our approach: at this point, we do not consider exceptions as a part of our analysis. Our intention is to focus on the aspects that are applied in that control flow. So, to build a CFG for a given method body we simply pass the body to the CFG constructors, e.g., *BriefUnitGraph ctrlFlowGraph = new BriefUnitGraph(body)*. After that, Soot provides a graph structure that is created with the source code shown in the listing 3.1.

Listing 3.1: Control flow graph creation

```
1 SootMethod src =
2 Scene.v().getSootClass("Application").getMethodByName("main");
3 ExceptionalUnitGraph dgraph = new
4 ExceptionalUnitGraph(src.getActiveBody());
5 CFGToDotGraph gr = new CFGToDotGraph();
6 DotGraph viewgraph = gr.drawCFG(dgraph,src.getActiveBody());
7 dgraph.getBody().getUnits();
8 viewgraph.plot("dotCFG.dot");
```

The listing 3.1 shows the control flow creation upon a *main* method in the system called *Application*. So, the listing 3.2 shows the Java code that we need to analyze.

Listing 3.2: Application system

```
public class Application {
  public static void main(String[] args) throws IOException {
   Shopping s = new Shopping();
   s.doStuff(2);
  }
}
```

Finally, the graph is created in the line 8 (listing 3.1) through a *dot* file representation. In order to visualize this file, we use a graphic tool called *Graphviz*. The result of this graphic tool is shown in the figure 3.4 and then we can have an idea that how the control flow graph looks like.

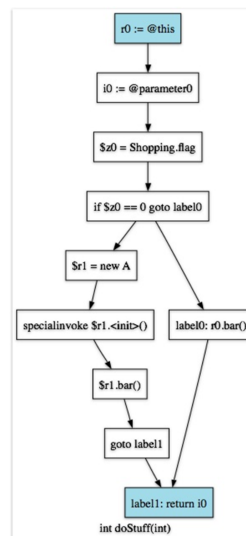


Figure 3.4: Control flow graph created by Soot

As a conclusion, we can say that Soot provides a detailed representation of the control flow inside of a method. However, when a *virtualinvoke* method appears in the control flow, it could be implemented by several classes. Since the dispatch to one of this implementation occurs based on the dynamic type of the receiving object, we are not able to know which one is the correct implementation. We will tackle this problem in section 3.2.3, using points-to analysis.

### 3.2.2 Call graphs

Soot includes a whole-program analysis framework for constructing call graphs [LLH04]. The information analyzed by this framework can be particularly useful to help developers understand their code and find bugs in it. In Soot, when a call graph is available (only in whole-program mode), it can be accessed through the environment class (Scene) with the method called *getCallGraph*. The simplest call graph is obtained through *Class Hierarchy Analysis* (CHA), for which no setup is necessary. CHA is simple in the sense that it assumes that all reference variables can point to any object of the correct type. The following listing 3.3 is an example of getting access to the call graph using CHA.

Listing 3.3: Call graph construction

```
Options.v().set_whole_program(true);
//  Code for setting up which classes to us as application classes.
```

```
3 CHATransformer.v().transform();
4 CallGraph cg = Scene.v().getCallGraph();
```

A call graph in Soot is a collection of edges representing all known method invocations. These invocations could be: explicit method invocations, implicit invocations of static initializers, implicit calls of Thread.run(), implicit calls of finalizers, implicit calls by AccessController, etc..

Each edge in the call graph contains four elements: source method, source statement (if applicable), target method and the kind of edge. There are different kinds of edges, e.g. static invocation, virtual invocation and interface invocation. The call graph has methods to query for the edges coming into a method, edges coming out of method and edges coming from a particular statement (edgesInto(method), edgesOutOf(method) and edgesOutOf(statement), respectively). Each of these methods return an *Iterator* over *Edge* constructs. Soot provides three so-called adapters for iterating over specific parts of an edge.

- *Sources* iterates over source methods of edges.
- *Units* iterates over source statements of edges.
- *Targets* iterates over target methods of edges.

So, in order to iterate over all possible calling methods of a particular method, we could use the code in the listing 3.4.

Listing 3.4: Call graph use

```
1 public void printPossibleCallers(SootMethod target) {
2   CallGraph cg = Scene.v().getCallGraph();
3   Iterator sources = new Sources(cg.edgesInto(target));
4   while (sources.hasNext()) {
5      SootMethod src = (SootMethod)sources.next();
6      System.out.println(target + " might be called by " + src);
7   }
8 }
```

Not only control flow graphs have dot file representation but also call graphs. The figure 3.5 shows an example of a dot file representation of a call graph. Since the call graph can become very large, we have removed some of branches and edges in order to keep the graph understandable.

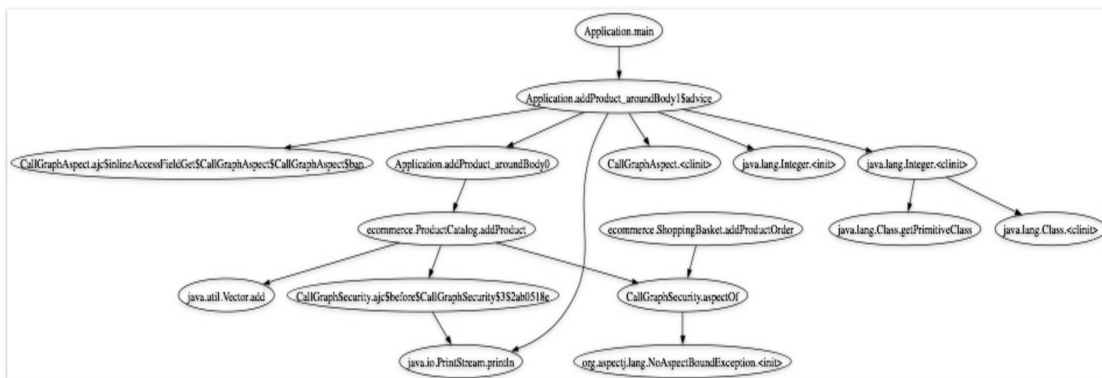


Figure 3.5: Call graph created by Soot (abridged)

In addition, Soot supports more specific information to analyze the code. It provides two more constructs for querying the call graph in a more detailed way: *ReachableMethods* and

*TransitiveTargets*. The *ReachableMethods* object keeps track of which methods are reachable from entry points. The method called *contains(aMethod)* tests whether *aMethod* is reachable and the *listener()* method returns an iterator over all reachable methods. The second object called *TransitiveTargets* is very useful for iterating over all methods possibly called from a certain method or any other method it calls (traversing call chains). The constructor accepts (aside from a call graph) an optional *Filter*. A *Filter* represents a subset of edges in the call graph that satisfy a given *EdgePredicate*. This *EdgePredicate* is a simple interface of which there are two concrete implementations such as *ExplicitEdgesPred* and *InstanceInvokeEdgesPred*.

### 3.2.3 Points-to analysis

In this section we present two additional frameworks for doing points-to analysis in Soot, the Soot Pointer Analysis Research Kit (SPARK) and Paddle frameworks. The goal of points-to analysis is to determine the set of objects pointed to by a reference variable or a reference object field. By computing such points-to sets for variables and fields, the analysis constructs an abstraction of the run-time memory states of the analyzed program. This is necessary in order to do many other kinds of analysis like alias analysis or for improving the precision of, for instance, a call graph. In our case, we are interested in improving the call graph because, as explained before, the virtual method calls have to be resolved.

Soot provides the *PointsToAnalysis* and *PointsToSet* interfaces which any points-to analysis should implement. The *PointsToAnalysis* interface contains the method called *reachingObjects(Local loc)* which returns the set of objects pointed to by *loc* as a *PointsToSet*. *PointsToSet* contains methods for testing for nonempty intersection with other *PointsToSets* and a method which returns the set of all possible runtime types of the objects in the set. These methods are useful for implementing alias analysis and virtual method dispatching. The current points-to set can be accessed using the *Scene.v().getPointsToAnalysis()* method.

To create the current points-to interface Soot provides three implementations: CHA, SPARK and Paddle. The first implementation was explained above, so we continue explaining briefly, the others implementations.

- *SPARK* (Soot Pointer Analysis Research Kit) is a framework for experimenting with points-to analysis in Java and supports both subset-based and equivalence based points-to analyses and anything in between. SPARK is very modular which makes it excellent for benchmarking different techniques for implementing parts of points-to analysis. This framework provides the following options:
  - *verbose* which makes SPARK print various information as the analysis goes along.
  - *propagator* SPARK supports two points-to set propagation algorithms, a naïve iterative algorithm and a more efficient worklist based algorithm.
  - *simple-edges-bidirectional* if true this option makes all edges bidirectional and hence allows an equivalence based points-to analysis.
  - *on-fly-cg* if a call graph is created on the fly which in general gives a more precise points-to analysis and resulting call graph.
  - *set-impl* describes the implementation of points-to set. The possible values are hash, bit, hybrid, array and double. Hash is an implementation based on the Java Collections hash set. Bit is implemented using a bit vector. Hybrid is a set, which keeps an explicit list of up to 16 elements and switches to bit vectors when the set gets larger. Array is implemented using an array always kept in sorted order. Double is implemented using two sets, one for the set of new points-to objects which have

not yet been propagated and one for old points-to object which have been propagated and need to be reconsidered.

  – *double-set-old* and *double-set-new* describes implementation of the new and the old set of points-to objects in the double implementation and double-set-old and double-set-new only have effect when double is the value of set-impl.

- *Paddle* is of comparable accuracy to SPARK for context-insensitive analysis, but also provides very good accuracy for context-sensitive analysis. In addition, it is implemented using Binary Decision Diagrams (BDD). The use of BDDs promises efficiency in terms of time and space, since BDDs provide a more compact set representation than the ones used in SPARK and other frameworks. Similar to SPARK, Paddle is equipped with a large set of options for configuring the analysis. The options verbose, set-impl, double-set-new, and double-set-old are the same as for SPARK. The *q* option determines how queues are implemented, and *enabled* option needs to be true for the analysis to run. The *propagator* option controls which propagation algorithm is used when propagating points-to sets, we leave it up to Paddle to choose and set it to *auto*. The *conf* option controls whether a call graph should be created on-the-fly or ahead of time. The implementation of Paddle is subset-based but equivalence-based analysis can be simulated by setting the *simple-edges-bidirectional* option to true. The last four options are the most essential for the working of Paddle so we describe them in some detail.

  – *bdd* option toggles BDD on or off. If true then use the BDD version of Paddle, if false do not. Default is false.
  – *backend* option selects the BDD backend. Either buddy (BuDDy), cudd (CUDD), sable (SableJBDD), javabdd (JavaBDD) or none for no BDDs. Default is buddy.
  – *context* option controls the degree of context-sensitivity used in the analysis. Possible values are: *insens*, *1cfa (1-control flow analisys)*, *kcfa*, *objsens*, *kobjsens* and *uniqkobjsens*. Therefore, with *insens* Paddle performs a context-insensitive analysis like SPARK. The *1cfa* value performs a 1-cfa context-sensitive analysis. *kcfa* performs a k-cfa context-sensitive, where *k* is specified using the *k* option. *objsens* and *kobjsens* values makes Paddle perform a 1-object-sensitive and k-object-sensitive analysis respectively. Finally, the *uniqkobjsens* value makes Paddle perform a unique-k-object-sensitive analysis. Default is *insens*.
  – *k* option specifies the maximum length of a call string or receiver object string used as context when the value of the context option is either of *kcfa*, *kobjsens*, or *uniqkobjsens*.

After all things considered, we saw that it is rather complicated to setup the two frameworks. In addition, the SPARK framework and Paddle framework provide a more accurate analysis at the cost of more complicated setup and speed. Nevertheless, we decide to use SPARK to analyze the virtual method calls because it supports them through the Variable-Type Analysis (VTA). The implementation of VTA [Kwo00] in Soot is based on the VTA algorithm presented in [SHR+00]. This implementation is almost the same but it has minor changes to improve precision. Therefore, VTA uses the name of a variable as its representative. For instance, when we analyze Jimple code, we have three kinds of references and they receive representative names in [SHR+00]. These names are: *ordinary references*, *field references* and *array references*. After that, this algorithm builds a *type propagation graph* where nodes represent variables and edges references flow of types due to assignments, including the implicit assignments due to method invocation and method returns. The *type propagation graph* contains at most one node for each variable with an object (reference) type. Once all of the nodes have been created, the algorithm

adds edges for all assignments that involve assigning to a variable with an object type. These may be either explicit assignments via assignments statements, and implicit assignments due to method invocation and returns. Finally, VTA is useful for further reducing the size of the call graph, and in getting more compaction by removing additional methods.

## 3.3 Combination of CFGs and CGs

In order to be able to see a complete and detailed control flow of a certain method, we need to combine both CFGs and CGs. The combination of these graphs can be realized with different techniques. These techniques are presented in the following sections.

### 3.3.1 Technique 1: Inlining

In this technique, a simple solution to combine CFGs and CGs is applied [RHS95]. So, we suppose that a CFG called *P()* has an invocation to another CFG called *Q()*. And for every call to *Q()*, inline the CFG of *Q()*. The advantage of this technique is that it distinguishes between different calls to the same function. Nevertheless, if there is several calls to e.g. *Q()*, we loose efficiency because the CFG grow up exponentially. Another disadvantage is when infinite graph is created in case of a recursive call.

### 3.3.2 Technique 2: Connecting

Another technique can be making a new "supergraph" [RHS95]. To this technique we assumed the same example that in the first technique. In this case, we have to replace each call from *P()* to *Q()*. In order to satisfy this replacement two main additions have to be done:

- Add an edge from point before the call (call point) to Q's entry point.
- Add an edge from Q's exit point to the point after the call (return point).

The main advantage of this technique is that a graph of each function included exactly once in the "supergraph". In addition, the "supergraph" works for recursive functions (although local variables need additional treatment). Once again, there is still a problem with this technique called the *unrealizable paths problem*, where dataflow facts can propagate along infeasible control paths. This means that one CFG could transfer its control to another CFG and then goes to another CFG that it is different to the first one. The figure 3.6 shows an example of this situation.

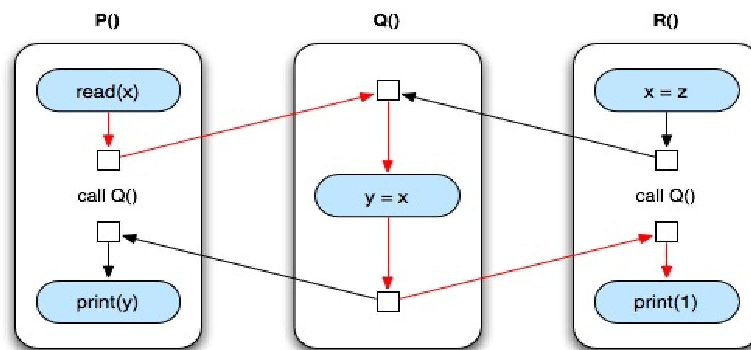


Figure 3.6: Unrealizable path

As we note in the figure 3.6, the CFG of *P()* calls *Q()*. After the execution of *Q()*, the control flow can return to the CFG of *R()*, instead of *P()*. In this way, the result of the analysis can be wrong (the unrealizable path is represented by red line).

### 3.3.3 Technique 3: Connecting + labelling

In order to solve the problem discussing above, in which unrealizable paths could occur, we introduce a technique that adds labels to the combination between CFGs and CGs [RHS95]. Taking the figure 3.6 as base case, we must add additional labels for each call *i*. For instance, when *P()* calls to *Q()* a label $(_1$ has to be added and when its return appears a label $)_1$ has to be added. Therefore, to the next call we use the labels $(_2$ and $)_2$ and so on. Finally, the adding result is shown in the figure 3.7.

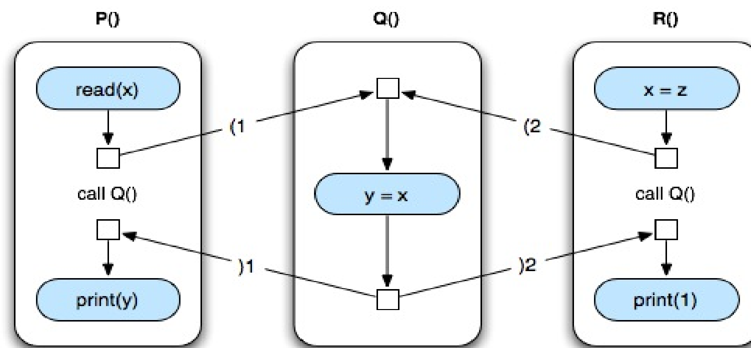


Figure 3.7: Valid paths

Once we have these additional labels, we make sure that when a label $(_i$ is stacked, it must to be followed by a label $)_i$. Otherwise, any other combination of labels is not allowed and the path is discarded.

### 3.3.4 Practical Realization

As we note during the practical realization, Soot does not provide the combination of CFG and CG directly. So, above all, we must change the Soot's source code and re-compile it again. This change is necessary to avoid some lack of information. The problem is that Soot resets some important structures when we use both CFGs and CGs. Therefore, the code line *PackManager.v().writeOutput();*, in the method called *run* at the Main class, has to be commented.

After all considered, we focus on our implementation with respect to the combination between CFGs and CGs. To do this, we use Soot as a stand-alone tool and we execute it using command line. Therefore, when we use command line, we must pass the arguments to Soot. Afterwards, Soot makes analyses and transformations, depending of those arguments. The arguments are:

- *-src-prec class*. Sets source precedence to *class* files.
- *-f jimple*. Set output format for Soot. In our case, we produce *jimple* files.
- *–keep-line-number*. Keep line number tables.
- *-d /Users/pabloquiroga/Documents/workspace/CallGraph/sootOutput/*. Store output files in a given directory.

- *-app*. Run in application mode.
- *-w*. Run in whole-program mode.
- *-p cg.spark on*. Set *SPARK* option to on.
- *-p cg.spark vta:true*. Set *VTA* option to true.
- *Caller*. The class to analyze.

In addition, we use an extra argument such as *-Xmx400m* to the virtual memory. Afterwards, we create the main method to invoke Soot tool. The listing 3.5 shows the main method.

Listing 3.5: Combination between CFGs and CGs

```
1 public class Main {
2  public static void main(String[] args) throws IOException {
3   soot.Main.main(args);
4   SuperGraph sg = new SuperGraph("SuperGraph.dot");
5   HashSet<String> tgtCallReturn = new HashSet<String>();
6   sg.cfg(args[args.length-1], "main", tgtCallReturn, 0);
7   sg.closeSuperGraph();
8  }
9 }
```

As we note, in the line 3 (listing 3.5), Soot's modified class is called with the arguments detailed above. This call executes Soot tool and its results are used in our *SuperGraph* class. This class is in charge of the combination's creation. It has a variable called *dsg* that represents a *DotSuperGraph* object. The *DotSuperGraph* object represents a *dot* file and manipulate all the graph's nodes and the connectors among them. In the line 6 (listing 3.5), we create the first control flow graph with the *main* method. Afterwards, the recursive *cfg* method goes through the control flow looking for an invocation statement. If there is such invocation, we create a new control flow graph whose origin is connected with the invocation of the first control flow. In addition, we add a connector between the end of the control flow created at the second time, and the return of the control flow which made the invocation.

Finally, in the line 7 (listing 3.5), we finish the graph's combination closing the *dot* file representation. This file is shown in the figure 3.8.

All in all, we can see a detailed control flow graph that shows the possible paths in the main method of a certain class. Of course, inside of this control flow could appear some aspects that are woven with the base code. In addition, every possible path has to start at the ClassName.ENTER.MethodName node and finishes at the ClassName.EXIT.MethodName node.

## 3.4 Summary

In this chapter, we have presented an overview of Soot tool. In addition, we saw some advantages provided by Soot like control flow graphs, call graphs and point-to analysis. These are not the only ones but the more important to our dissertation. Afterwards, when we tried to combine CFGs with CGs, we encountered some difficulties and then we showed how to solve them. Finally, we present our implementation about the combination of those graphs.

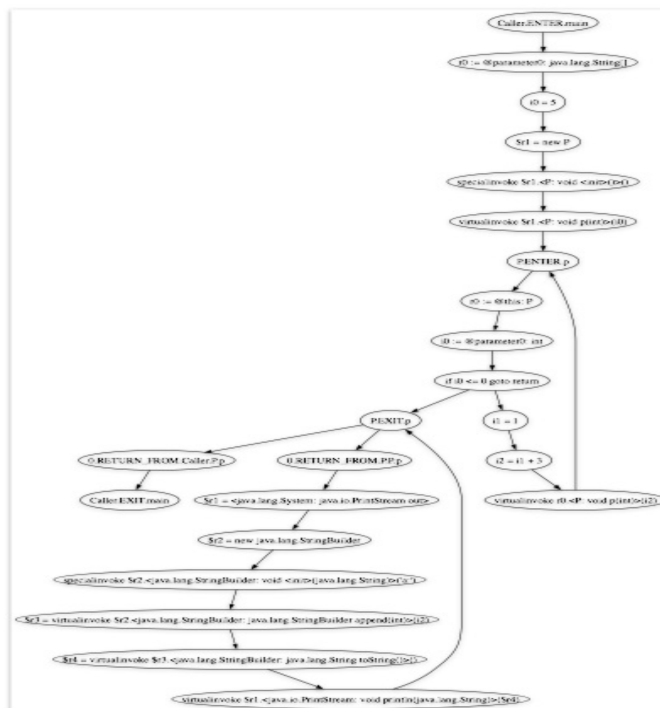


Figure 3.8: Combination between CFGs and CGs

# 4 Formal Documentation

In this chapter we present a formal documentation of aspects to specify in what control-flow configuration these aspects should be applied. We define each specification as a *policy*. For instance, one policy could specify that a *login* aspect should be executed in every control flow path of the *main* method.

In order to express those policies in a general manner, we use predicate logic in our approach. As we know, predicate logic is considered a knowledge representation language [Mag05]. It allows us to represent complex facts about the world, and to derive new facts in a way that guarantees that, if the initial facts were true then so are the conclusions. In addition, predicate logic is a well understood formal language, with well-defined syntax, semantics and rules of inference. For instance, we are able to build up complex expressions by combining atomic propositions with the logical connectives. The figure 4.1 shows each of them.

Table 4.1: Logical connectives

| Symbol | What it is called | What it means |
|---|---|---|
| $\neg$ | negation | 'It is not the case that. . .' |
| & | conjunction | 'Both. . . and . . .' |
| $\vee$ | disjunction | 'Either. . . or . . .' |
| $\rightarrow$ | conditional | 'If . . . then . . .' |
| $\leftrightarrow$ | biconditional | '. . . if and only if . . .' |

In addition, we are allow to use quantifiers like *for all* and *exists* that are represented by the symbols such as $\forall$ and $\exists$ respectively.

## 4.1 Policy language

In our approach, we use some predefined predicates that represent knowledge relevant to our problem domain, i.e. control flow, methods, advices, etc. Policies can then be expressed as logical formulae that employ these predicates, as well as the standard logical connectives and quantifiers.

### 4.1.1 Predefined predicates

In order to represent some knowledge about our problem domain we predefine certain predicates. These are:

- *path(<method>,<path>)*. This predicate holds when *<path>* is a path from *<method>*.

- *member(<method>,<path>)*. This predicate holds when <*method*> is on <*path*>.
- *matches(<methodpattern>,<method>)*. This predicate holds when <*method*> is matched by a certain <*methodpattern*>.
- *adviceof(<method>,<aspect>)*. This predicate holds if <*method*> is an advice method defined in <*aspect*>.

A <*path*> argument represents a sequence of methods. For instance, a <*path*> = *{methodA,methodB}* represents a control flow path through *methodA* and *methodB*, i.e. the *methodA* method calls to the methodB method (see figure 4.1).

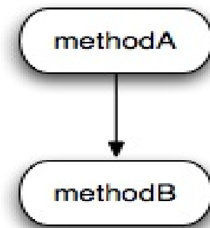


Figure 4.1: Example argument. (path={methodA,methodB})

In order to select methods, we will use the well-known method patterns from AspectJ's pointcut language. This means that, as an argument <*methodpattern*>, we are able to use statements according to the grammar of 4.1. The matching semantics of these method patterns (as embodied by the predicate *matches/2*) is also reused from AspectJ.

Listing 4.1: AspectJ's pointcut language (method patterns)

```
<methodpattern> ::= [<modifierspattern>] <typepattern>
                [<typepattern> "."] <idpattern> <formalspattern>
                [<throwspattern>]
<modifierspattern> ::= ["!"] <javamodifier> (<modifierspattern>)*
<typepattern> ::= <idpattern> ["+"] ("[]")*
          | "!" <typepattern>
          | <typepattern> "&&" <typepattern>
          | <typepattern> "||" <typepattern>
          | "(" <typepattern> ")"
<idpattern> ::= (<letter>)+ ["*"] ["."] [<idpattern>]
<formalspattern> ::= <typepattern>
          | ".." ("," <formalspattern>)*
<throwspattern> ::= THROWS <typepattern> ("," <typepattern>)*
<javamodifier> ::= PUBLIC | PRIVATE | PROTECTED | STATIC
          | SYNCHRONIZED | FINAL
<id> ::= <letter> (<letter> | <digit>)*
<letter> ::= "A" .. "Z"
          | "a" .. "z"
<digit> ::= 0 .. 9
```

Based on these predicates, other higher level predicates can be built, in order to facilitate the definition of policies.

### 4.1.2 Derived predicates

As we mentioned above, the policies help to the developer to specify what aspects should be applied. Nevertheless, a policy could specify a more general case without aspects. For instance,

if we want to say that a certain method *A* must apply in the control flow of another method *B*, we express this situation as follows:

must(<methodpattern>,<methodpattern>)

The *must* predicate is developed with the combination of the predefined predicates. In addition, we use the quantifiers like ∀ and ∃. So, the new predicate specify that the *A* method must apply in the control flow of the *B* method.

must(A,B) ↔ ∀ X: matches(B,X) → [∀ P: path(X,P) → (∃ Y: matches(A,Y) → member(Y,P))]

This policy checks if there is a certain method A inside the control flow of method B. This verification has to be realized for every path that starts from method B.

## 4.2 Problems Revisited

As we have seen in the section 1.1 our motivation introduced several cases of interactions. First of all, we presented a simple case in which the control-flow is affected by an aspect. So, to provide support to this case, we will add a new predicate to our approach such as *mustApply(<methodpattern>, <method>)*. This predicate express that an aspect that is selected from the pattern *<methodpattern>* must apply in the control flow of a certain method. Therefore, the predicate *mustApply* can be expressed with the policy language as follows:

mustApply(A,B) ↔ ∀ X: matches(B,X) → [∀ P: path(X,P) → (∃ Y: adviceof(Y,A) → member(Y,P))]

In short, this policy checks that for every path that starts from the method B, the advices of aspect A must be applied (if there is some one). Secondly, we introduced an interaction where two aspects are depending each other. In this case, we consider a predicate with three arguments to specify the aspects that are involved and the method where the control flow starts:

*mustDepend(<methodpattern>, <methodpattern>, <method>)*

Therefore, as we said before it is possible to express this new predicate with the policy language as follows:

mustDepend(A1,A2,B) ↔ ∀ X: matches(B,X) → [∀ P: path(X,P) → (∃ Y: adviceof(Y,A1) → ∃ Z: adviceof(Z,A2) & member(Y,P) & member(Z,P))]

The *mustDepend* policy checks the same paths that the previous policy. But, in this case, it looks for a certain advice of an aspect in the control flow and then it tries to find advices of the dependent aspect. Finally, the last interaction of our motivation was related on the use of two excluded aspects each other:

mustExclude(<methodpattern>, <methodpattern>, <method>)

Therefore, the policy is represented as follows:

mustExclude(A1,A2,B) ↔ ∀ X: matches(B,X) → [∀ P: path(X,P) → (∃ Y: adviceof(Y,A1) → ¬ ∃ Z: adviceof(Z,A2) & member(Y,P) & member(Z,P))]

As we noted, this policy is similar to the previous one with the different that it must not apply the second aspect when the first one is applied.

## 4.3 Implementation of a light-weight logic engine

At this time we know that policies can express what should occur in the control flow of a certain method. To help the developer, these policies have to be analyzed to determine whether are true or false (i.e. whether they are honoured or violated). Therefore, we create a light-weight logic engine to evaluate the logic formulae that make up the policy. To better understand the design of our logic engine we have developed a graphical diagram. Following this, the main concepts

are explained such as general concepts, goals and predicates of the logic engine. Of course, we include some illustrations of its functionality as well.

### 4.3.1 Proposed engine

In order to evaluate the logic formulae, we present an overview of a logic engine that involve predicates, connectives and goals. The figure 4.2 shows the main classes that are involved in the design of this engine. We will explain each of the concepts below.

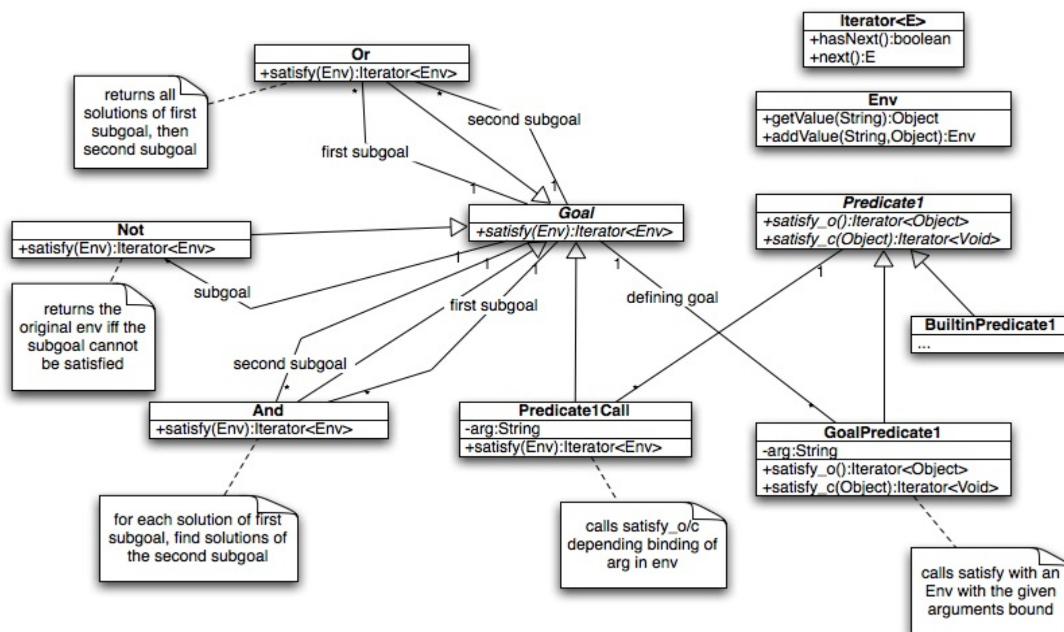


Figure 4.2: An overview of a logic engine

### 4.3.2 General concepts

One of the main concepts of our logic engine is based on the environments. An *environment* provides a mapping from a certain set of *variables* to *values*. We use *strings* to represent *variables* and regular *objects* to represent *values*.

In order to be able to show different possible environments, we use an auxiliary structure to iterate through the environments. Therefore, the figure 4.2 shows this structure as the generic *Iterator<E>* class (E represents the type of elements that can be iterated). By the same token, an environment is represented by *Env* class.

### 4.3.3 Goals

A *goal* represents something that we want to verify. It can be satisfied starting from an initial environment, and it produces an extended environment for each encountered solution. The figure 4.2 shows the *Goal* interface.

The bindings in the initial environment represent the *a priori* assumptions that we use when satisfying the goal. In contrast, the bindings that are added in the extended environment represent the additional (*a posteriori*) assumptions that are necessary in order to obtain a

solution. If the assumptions from the initial environment already prohibit a solution, than no environments are returned. As an example, consider the goal *female(X)*. Given an initial environment that is empty, this goal can be satisfied in a large number of ways, represented by different resulting environments: *X=maria*, *X=analia*, etc. Given an initial environment *X=rosa*, this goal can only be satisfied in one way, namely the initial environment. And given the initial environment *X=pablo*, the goal cannot be satisfied, so no resulting environments are produced.

The logical connectives can be modeled as new kinds of goals that can each be defined in terms of their subgoal(s). So, new kinds of goals are defined as the And, Or, and Not classes. We briefly explain each of them:

- The *And* goal is satisfied if and only if both subgoal are satisfied. This goal tries to satisfy the first subgoal, and for each encountered solution, it tries to satisfy the second subgoal as well. The resulting environments represent solutions for both subgoals, so these will be returned by the resulting iterator of the *And* goal (or an empty iterator if there is no common solution).

- The *Or* goal is satisfied if and only if either one of the subgoal is satisfied. To obtain these solutions, the resulting iterator first returns the solutions of the first subgoal, and afterwards the solutions of the second subgoal.

- The *Not* goal returns the original *environment* if and only if the subgoal cannot be satisfied. Otherwise the empty iterator is returned.

### 4.3.4 Predicates

A predicate represents a goal that is parameterized with formal arguments. Some examples are: *male/1*, *female/1*, *parent/2*, etc. where the number after the name represents the number of arguments. In our engine, we use the interface *Predicate1* for a predicate with one argument, *Predicate2* for a predicate with two arguments, etc. (in figure 4.2 only *Predicate1* is shown). The implementation of the interfaces has to provide a way to satisfy the predicate based on the possible state of the arguments. In case of one argument, that argument can either be "open" (i.e. unknown or uninstantiated) or "closed" (i.e. known). The interface therefore includes a method *satisfy_o()*, for the case where the argument is open: in that case possible argument values will be generated. In case the argument is closed, the interface provides the method *satisfy_c(Object)*, which takes the argument value as a parameter, and which will return an iterator of the meaningless type (*Void*) to indicate if there are solutions.

We distinguish two kinds of implementations for the *Predicate* interface. Built-in predicates will be implemented in Java, and will, in the context of this thesis, employ the Soot analysis in order to determine their result. On the other hand, the derived predicates are implemented by employing some goal as their definition, and by stating the formal arguments that are used in their definition. A derived predicate with one argument is modeled as the call *GoalPredicate1*: it will provide an implementation of the methods *satisfy_o()* and *satisfy_c(Object)* by calling the *satisfy* method of the defining goal with an environment where the argument is respectively unbound or bound to the given argument value. In order to show an example of *GoalPredicate1* predicate, we define the predicate *macho/1* using the goal represented by the conjunction of *male(X)* and *latin(X)*. Therefore, the arguments of the new predicate will be named with *X*. So, the predicate defined by another goal will be *macho(X)*.

Finally, when a predicate is called with certain arguments values, this call constitutes another kind of goal (recall the example $female(X)$ that we employed previously). In case of a predicate that takes one argument, this is represented by the class *Predicate1Call*. It will

implement the *satisfy* method of the *Goal* interface by employing the *satisfy_o* and *satisfy_c* method of the predicate. If the argument value is unbound in the initial environment, *satisfy_o* is employed; otherwise *satisfy_c* is called.

## 4.4 Summary

In this chapter, we have presented a formal documentation of aspects to specify what aspects should be applied. In order to be able to provide this documentation we took the concept of predicate logic. Thus, we could use logical connectives and quantifiers to have more expressiveness in our approach. Afterwards, a policy language was defined. In this language, an example of a policy definition was explained with own predicates. As we know, a policy has to be verified whether is true or false. Therefore, a light-weight logic engine was created. This engine is able to accept new goals to extend its functionality. Finally, the logic engine helps us to know whether a certain policy was violated or not.

# 5 Conclusions

The main goal of this dissertation is to provide support for the management of control-flow interactions among the aspects deployed in a system. We have shown different interactions that motivated this dissertation. Starting with a simple aspect applied at the control-flow and continuing with aspects that are depending or excluding each other. After we explained the motivation we take a look at the state-of-the-art of feature interaction. We found that, in most of the cases, all approaches encountered focus on the interactions with aspects applied at the same join point. In order to support broader, control flow interactions a technique was proposed in chapters 3 and 4. Section 5.1 below, presents an evaluation of the applied technique. Afterwards, contributions of this dissertation are shown. Finally, we conclude with some future work.

## 5.1 Evaluation of the Technique

The technique involves a number of consecutive steps, each with its own objectives. The objectives of the different steps were solved in a successful manner. In each stage we could of course find a possible solution and with its advantages and disadvantages. So, we display the conclusions of each step below.

### 5.1.1 Static analysis

One of the points of our technique consisted in a static analysis of the woven result to reproduce possible control flow. This analysis was successful because we are able to create all the possible control flows, and we can see what aspects are applied inside those control flows as well. This success was possible with the support of Soot tool. We took control flow graphs and call graphs from Soot, because we think that they provide good representations and a detailed analysis. We used these graphs to provide a combined graph that holds all possible control flow. However, Soot does not provide the combination of both graphs by itself. Therefore, this combination was a big challenge.

### 5.1.2 Formal documentation

Another point of our technique required a formal documentation of aspects to specify what aspects should be applied in a certain control flow. This allows the developer to document his assertions about the aspects in a system, in order to avoid unexpected interactions. To express this documentation we employ the concept of predicate logic. As such, a powerful and general language is available to write complex control-flow policies. In addition, we remark that this technique is not restricted to aspect systems alone, it could be applied in a standard (e.g. object-oriented) software system.

### 5.1.3 Logic engine

Finally, the last point of the technique was to create an algorithm to detect violations of the policies on a certain control flow. Since we use logic formulae from predicate logic to express the policies, we need to be able to evaluate these logic statements in order to determine their truth value. To this end, a light-weight object-oriented logic engine was created. This engine allows the definition of built-in predicates in order to employ the results of the static analysis. Additionally, users can build their own predicates on top of existing ones, in order to abstract and reuse complex definitions.

## 5.2 Contributions

In this section, we comment all the contributions of this dissertation. Above all, we propose a technique for managing control-flow interactions between aspects in a software system, an area that has not been considered before (to the best of our knowledge). This technique involves a static analysis of the woven result, using existing, well-know tools (i.e. Soot [SOO07]). We combine different analysis results, namely control flow graphs and call graphs, in order to obtain a result suitable for our purposes. Moreover, we propose a set of predicates, as in predicate logic, that represent relevant situations regarding the control-flow relations of methods and advices. For instance, with these predicates, we can specify that we must apply certain aspects in the control flow of all methods matched by a certain method pattern. In addition, we are able to define several complex policies because we can combine previous policies with each other. Last but not least, we present an application made in Java that implements the logic engine. This application checks if each policy defined by the user is satisfied. As such, the developer obtains a complete tool for the management of control flow interactions between aspects in his software system.

## 5.3 Future work

Following the investigations described in this dissertation, we think that there are several lines of research arising from this work which still require further investigation:

- The combined graph visualization, as we described in section 3.3 has still some scalability issues. As we have seen in the figure 3.8, resulting visualizations are overloaded in information on every node of the graph. This might be solved by presenting the information in a more useful way. For instance, we could separate certain relevant information such as the method name that is invoked, its arguments, to what class belongs this method, etc.

- The static analysis of the woven result could be improved. In the case where we introduce the interaction on two aspects that depend on each other, an aspect could cut part of the flow control of another aspect (Although we have found both aspects in this control flow). We could solve this problem by doing the static analysis before and after applying the aspects. Afterwards, we would have to compare both control flows to determine whether this interaction occurs.

- Results of the static analysis currently do not consider exceptions. Therefore, we could add these exceptions with implementations that Soot tool provides.

- The logic engine shown in section 4.3 could be implemented with another language different from Java. Since our policies are expressed using predicate logic, it would be

possible to be implemented it by means of a full-blow logical programming language such as PROLOG. This would allow for advanced techniques, such as cutting of the search tree, or retraction of assertions, etc.

- Evaluate on a large case study, e.g. J2EE applications, it could be a more powerful way to use our approach.

# Bibliography


[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. 3880:135–173, 2006.

[CV93] E. Jane Cameron and Hugo Velthuijsen. Feature interactions in telecommunications systems. *Communications Magazine, IEEE*, 31(8):18–23, Aug 1993.

[DBA06] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In L. Bergmans A. Nedos A. Rensink R.Chitchyan, J. Fabry, editor, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, Lancaster University, Jul 2006.

[DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. Detection and resolution of aspect interactions. Rapport de recherche 4435, Inria, Avril 2002.

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005.

[Hir03] Robert Hirschfeld. Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, R. Unland, editors, Objects, Components, Architectures, Services, and Applications for a Networked World, pp. 216-232, LNCS 2591, Springer, 2003.

[JAs05] Jasco language reference 0.8.6. `http://ssel.vub.ac.be/jasco/lib/exe/fetch.php?cache=cache&media=documentation:jasco.pdf` , 2005. Host site, which contains a document that gives an overview of the JAsCo language.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072, pages 327–353. Berlin, June 2001.

[Kwo00] Felix Kwok. Notes on vta implementation in soot. `citeseer.ist.psu.edu/kwok00notes.html`, 2000.

[Lad03] Ramanivas Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[LLH04] Jennifer Lhoták, Ondřej Lhoták, and Laurie Hendren. Integrating the Soot compiler infrastructure into an IDE. In E. Duesterwald, editor, *Compiler Construction, 13th International Conference*, volume 2985 of *LNCS*, pages 281–297, Barcelona, Spain, April 2004. Springer.

[Mag05] P.D. Magnus. forallx: An introduction to formal logic. www.fecundity.com/logic/, 2005.

[MW99] K. Mehner and A. Wagner. An assessment of aspect language design. Young Researchers Workshop, First International Symposium on Generative and Component-Based Software Engineering, September 1999.

[O'R04] Graham O'Regan. Introduction to aspect-oriented programming. *O'Reilly OnJava.com*, January 2004.

[PSC+02] Elke Pulvermueller, Andreas Speck, James Coplien, Maja D'Hondt, and Wolfgang De Meuter. Feature interaction in composed systems. In *ECOOP '01: Proceedings of the Workshops on Object-Oriented Technology*, pages 86–97, London, UK, 2002. Springer-Verlag.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.

[SHR+00] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for Java. *ACM SIGPLAN Notices*, 35(10):264–280, 2000.

[SOO07] Soot: a java optimization framework. www.sable.mcgill.ca/soot/ , (version 2.2.3) 2007. Host site, which contains documentation and open source.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[Tel03] A simple telecom simulation. www.eclipse.org/aspectj/doc/released/progguide/examples-production.html , 2003. Host site, which contains example documentation.

[TN05] Éric Tanter and Jacques Noyé. A Versatile Kernel for Multi-Language AOP. In Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005), LNCS, Springer-Verlag, Tallin, Estonia, September, 2005.

[VRGH+00] Raja Vallee-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In *Computational Complexity*, pages 18–34, 2000.