Un interpréteur extensible pour le prototypage des langages d'aspects

Thèse de Doctorat

Discipline: Informatique

Spécialité : Informatique

Présentée
et soutenue publiquement par

Ali Assaf

le 21 octobre 2011, à l'École des Mines de Nantes,
devant le jury ci-dessous

Président	:	Frédéric Benhamou, Professeur	Université de Nantes
Rapporteurs	:	Roland Ducournau, Professeur	Université de Montpellier 2
		Christian Percebois, Professeur	Université de Toulouse 3
Examinateurs	:	Lionel Seinturier, Professeur	Université de Lille 1
		Frédéric Benhamou, Professeur	Université de Nantes
		Pierre Cointe, Professeur	Ecole des Mines de Nantes
		Jacques Noyé, Maître-Assistant	Ecole des Mines de Nantes

Équipe d'accueil : ASCOLA - INRIA/EMN, LINA UMR CNRS 6241

Laboratoire : département informatique de l'école des mines de nantes
4, rue Alfred Kastler, BP 20722 – 44307 Nantes, Cedex 3.

N° ED 0366-XXX

Un interpréteur extensible pour le prototypage des langages d'aspects

A Common Aspect Languages Interpreter

Ali Assaf

⋈

favet neptunus eunti

Université de Nantes

Ali Assaf

Un interpréteur extensible pour le prototypage des langages d'aspects

222+iv p.

This document was edited with these-LINA v. 2.7 LaTeX2e class of the "Association of Young Researchers on Computer Science (LoGIN)" from the University of Nantes (available at : http://login.irin.sciences.univ-nantes.fr/). This LaTeX2e class is under the recommendations of the National Education Ministry of Undergraduate and Graduate Studies (circulaire n° 05-094 du 29 March 2005) of the University of Nantes and the Doctoral School of « Technologies de l'Information et des Matériaux(ED-STIM)

Print : thesis.tex – 05/07/2012 – 10:18.

Last class review: these-LINA.cls,v 2.7 2006/09/12 17:18:53 mancheron Exp

A Common Aspect Languages Interpreter

The value of using different (possibly domain-specific) aspect languages to deal with a variety of crosscutting concerns in the development of complex software systems is well recognized. One should be able to use several of these languages together in a single program. However, on the one hand, developing a new Domain-Specific Aspect Language (DSAL) in order to capture all common programming patterns of the domain takes a lot of time, and on the other hand, the designer of a new language should manage the interactions with the other languages when they are used together.

In this thesis, we introduce support for rapid *prototyping* and *composing* aspect languages based on interpreters. We start from a *base* interpreter of a subset of Java and we analyze and present a solution for its modular extension to support AOP based on a common semantics aspect base defined once and for all. The extension, called the *aspect* interpreter, implements a common aspect mechanism and leaves holes to be defined when developing concrete languages. The power of this approach is that the aspect languages are directly implemented from their operational semantics. This is illustrated by implementing a lightweight version of AspectJ. To apply the same approach and the same architecture to full Java without changing its interpreter (JVM), we reuse AspectJ to perform a first step of static weaving, which we complement by a second step of dynamic weaving, implemented through a thin interpretation layer. This can be seen as an interesting example of reconciling interpreters and compilers. We validate our approach by describing prototypes for AspectJ, EAOP, COOL and a couple of other DSALs and demonstrating the openness of our AspectJ implementation with two extensions, one dealing with dynamic scheduling of aspects and another with alternative pointcut semantics. Different aspect languages implemented with our framework can be easily composed. Moreover, we provide support for customizing this composition.
Keywords Aspect-Oriented Programming (AOP), interpreter, semantics, prototyping, composition, Domain-Specific Aspect Language (DSAL)

Un interpréteur extensible pour le prototypage des langages d'aspects

L'intérêt de l'utilisation de différents langages d'aspects pour faire face à une variété de préoccupations transverses dans le développement de systèmes logiciels complexes est reconnu. Il faudrait être capable d'utiliser plusieurs de ces langages dans un seul logiciel donné. Cependant, d'une part la phase de développement d'un nouveau langage dédié capturant tous les patrons de programmation du domaine prend beaucoup de temps et, d'autre part, le concepteur doit gérer les interactions avec les autres langages quand ils sont utilisés simultanément.

Dans cette thèse, nous introduisons un support pour le prototypage rapide et la composition des langages d'aspects, basé sur des interpréteurs. Nous partons d'un interpréteur d'un sous-ensemble de Java en étudiant et en définissant son extension modulaire afin de supporter la programmation par aspects en se basant sur une sémantique d'aspects partagée. Dans l'interpréteur d'aspects, nous avons implémenté des mécanismes communs aux langages d'aspects en laissant des trous à définir pour implémenter des langages d'aspects concrets. La puissance de cette approche est de permettre d'implémenter directement les langages à partir de leur sémantique. L'approche est validée par l'implémentation d'une version légère d'AspectJ.

Pour appliquer la même approche et la même architecture à Java sans modifier son interpréteur (JVM), nous réutilisons AspectJ pour effectuer une première étape de tissage statique, qui est complétée par une deuxième étape de tissage dynamique, implémentée par une mince couche d'interprétation. C'est un exemple montrant l'intérêt qu'il peut y avoir à concilier interprétation et compilation. Des prototypes pour AspectJ, EAOP, COOL et des langages dédiés simples, valident notre approche. Nous montrons le caractère ouvert de notre implémentation d'AspectJ en décrivant deux extensions: la première permet l'ordonnancement dynamique des aspects, la deuxième propose des sémantiques alternatives pour les points de coupe. Les langages d'aspects implémentés avec notre approche peuvent être facilement composés. En outre, cette composition peut être personnalisée.
Mots-clés Programmation Par Aspects (PPA), interpréteur, sémantique, prototypage, composition, langage d'aspects dédié

Acknowledgements

First of all, I would like to express my deep gratitude to my supervisor, Jacques Noyé. Since i have done my master and during the thesis, i have learnt enormously by interacting with you, on topics as vast as critical mind, self questioning, research attitude, scientific writing and ethics. Thank you for all these years of discussion and advice.

I also profoundly thank Pierre Cointe, the head of laboratory, Mario Südholt, the head of the research team and Narendia Jussien, the head of the Computer Science departement in Ecole des Mines de Nantes (EMN) for providing financial support in various opportunities and events, in particular to participate to conferences.

I also thank each member of the Computer Sciences Departement in EMN and each person in EMN for the friendliness that made this PhD work possible.

I would like to express my gratitude to all the members of the jury for reviewing and commenting on this work.

I want to greet a great person, who wrote the One Thousand and One Nights book in sacrifice and altruism, who had the biggest merit in my scientific and life career, My Mother Shehrazad.

I would like to thank a real jewel, which was my encouragement and my guardian angel for life and work. Who gives my life the color of her eyes, who gives taste to my days, she is my love Diana.

From the bottom of my heart, I want to thank my family, my sisters, my brothers, and friends, for support and care during my years of study.

This thesis was financially supported by the Minister of National Education in France.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Contents

1.1 Separation of Concerns

It has been quite some time since I left Lebanon to come to France where I did a master's degree in Software Engineering. Many things have changed for me and especially food. I thought that there was a big difference between the two cuisines and I found myself poorly adapted to the way of cooking in France.

It is common in Lebanese cuisine, as in all the Mediterranean ones, to mix a variety of fresh vegetarian recipes, salads and stews with a flavorsome combination of herbs and spices then to cook them. This leads to a delicious meal that we eat directly without adding new ingredients. However, this manner has caused problems in our house: a person that did not like one of the ingredient could not eat the prepared meal. The solution was that my mother cooked another meal with an alternative main ingredient. The *Mgadara*, prepared by cooking together lentils, onions and either rice or bulgur, causes problems if we choose to use rice while a person does not like it. We need to cook another time the same ingredients (except the rice) with the bulgur.

In France, I noticed that each ingredient is cooked separately. There is a base ingredient (like meat) and several added ingredients like potato, rice, etc. For example, if French people want to cook *Mgadara*, they cook lentils, bulgur and rice separately and choose what to mix in their plates. I found that the French manner of cooking can help when they are several versions of the same meal using alternative ingredients as we see for the *Mgadara*, when rice, bulgur and lentils are separately cooked.

Now let us leave delicious cuisine and go to work. This comparison has helped me to understand the *Separation of Concerns* [34] proposed by Dijkstra in the early 1970's. Cooking is equivalent to programming and cooking ingredients separately is equivalent to developing several modules, each module deals with one concern. These modules will be composed together in order to elaborate the final product. Separation of concerns is at the core of Software Engineering. When we want to develop a solution of a problem as software, considering this problem as an atomic block makes the elaboration of its solution complex. Separation of concerns in Software Engineering consists of breaking the problem into loosely-coupled subproblems to reduce software complexity and to improve comprehensibility. The subsolutions of these subproblems can then be composed to yield a solution to the original problem. When developing a software system, the Separation of Concerns enables us to focus our attention upon some subprograms from one point of view without caring too much about other irrelevant aspects.

The organization of programs into modules is one of the key issues tackled by programming languages. The research history in the domain of programming languages can be seen as a perpetual quest for ideal modularization. Existing programming paradigms like Object-Oriented Programming (OOP) and Component-Oriented Programming (COP) are useful to modularize most *functional concerns* (belonging to the *base* application) of the system but they fail to provide support for the modularization of a type of concerns, called *crosscutting concerns*, which are aspects of the system which affect (crosscut) other concerns like logging, synchronization or security. Often, these concerns cannot be cleanly decomposed from the rest of the system, and their corresponding code is scattered across the program and tangled with other concerns. For example, logging is typically scattered in different places in the code. Some functional concerns (for instance, billing) may also be hard to properly modularize. Some programming techniques like Meta-Object Protocols [58] and Reflection [30] have tried to solve this problem but they are very complex and do not provide language support for modularization.

Aspect-Oriented Programming (AOP) [47] has been proposed to attack the problem of crosscutting concerns by providing language support to modularize them. It realizes this by letting the user define the behavior of a crosscutting concern and then declaratively describe where this concern has to crosscut the other modules. It is the responsibility of the language infrastructure to elaborate the final program. The next section overviews AOP and motivates the use of several Aspect-Oriented languages to express the different types of crosscutting concerns that *co-crosscut* the system.

1.2 Aspect-Oriented Programming

In Software Engineering, Aspect-Oriented Programming [47] (AOP) attempts to aid programmers in the Separation of Concerns, specifically crosscutting concerns, as an advance in modularization. AOP does so primarily using language changes and extensions. AOP allows programmers to modularize crosscutting concerns as *aspects* while widely spread programming languages and techniques such as OOP and design patterns do not elegantly modularize them. Separating crosscutting concerns using AOP improves the quality of modules as they are loosely-coupled from the other modules and thus can be easily maintained and reused for other purposes. Currently, AOP is mostly realized as an extension of OOP (in particular, several extensions of Java), and implemented by using *program-transformation* technologies applied to a program in order to statically compose aspects into this program. Several Aspect-Oriented Programming Languages (AOPLs) were proposed in order to make it possible to express crosscutting concerns. The *where* concept in the aspect composition technique is captured with the notion of a *join point* [57]. Join points are conceptual points in a software artifact that are of interest to the composition of multiple concerns. Join points are the essential elements through which two or more concerns can be composed. The understanding of the implementation of a join point in different artifacts of the software development life-cycle is important. To determine the set of join points which one is interested in, the majority of AOPLs provide a sublanguage, called the *pointcut* language. Pointcut expressions determine the set of join points needed to be captured by the aspect. A pointcut is associated with a statement that will be executed when the pointcut matches a join point. This statement is commonly called *advice*. This type of AOPL is said to follow the *Pointcut-Advice* model.

There are proposals for both *General-Purpose* and *Domain-Specific* Aspect Languages (GPALs and DSALs, respectively). Domain specificity presents many benefits: declarative representation, simpler analysis and reasoning, domain-level error checking, and optimizations [32]. The DSAL approach follows the language-oriented programming approach [102], which is a style of computer programming via meta-programming in which, rather than solving problems in general-purpose programming languages, the programmer creates first one or more domain-specific programming languages for the problem, and solves the problem in these languages. Several domain-specific aspect languages were indeed linked to the birth of AOP (COOL and RIDL at Xerox Parc [68]), and, after a focus on general-purpose aspect languages (AspectJ in the same group [57]), the interest in DSALs has been revived [28, 100, 92]. A DSAL provides a means to simplify the development of one concern like concurrency, distribution, serialization, etc. But the development of large distributed applications involves the integration of multiple concerns, with multiple stakeholders manipulating the system via multiple different viewpoints. When several aspects are handled in the same piece of software, it is attractive to be able to combine several AO approaches, various DSALs [86, 92] or a GPAL with one or several DSALs, for instance AspectJ and COOL.

1.3 Motivating Problems

The domain of AOSD lacks design spaces of AOPLs (DSALs and GPALs), which could enhance the quality of AOPLs in terms of understandability, reusability and maintainability. Also, the complexity of implementation techniques used for AOPLs makes it difficult to define and test new features and alternative semantics while the diversity of these techniques makes the composition of AOPLs even more difficult. These motivating problems can be decomposed into three parts:

1. The implementation of a DSAL can be tricky. The first step in the process of designing a DSAL is to consider common programming patterns in the area of a concern. The second step is to define the basis of the syntax and the semantics. Note that all the language features are progressively captured by testing the language in real situations. Given this fact, the need to develop a prototype of this language becomes very important to help us test the proposed language features. The resulting prototype must be scalable and maintainable in order to add new features to the language as more common patterns from the domain are captured.
2. Experimenting with various alternative semantics and exploring the design space of AOPLs (DSALs and GPALs) is common in AOSD research. Let us consider the extension of AspectJ pointcut semantics that have undergone changes since the first version (i.e. `execution` pointcut with respect to inheritance [18]). Unfortunately, this is not easy since most existing compilers may not have been designed with extensiblity as one of the main goals (*ajc*: AspectJ Compiler [2]). Even if this the case with *abc*, The AspectBench Compiler for AspectJ [15], working with this tool requires the knowledge of all the machinery of *abc* like Polyglot (extensible front end for Java), Soot (optimization framework for Java programs), Jimple (intermediate representation of Java programs defined within Soot), etc. and it is easier to propose alternative pointcut semantics than to modify complex semantics like *aspect scheduling*.
3. Working with large applications imposes to deal with different concerns, and specially crosscutting concern like synchronization, concurrency, etc. The use of DSALs for each concern (domain) calls for composing the different used DSALs together. However, the diversity of languages implementation techniques makes it almost impossible. To clarify the problem, let us consider two AOPLs, AspectJ and COOL. For each of these two languages, there is a compiler that weaves the corresponding aspects. The compiler takes a base application and makes a specific representation of it then matches the places where crosscutting concerns have to be added and weaves the advice. When making the specific representation, each compiler adds, in the transformed program, some implementation specific code, which can be considered as normal code by the second weaver (assuming weaving takes place in sequence). This leads to unexpected behavior as described in [70].

1.4 Thesis

The overall objective of this thesis is to facilitate the prototyping and the composition of AOPLs.

Approach The general approach is to use interpreters instead of compilers (weavers) because:

1. Interpreters help us to directly implement the semantics of aspect languages when prototyping them and to easily manage the languages interactions when composing them.
2. Interpreters are open and more extensible than compilers.
3. It is easier to compose interpreters than weavers in order to compose AOPLs.

To facilitate the prototyping of AOPLs, our approach consists of defining a common interpreter for AOPLs, which can be extended to build a concrete interpreter for a specific AOPL, whereas the extensions can be composed in order to compose AOPLs. We start from two existing points:

1. The Common Aspect Semantics framework (CASB) [36] as the basis to express the semantics of our interpreter.
2. A metamodel of AOPLs [26] where the common language concepts of aspect languages are represented.

The common interpreter implements the common semantics of AOPLs based on the CASB. As an example of common semantics, let us mention the *interactions* of between the *base interpreter* and the *aspect interpreters*, *matching aspects*, *scheduling aspects*, *executing pieces of advice*, *proceed*. The common interpreter keeps the semantics of a concrete language, like its pointcut semantics, abstract. This semantics is provided by a *plugin* for the language. The common interpreter is designed to be open and flexible so that even the common semantics can be configured. As an example, let us mention the notion of *scheduling aspects*, where it is important to have different types of scheduling strategies, dynamic or static. The prototyping of concrete aspect languages is reduced to the specification of the abstract features in the interpreter. Regarding the composition of aspect languages, independently developed aspect languages (extending the common interpreter) can be easily assembled. The composition of aspect languages is reduced to the composition of aspects because the aspects of all the languages have to extend the abstract aspect notion and similarly interact with the interpreter. The framework provides default configuration of aspect composition and supports the configuration of this composition.

Prototypes The semantics of the CASB assumes that there is a base interpreter which implements the semantics of the base language and the common interpreter must manage their interactions. According to the base interpreter that we consider and following our propositions, we build two prototypes:

1. Extended MetaJ: Dealing with a base interpreter like the Java Virtual Machine (JVM) is very complex. We rather start from a source-level interpreter, MetaJ, a Java interpreter for a subset of Java, and extend it with a common aspect interpreter.
2. CALI: To apply the same architecture to Java without making changes in the JVM, we use AspectJ to implement join points and forward them to a thin interpretation layer responsible for aspect-specific management. This can be seen as an interesting example of reconciling interpreters and compilers, the dynamic and the static world. The resulting framework is called CALI, for *Common Aspect Language Interpreter*.

Validation As a validation of our work, we present a prototype of AspectJ implemented using CALI. The extensibility of this AspectJ implementation is validated by two AspectJ variants, one dealing with *dynamic aspect scheduling*, another with *alternative selector semantics*. CALI is also used to prototype very different AOPLs like Event-Based AOP (EAOP) [37] and the COOrdination Language (COOL) [68].

The CALI-based implementations can be easily composed because they are all based on the same abstract language. Moreover, CALI supports the configuration of AOPL composition and the resolution of aspect interactions at different levels (language and aspect). Being implemented with CALI, the composition of AspectJ and COOL prototypes is directly supported.

Contributions Here are some of our achievements during our work on this thesis. We have:

– Presented the shortcomings of the translation approach for prototyping AOPLs.
– Used the CASB as an intermediate level of abstraction between the conceptual model and the implementation level of AOPLs.
– Extended the CASB in order to introduce a notion of aspect group, which is used for dynamic scheduling of aspects.
– Designed and implemented an aspect interpreter completely separated from the base one.
– Extended AspectJ in order to introduce dynamic scheduling by using the variable `thisAspectGroup`.
– Implemented a flexible prototype of AspectJ and presented how it is simple to extend it in order to implement two variants of AspectJ: one with dynamic scheduling, another with alternative semantics for pointcuts.
– Presented the configuration of AOPLs composition at aspect level while other frameworks provide a configuration at language level.

1.5 Structure of the Dissertation

After this introduction, we present the thesis as three parts: The first (from Chapter 2 to Chapter 4) is about the state of the art, the second (from Chapter 5 to Chapter 10) is about our contributions and the third is the conclusion.

State of the Art Chapter 2 presents the main features of AOPLs and their different implementation techniques. Chapter 3 overviews the existing approaches for prototyping and composing AOPLs and present the shortcomings of such approaches. The analysis of the existing approaches lead us to explicit the requirements necessary to fulfill our goals in Chapter 4.

Contributions According to the requirements discussed in the previous chapter, Chapter 5 presents the modular extension of a Java interpreter (MetaJ) and infers a general architecture linking the aspect and base interpreters. A lightweight AspectJ version is implemented by extending the aspect interpreter.

Chapter 6 applies the architecture of Chapter 5 to Java without modifying the JVM. The aspect interpreter is built to be extensible and the resulting framework is called CALI for *Common Aspect Language Interpreter*. In Chapter 7 and Chapter 8, we describe the implementation of AspectJ and two variants of its semantics with this approach. The first variant introduces dynamic aspect scheduling and the second introduces an alternative semantics of the pointcut expressions associated to method calls and executions.

Chapter 9 shows the implementation of a number of AOPLs on top of CALI.

Chapter 10 shows how AOPLs implemented using CALI can be composed.

Conclusion Chapter 11 evaluates the interpreter performance. We discuss some related work in Chapter 12. We conclude in Chapter 13 by summarizing our contributions and discussing limitations of this thesis before giving some perspectives in Chapter 14.

Part I

The State of the Art

Table of Contents

Chapter 2

Aspect-Oriented Languages

Contents

In this chapter, we give background information on programming languages and their implementation techniques that will be useful in the following chapters. We show why in some cases it is more useful to construct an interpreter than to construct a compiler. Additionaly, we discuss the relation between *operational semantics* and interpeters by describing MetaJ, an interpreter for a subset of Java written in Java.

After that, we turn to discuss the features of AOPLs as programming languages. We describe three representative approaches: AspectJ (standard approach), EAOP (sequence-based join-point matching) and COOL (domain-specific AOPL). We present the existing *aspect-weaving mechanisms* used in the implementations of AOPLs and we show how the semantics of an aspect extension can be formally described by extending the operational semantics of the base language. This means that an AOPL implementation could be realized by extending the interpreter of the base language. This idea is going to be the starting point of our contribution.

2.1 Definition and Implementation of Programming Languages

In this section, we very quickly present the elements that define a programming language and discuss interpretation vs compilation.

2.1.1 Definition of Programming Languages

A program written in a given programming language is a composition of *syntactic constructs* whose execution perform the desired computations. Designing a programming language consists of defining the *syntax* and the *semantics* of the language. The syntax defines the form of a valid program whereas the semantics defines the meaning of the program, that is, its computations.

2.1.1.1 Syntax

Most programming languages are purely textual; they use sequences of *tokens* including words, numbers, and punctuation, much like written natural languages. These sequences can be composed into *syntactic constructs* defined by the grammar of this language, typically a context-free grammar described in Backus-Naur Form (BNF) [9].

2.1.1.2 Semantics

The semantics of a language describes the computations performed by any program written in the given language. Ideally, this semantics is represented by (see for instance [81]) a mathematical model called *formal semantics* of the language. The model can take different forms:

Denotational semantics associates each syntactic construct with a function describing the *effect* of executing this construct.

Operational semantics associates each syntactic construct with a function that describes *how* the state of an underlying abstract machine is modified when executing this construct. In practice, the operational semantics of a language can also be described by an interpreter (see below).

In the rest of this thesis, we will be interested in operational semantics.

As we said, the operational semantics of a language describes *how* a program written in this language is executed.

Here we describe an example, found in [36], of the basic elements of an operational semantics:

A program C is a sequence (there are no compound instructions) of basic instructions i terminated by the empty instruction ϵ:

$$C ::= i : C | \epsilon$$

$i : C$ denotes the prefixing of a program C with an instruction i.

The state of the program is represented by Σ. Σ contains an environment, a stack, a heap, etc., depending on the precise semantics of the considered language. An execution of an instructions is described by a change in this state, which is captured by reduction rules of the form:

$$(i : C, \Sigma) \rightarrow (C', \Sigma')$$

This means that the execution of the instruction i in the state Σ takes the program state to Σ', and proceeds with the execution of the program C'.

2.1.2 Implementation of Programming Languages: Interpretation vs Compilation

Once we have defined (when using an interpreter to define the operational semantics of a language, the definition of the semantics also provides a first implementation of the language) the syntax and the semantics of a programming language, we can start to implement it. We have to choose between an *interpretation* or a *compilation* of our language. In the following, an `X`-program is a program written in the programming language `X`. For the sake of simplicity, we suppose that the programs are without parameters.

Interpreter An interpreter $int_I(S)$ is an I-program that executes S-programs. I is the *defining* or *implementation language*, while S is the *defined* or *source language*. The result of the execution of `int` with an S-program p_S as input, is: $output = int_I(p_S)$.

Compiler A compiler `comp` is an I-program that takes an S-program p_S as input and returns a T-program p_T: $p_T = comp_I(p_S)$. As above, S is the *defined* or *source language* and T is the *target language*.

2.1.2.1 Why using interpreters

When developing new programming language, there are several reasons to make an interpreter for it (see, for instance [17]):

```java
public class Main {
  public static Environment globalE;
  public static void main(String[] args) {
    // global environment for user classes
    Main.globalE = new Environment(null, null, null);
    // user input file parsing
    Exp prog = Parser.File2Exp(args[0]);
    // main entry point: Main.main()
    prog = new ExpS(prog, new ExpMethodClass("Main", "main", new
       ExpList(null, null)));
    prog.eval(null);
  }
}
```

Listing 2.1: Launching the evaluation of a program in MetaJ

1. An interpreter is normally written in a high-level language and will therefore run on most machine types, whereas generated object code will only run on machines of the target type: in other words, portability is increased.
2. As we said before, an interpreter written in a sufficiently abstract, concise, and high-level language, can serve as a language definition: *an operational semantics* for the interpreted language (defined language) and it directly implements the operational semantics.
3. Writing an interpreter back-end is much less work than writing a compiler back-end. One reason is that the implementer thinks only of one time (the execution time), whereas a compiler must perform actions to generate code to achieve a desired effect at runtime. This is an advantage when rapidly prototyping, testing and extending the defined language.
4. Performing the actions straight from the semantic representation allows better error checking and reporting to be done. This is not fundamentally so, but is a consequence of the fact that compilers (front-end/back-end combinations) are expected to generate efficient code. As a result, most back-ends throw away any information that is not essential to the program execution in order to gain speed; this includes much information that could have been useful in giving good diagnostics, for example source code line numbers.
5. Increased security can be achieved by interpreters; this effect has played an important role in *Java's rise to fame*. Again, this increased security is not fundamental since there is no reason why compiled code could not do the same checks an interpreter can. Yet it is considerably easier to convince oneself that an interpreter does not play *dirty tricks* than that there are no *booby traps* hidden in binary executable code.

2.1.2.2 MetaJ: A Java interpreter

As an example of interpreter, we choose MetaJ [39, 40], a Java interpreter (for a subset of Java comprising all essential object-oriented and imperative features, such as classes, objects, fields, methods, local variables, assignment statements, etc.) written in Java. When we say Java interpreter, this means that it takes a Java program and performs the corresponding computations according to the *Java language semantics* [50].

The purpose of introducing MetaJ is that we will use it as our base interpreter in Chapter 5. We will instrument MetaJ in order to generate *join points* and to communicate them to an aspect interpreter. The challenge will be how to instrument the base interpreter in a modular way and with as few changes as possible.

The architecture of MetaJ is presented in Figure 2.1. Listing 2.1 shows the main class, which takes as parameter an `.mjava` file (see Listing 2.5) and evaluates it. The grammar of the subset of Java is defined as an LL(k) grammar in `parser.jjt`. JavaCC [6], which is a parser generator,

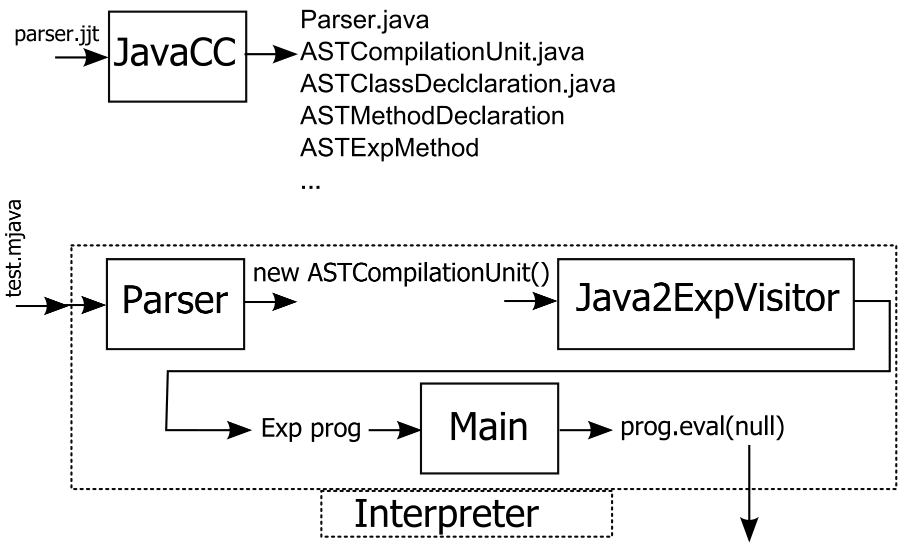

Figure 2.1: The MetaJ architecture

```
public abstract class Exp {
  abstract Data eval(Environment localE);
}
```

Listing 2.2: The `ExpId` class

takes the grammar definition and generates a `Parser.java` file and a list of Java files representing the concrete syntax nodes defined by the grammar. The parser takes a MetaJ program written in an `.mjava` file and creates a concrete syntax tree for it. The root of this tree is visited by `Java2ExpVisitor` to an abstract syntax tree of the program that is ready to be evaluated. The result of visiting the tree is an instance of a concrete subclass of the class `Exp` (see Listing 2.2). The interpretation of the program is launched by invoking the method `eval` on the instance. The structure of MetaJ is object-oriented with the modular property that a new class may be easily added without modifications to the existing code. For example, a variable and an assignment are encoded by the classes `ExpId` (Listing 2.3) and `ExpAssign` (Listing 2.4), respectively.

Adding a new operation, however, involves modifying every single data class to implement the data specific behavior of the new operation. Each of these classes provides a method `Data eval(Environment localE)` that takes the value of local variables in `localE` and returns the

```
public class ExpId extends Exp {
  private String id;
  ExpId(String id) { this.id = id; }
  Data eval(Environment localE) {
    return localE.lookup(this.id);
  }
}
```

Listing 2.3: The `ExpId` class

```java
public class ExpAssign extends Exp {
  private Exp lhs;
  private Exp rhs;
  ExpAssign(Exp lhs, Exp rhs) { this.lhs = lhs; this.rhs = rhs; }
  public Data eval(Environment localE) {
    // eval right-hand side first
    Data d1 = this.rhs.eval(localE);
    // eval left-hand side
    Data d2 = this.lhs.eval(localE);
    // assign right-hand side to left hand side
    d2.write(d1.read());
    // return left-hand side
    return d2;
  }
}
```

Listing 2.4: The `ExpAssign` class

value of the expression. For example, the class `ExpId` (Listing 2.3) holds the name of a variable and the evaluation method.

2.2 Aspect-Oriented Programming Languages

2.2.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [57] is a technology that aims to improve the separation of crosscutting concerns, concerns that would otherwise be *tangled* and *scattered* across other concerns. This separation was very difficult with past technologies like OOP and COP (using Reflection and Meta-Object Protocols [58]).

2.2.1.1 Crosscutting concerns

As we have mentioned in Chapter 1, separation of concerns consists of a *divide and conquer* strategy for problem solving by breaking a complex problem into subproblems instead of tackling it at once, solving each subproblem independently then composing the solutions to get a solution to the overall problem.

As developing software is a complex task, such an approach is useful at many levels. At the first levels of the development life cycle, it is usual to reason in terms of *concerns* that have to be addressed by the design of the application. To clarify the notion of concerns, let us consider an application of bank management. We call concerns terms such as *deposit* or *withdrawal*, as well as *logging*, *transaction handling* and *security management*.

The design of the concerns deposit and withdrawal is described by the class diagram in Figure 2.2, and the skeleton of its implementation is given in Listing 2.6.

Assembling the full application consists simply of putting the different concerns together. Well, this is not so simple actually because the different parts must interact, which means that there should be proper interfaces.

Let us consider the logging concern. Using the Logging Java API (`java.util.logging`) [7], we can implement (see Listing 2.7) the logging concern in our application by creating a logger with an associated file using the logging API. Then logging can take place by calling the entering and exiting methods of the class `Logger` on the logger.

The logging concern interacts with almost all the other concerns because one would have the log of all the operations executed inside the application. We can see this interaction in the code

```
class Pair {
  public String fst;
  public String snd;
  Pair(String fst, String snd) { this.fst = fst; this.snd = snd; }
  public void swap() {
    String tmp = this.fst;
    this.fst = this.snd;
    this.snd = tmp;
  }
}
class Main {
  void main() {
    Pair pair = new Pair("1", "2");
    pair.swap();
    pair.fst = "3";
    pair.swap();
  }
}
```

Listing 2.5: An example of a program in MetaJ

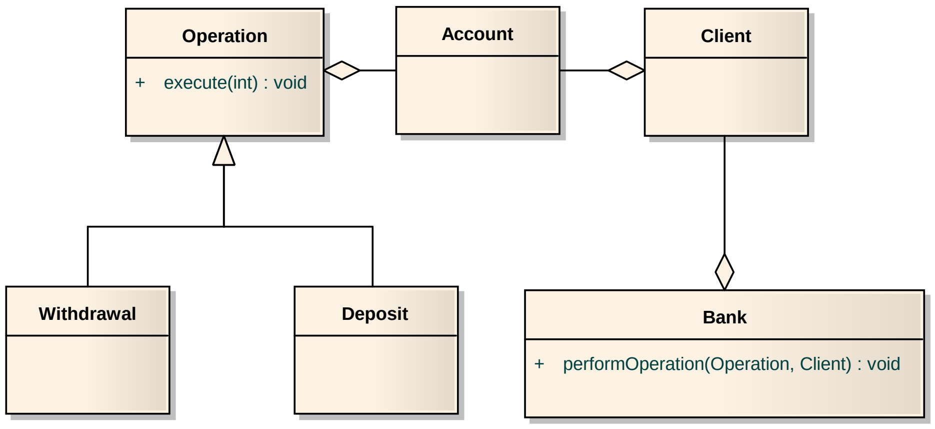

Figure 2.2: Class diagram of the bank application

```java
package BankManagement;
public class Bank {
  public Client client;
  public Bank(){ .. }
  public void performOperation(Client client, Operation op){ .. }
}

public class Operation {
  public Account account;
  public Operation(){ .. }
  public void execute(int amount){ .. }
}
```

Listing 2.6: The implementation of the bank application

```java
package BankManagement;
public class Bank {
  protected static Logger logger =
                     Logger.getLogger("BankManagement.Bank");
  public Client client;

  public Bank(){
    Handler fh = new FileHandler("myLog.log");
    logger.addHandler(fh);
  }

  public void performOperation(Client client, Operation op){
    Object[] objs = new Object[2];
    objs[0]=client;
    objs[1]=op;
    logger.entering(this.getClass().getName(),"performOperation",
       objs);
    // ..
    logger.exiting(this.getClass().getName(),"performOperation",
       objs);
  }
}

public class Operation {
  protected static Logger logger =
                     Logger.getLogger("BankManagement.Operation");
  public Account account;
  public void execute(int amount){
    logger.entering(this.getClass().getName(),"execute",amount);
    // ..
    logger.exiting(this.getClass().getName(),"execute",amount);
  }
}
```

Listing 2.7: Logging the bank application

```
aspect Logging{
  pointcut log(): call(* Bank.performOperation(..)) ||
                  call(* Operation.execute(..));
  around(): log(){
    logger.entering(thisJoinPoint.getTarget().getClass().getName(),
                    thisJoinPoint.getSignature().getName(),
                    thisJoinPoint.getArgs()[0]);
    proceed();
    logger.exiting(thisJoinPoint.getTarget().getClass().getName(),
                   thisJoinPoint.getSignature().getName(),
                   thisJoinPoint.getArgs()[0]);
    out.close();
  }
}
```

Listing 2.8: The AOP implementation of the logging concern using AspectJ

where the logging code is tangled across different classes as we see in the `Bank` and `Operation` classes.

A *crosscutting concern* is a concern that affects several classes or modules, a concern that is not well localized and modularized.

Symptoms of a crosscutting concern are:

– *Code tangling* when a module or code section is managing several concerns simultaneously.
– *Code scattering* when a concern is spread over many modules and is not well localized and modularized.

These symptoms affect software in various ways; they make it harder to maintain and reuse software.

2.2.1.2 Separation of concerns with AOP

Aspect-Oriented Programming solves the problem of crosscutting concerns by adding an extra dimension to the design space of software; concerns that were spread over many modules are written in independent modules called *aspects*, which are automatically associated (*woven*) according to rules declaratively defined by the user to precise *where* they crosscuts other concerns. The points where two concerns crosscut are called *join points*. *Structural* join points are locations in the program text whereas *behavioral* join points are points in the program execution control flow, such as invocations of a method, accesses to a field, etc. AspectJ is the most popular AOP language [57, 94, 59, 66]. It extends Java with aspects, which contain the definition of *pointcuts* and *advices* in addition to standard Java members like fields and methods. A *pointcut* selects the points of the so-called *base program* where extra code (the *advice*) has to be executed.

Listing 2.8 uses AspectJ as an AOPL to implement the bank application, logging is encapsulated in a `Logging` aspect.

In Listing 2.8, we say that we want to intercept the calls to the two methods `performOperation` of class `Bank` and `execute` of class `Operation` in a pointcut named `log`. Then, we declare in the `around` clause what to execute when the pointcut `log` selects a join point. AspectJ offers a reflective way to access the information that we need here to write in the log file. For example, we can call `thisJoinPoint.getTarget()` to get the target of the message. The expression `proceed()` makes it possible to execute the selected join point, here a call.

The difference between the use of the logging API and AspectJ is that in the first approach, the code of logging is tangled across the two classes. If we want to change the log policy, we must change code in different places while in the second solution using AspectJ, the logging code is well modularized in the `Logging` aspect, which is better for the understandability and the maintenance of the application.

2.2.1.3 Structural and Behavioral AOP

Structural AOP is the weaving of modifications into the *static structure* (classes, interfaces, etc.) of the program. For crosscutting concerns that do operate over the static structure of type hierarchies, AspectJ provides *inter-type declarations* [94]. AspectJ aspects can declare inter-type members (fields, methods, and constructors) to be owned by other types. They can also declare that other types implement new interfaces or extend a new class. Consider the problem of expressing a capability shared by some existing classes that are already part of a class hierarchy, i.e. they already extend a class. In Java, one creates an interface that captures this new capability, and then adds to each affected class a method that implements this interface.

AspectJ can express the concern in one place, by using inter-type declarations. The aspect declares the methods and fields that are necessary to implement the new capability, and associates the methods and fields to the existing classes.

```
public Data Exp.evalWithAspect(Environment env,
                               Environment aspenv){
  return eval(env);
}
```

Listing 2.9: Adding an `evalWithAspect` to `Exp`

An example of inter-type declaration of AspectJ is shown in Listing 2.9 where we add a new method having the signature `Data evalWithAspect(Environment env, Environment aspenv)`, to the class `Exp`.

Behavioral AOP is the weaving of new behavior into the execution of a program. It augments or even replaces the program execution flow in a way that crosses modules boundaries, thus modifying the system behavior. AspectJ provides the notion of *pointcut* to pick out certain join points in the program flow. For example, the pointcut:

```
call (* Bank.performOperation(..))
```

picks out each join point that is a call to any method called `performOperation` in the class `Bank`.

2.2.1.4 General-Purpose and Domain-Specific AOPLs

A current trend in AOP is to take a programming language like Java or C++, then integrate a certain aspect extension providing support for expressing crosscutting concerns. This extension could be general purpose or domain specific hence the distinction between a *general-purpose aspect language* (GPAL) and a *domain-specific aspect language* (DSAL).

GPALs are designed to be used for every kind of crosscutting concerns. They usually have the same level of abstraction as the base language and also the same set of constructs, as it must be possible to express arbitrary code in the pieces of advice. AspectJ is an example of a GPAL.

A DSAL is a custom language that allows special forms of crosscutting concerns to be decomposed into different modules using constructs that provide an appropriate expressivity and abstraction with respect to the domain. Seminal examples of DSALs include languages for dealing with coordination concerns like COOL, for controlling thread synchronization over the execution of the components; and RIDL, for programming interactions between remote components and class graph traversal concerns [68].

In the following, we are going to describe three typical languages: two GPALs, AspectJ and EAOP, and a DSAL, COOL.

2.2.2 An Overview of AspectJ

AspectJ is an AOPL that extends the Java programming language. This means that all valid Java programs are also valid AspectJ programs. AspectJ allows programmers to separate crosscutting concerns in a unit called *aspect*. In addition to normal members such as instance variables and methods, an aspect additionally encapsulates three other kinds of members:

Inter-type declarations allow a programmer to add methods, fields, or interfaces to existing classes from within the aspect. This example adds an `acceptVisitor` method (see visitor pattern in [49]) to the `Point` class:

```
aspect VisitAspect {
  void Point.acceptVisitor(Visitor v) {
    v.visit(this);
  }
}
```

Pointcuts allow an *aspect programmer* to specify join-point selection. At this stage, it is important to say that *base programmer* is *oblivious* of aspect system [45]. All pointcuts are expressions that determine whether a given join point matches: these expressions are able to *quantify* over join points. For example, this pointcut matches the execution of any instance method in an object of type `Point` whose name begins with `set`:

```
pointcut set() : execution(* set*(..) ) && this(Point);
```

For reuse purposes, pointcuts can be named and parametrized. Pointcuts are built using logical connectors, in-scope named pointcuts, and built-in pointcut *descriptors*, in the example `execution`, which refers to the callee-side entry point of a method, and `this`, which refers to the object emitting the join point.

Advice allows a programmer to specify an *action* to run at a join point selected by a pointcut. An advice binds a pointcut and its associated action, or *advice body*: a Java block of statements. The actions can be performed *before*, *after*, or *around* the specified join point. In the body of an around advice, the special instruction `proceed` specifies where the execution of the join point should take place. If there is no `proceed`, the execution of the join point is replaced by the execution of the advice body. A *before* (*after*) action can be implemented by an around advice with a construct `proceed` inserted at the end (beginning) of the advice.

In the following, we briefly review the principles of join point matching as well as the handling of aspects (extension, instantiation, and composition) in AspectJ, based on the AspectJ Programming Guide [94] (Appendix B, Language semantics). For the sake of simplicity, we do not deal with exceptions and control flows (see [94] for details), which are not central to our work.

2.2.2.1 AspectJ pointcuts

Method-related pointcuts AspectJ provides two primitive pointcut designators designed to capture method-call and execution join points.
– `call(`*MethodPattern*`)`
– `execution(`*MethodPattern*`)`
where *MethodPattern* is a pattern on method signatures mainly based on types and string patterns on method names. The pointcut `call` refers to the caller side and `execution` to the callee side. The way types are handled is further discussed in Section 7.3.

Field-related pointcuts AspectJ provides two primitive pointcut designators designed to capture field reference and assignment join points:
– `get(`*FieldPattern*`)`
– `set(`*FieldPattern*`)`
where *FieldPattern* is a pattern on field signatures mainly based on types and string patterns on field names.

Object creation-related pointcuts AspectJ provides primitive pointcut designators to select various join points related to object instantiation:
– `call(`*ConstructorPattern*`)`
– `execution(`*ConstructorPattern*`)`

– `initialization(`*ConstructorPattern*`)`
– `preinitialization(`*ConstructorPattern*`)`
where *ConstructorPattern* is similar to a method pattern with the keyword `new` used instead of a method name.

Advice execution-related pointcuts AspectJ provides one primitive pointcut designator to capture execution of advice:
– `adviceexecution()`

State-based pointcuts These designators make it possible to select join points based on information captured by the join point about the context of the execution or expose part of this information (for further selection or use in advice). The descriptor `this` refers to the object within which the join point is executed, the descriptor `target` to the *target* of the joint point and the descriptor `args` to its *arguments*. The exact meaning of `target` (mainly useful with respect to method calls) and `args` depends on the join-point kind. When used for selection, the descriptors take a dynamic type as a parameter, otherwise they take a formal parameter, which is bound to the value taken by the referenced information at runtime.
– `this(`*Type* | *Id*`)`
– `target(`*Type* | *Id*`)`
– `args(`*Type* | *Id* or `"..", ...)`

Program text-based pointcuts While many concerns cut across the runtime structure of the program, some must deal with its lexical structure. AspectJ allows aspects to select join points based on where their associated code is defined (within classes or interfaces, methods, or constructors):
– `within(`*TypePattern*`)`
– `withincode(`*MethodPattern*`)`
– `withincode(`*ConstructorPattern*`)`

Expression-based pointcuts The pointcut `if` select join points based on a dynamic property expressed as a boolean expression:
– `if(`*BooleanExpression*`)`

2.2.2.2 Signature of a join point

In AspectJ, a very important property of a join point is its *signature*, which is used by many of AspectJ's pointcut designators to select particular join points. The selection is based on comparing what is called the *qualifying type* of the join point and the *declaring type* given in the pointcut. The general rule is that the qualifying type should be a subtype of the declaring type. The following defines the *qualifying type* depending on the type of the join point. Note that AspectJ computes the qualifying type statically (this is linked to a global strategy which aims at reducing the number of join points emitted at runtime). The notions of *declaring type* and *qualifying type* are the key of our work on alternative semantics of AspectJ pointcuts in Chapter 8. Here we are going to describe the signature of different types of AspectJ join points associated with methods, fields, etc.

Methods Method-call and method-execution join points typically have method signatures consisting of a method name, parameter types, return type, receiver type. At a method call join point, the signature is a method signature whose *qualifying type* is the static type used to access the method. This means that the signature for the join point created from the call `((Integer)i).toString()` is different than that for the call `((Object)i).toString()`, even if `i` is the same variable. In the former case, the qualifying type is `Integer` whereas in the latter case, it is `Object`. At a method execution join point, the signature is a method signature whose *qualifying type* is the declaring type of the method.

Fields Field-set join points typically have field signatures consisting of a field name and a field type. A field reference join point has such a signature, and no parameters. A field set join point has such a signature, but has a single parameter whose type is the same as the field type.

Constructors Constructor-call join points typically have constructor signatures consisting of a parameter type, receiver type, and the declaring type. At a constructor call join point, the signature is the constructor signature of the called constructor.

At a constructor execution join point, the signature is the constructor signature of the currently executing constructor.

At object initialization and pre-initialization join points, the signature is the constructor signature for the constructor that started this initialization: the first constructor entered during this type's initialization of this object.

Others At a handler execution join point, the signature is composed of the exception type that the handler handles.

At an advice execution join point, the signature is composed of the aspect type, the parameter types of the advice, the return type (void for all but around advice) and the types of the declared (checked) exceptions.

2.2.2.3 The reference `thisJoinPoint`

In the AspectJ Programming Guide [94], the special reference variable `thisJoinPoint` is explained as the container of the reflective information about the current join point for the advice to use. The `thisJoinPoint` variable can only be used in the context of advice, just like `this` can only be used in the context of non-static methods and variable initializers. This variable gives information about the join point kind, signature, etc. For example, `thisJoinPoint.getKind()` returns a string representing the kind of the current join point.

2.2.2.4 Aspect Extension

In AspectJ, aspects can be extended in the same way as classes with some restrictions:

Classes may not extend aspects

Aspects extending aspects Aspects may extend other aspects, in which case not only are fields and methods inherited but so are pointcuts. However, aspects may only extend abstract aspects. It is an error for a concrete aspect to extend another concrete aspect.

Aspects may extend classes and implement interfaces An aspect, abstract or concrete, may extend a class and may implement a set of interfaces. Extending a class does not provide the ability to instantiate the aspect with `new`.

2.2.2.5 Aspect Instantiation

Aspects are not explicitly instantiated as classes but they are automatically created at runtime. All methods and advice within an aspect run in the context of one aspect instance. A program can get a reference to an aspect instance using the static method `aspectOf`. There are several policies for aspects instantiation in AspectJ:

Singleton An aspect has exactly one instance that potentially cuts across the entire program.

Per-object – `aspect` *Id* `perthis(`*Pointcut*`)` ...
– `aspect` *Id* `pertarget(`*Pointcut*`)` ...

The `perthis(`*Pointcut*`)` instantiation policy creates an aspect instance for each unique object bound to `this` at join points matched by *Pointcut*. In other word all join points having the same executing object and selected by the *Pointcut* will be run in the same aspect instance.

Similarly, the `pertarget(`*Pointcut*`)` instantiation policy creates an aspect instance for every object that is the target object of the join points selected by *Pointcut*.

2.2.2.6 Aspects Composition

Aspect ordering AspectJ uses precedence decralations to determine the order of advice execution at a *shared* join point. Note that a piece of around advice controls whether advice of lower precedence will run by calling `proceed`. Also, a piece of before advice can prevent advice of lower precedence from running by throwing an exception.

Aspect of aspects In AspectJ, an aspect can select join points that occur within the control flow of any piece of advice in the same way it selects base join point.

2.2.3 An Overview of Event-Based AOP

The Event-Based AOP (EAOP) approach, introduced in [38, 37, 41], is based on the observation of execution *events* (another name for join points) and the insertion of instructions according to execution states.

Douence *et al.* [38] present a formal model for EAOP. The primitive constituents of the aspect language are basic rules $C \triangleright I$ where C is a *crosscut* and I an *insert* (not essentially different from pointcut and advice). Crosscuts are patterns matching join points whereas inserts are templates. The intuition behind a basic rule is that when the crosscut matches the current join point, it yields a substitution which is applied to the insert before executing it. As in AspectJ, a proceed instruction can be used in a crosscut to execute the join point.

Aspects match sequences of join points and they evolve according to the join points they match. Aspects are defined using the following grammar:

$$
\begin{aligned}
A ::= & \ \mu a.A \\
| & \ C \triangleright I; A \\
| & \ C \triangleright I; a \\
| & \ A \Box A
\end{aligned}
$$

An aspect is either:
- A recursive definition of an aspect denoted by the variable a which may be reused in A.
- A sequence formed using the prefix operation $C \triangleright I; X$, where X is an aspect or a variable. The matching of a crosscut C against a join point j produces a substitution ϕ which associates values to the variables of C: $C\ j = \phi$. The variables are replaced by their value and X becomes the aspect to be woven. Otherwise, we say that the crosscut does not match the join point and we write $C\ j = fail$.
- A choice construct $A \Box A$, which chooses the first aspect that matches a join point (the other is thrown away). If both match the same join point, the first is chosen.

An alternative way to represent the grammar of EAOP is to use the syntax of Finite State Processes syntax (FSP). FSP is explained in [72]. It is a process-calculus with a CSP-like syntax but a CCS semantics. FSP specifications generate finite Labelled Transition Systems. Processes are defined using action prefix, choice and recursion. Figure 2.3 shows a simplified grammar of FSP where all action prefixes and "real" choices are parenthesized.

In order to model EAOP, action prefixes are simply extended as shown in figure 2.4.

2.2.3.1 Example

The example [82] consists of an e-commerce application, where clients can *login* to identify themselves, then they can *browse* an on-line catalog or *logout* to end the session. In addition, the application administrator can *update* the application by publishing a working copy. The example is implemented by a class `Server` (Listing 2.10) having the methods `login`, `browse` and `logout`.

```java
public class Server {
  private State outOfSession = new OutOfSession();
  private State inSession = new InSession();
  private State state = outOfSession;

  public void login(){ state.login(); }
  public void logout(){ state.logout(); }
  public void update(){ state.update(); }
  public void browse(){ state.browse(); }

  private abstract class State {
    abstract void login();
    abstract void logout();
    abstract void update();
    abstract void browse();
  }
  private class OutOfSession extends State{
    void login(){
      state = inSession;
        System.out.println("server in session");
    }
    void logout(){
      System.out.println("logout ignored");
    }
    void update(){
      System.out.println("updating");
    }
    void browse(){
      System.out.println("browsing ignored");
    }
  }
  private class InSession extends State{
    void login(){
      System.out.println("login ignored");
    }
    void logout(){
      state = outOfSession;
      System.out.println("server out of session");
    }
    void update(){
      System.out.println("updating");
    }
    void browse(){
      System.out.println("browsing");
    }
  }
}
```

Listing 2.10: The class Server

Program	::= *ProcessDefinition*
Program	::= *ProcessDefinition Program*
ProcessDefinition	::= *SubProcessDefinition* `.`
ProcessDefinition	::= *SubProcessDefinition* `,` *ProcessDefinition*
SubProcessDefinition	::= *ProcessId* `=` *Body*
Body	::= *ProcessId*
Body	::= *Choice*
Choice	::= *ActionPrefix*
Choice	::= `(`*ActionPrefix* `|` *Choice*`)`
ActionPrefix	::= `(`*ActionId* `->` *Body*`)`

Figure 2.3: Simplified version of FSP with nested choices and action prefixes

ActionPrefix	::= `(`*Action* `->` *Body*`)`
Action	::= *ActionId*
Action	::= *ActionId* `>` *Advice*
Advice	::= *ActionId*
Advice	::= *ActionId* `,` *Advice*

Figure 2.4: Extending FSP grammar in order to support EAOP

The example uses the State Design Pattern [49]. The State Pattern is implemented by the abstract class `State` and two concrete subclasses `OutOfSession` and `InSession`.

Updating this application during a session may cause erroneous pricing results to the client. For this reason, an aspect could be added to solve this problem. This aspect is called `Consistency` and its role is to cancel updates during sessions.

Implementation with AspectJ The consistency concern can be realized using the AspectJ aspect `Consistency` 2.11. The crosscutting code for state transitions is modularized in the aspect using `after` advices. The idea behind using `after` advice is to ensure that the state transition is done after the join point execution.

Implementation with EAOP Instead of programmatically managing the state machine, EAOP provides language support to manage the aspect state evolution. A prototype for EAOP for Java is available at [5]. For instance, the example can be written in EAOP as:

$$\mu a.(login \triangleright proceed; \mu b.(update \triangleright skip; log; b) \Box logout \triangleright proceed; a)$$

Using the Extended FSP grammar, the example is rewritten as:

```
Server  =
  ( login > proceed  -> Session ),
Session =
  ( update > skip -> Session
  | logout > proceed -> Server ).
```

```
public aspect Consistency {
  static boolean inSession = false;
  pointcut login():
    call(void Server.login()) && if(inSession == false);
  pointcut logout():
    call(void Server.logout()) && if(inSession == true);
  pointcut update():
    call(void Server.update()) && if(inSession == true);

  after(): login() {
    inSession = true;
  }
  after(): logout() {
    inSession = false;
  }
  void around(): update(){}
}
```

Listing 2.11: The AspectJ implementation of the EAOP aspect

2.2.4 An Overview of COOL

COOL is a DSAL defined as part of the language framework D [68]. COOL provides means for dealing with mutual exclusion of threads, synchronization state, guarded suspension and notification. An aspect written in COOL is called a `coordinator`. We further describe COOL features by giving a standard example of using a coordinator.

2.2.4.1 Example

Consider the unbounded stack of Listing 2.12. The stack methods are not synchronized. This means that these methods can be called simultaneously. Two threads calling `push` may lead to a wrong stack index. This also means that they may execute in an invalid context. Calling `pop` when the stack is empty raises an exception.

COOL relieves the implementor of `Stack` from dealing with multi-threading. A separate coordinator `Stack` (see Listing 2.13) imposes the synchronization logic over the methods `push` and `pop` of the class `Stack` in an aspect-oriented manner. The use of the name `Stack` implies that it applies to the class `Stack` that could be called the *coordinator target*. In the presence of the coordinator `Stack`, the stack object operates correctly even when multiple client threads execute methods simultaneously. The synchronization policy is expressed using declarations (`selfex` for self exclusion, `mutex` for mutual exclusion, `condition`), expressions (requires), and statements (`on_exit`, `on_entry`). The `selfex` declaration specifies that if a thread is executing either `push`, any other threads are prohibited to execute `push`.

The `mutex` declaration prevents the concurrent execution of `push` and `pop`. In other words, only one method in a `mutex` declaration may be executed at any given time. The `requires` expressions further guard `push` and `pop` executions. If the guard is false, a thread suspends, even if the `mutex` and `selfex` conditions are satisfied. The execution resumes when the guard becomes true. `full` and `empty` are condition boolean variables. The `on_entry` and `on_exit` blocks update the aspect state immediately before and immediately after the execution of an advised method body, respectively. In Listing 2.13, their role is to track the number of elements in the stack and to update the condition variables `full` and `empty`.

```java
public class Stack {
  public Object[] buf;
  public int ind;
  public Stack(int capacity) {
    buf = new Object[capacity];
  }
  public void push(Object obj) {
    buf[ind] = obj;
    ind++;
  }
  public Object pop() {
    Object top = buf[ind - 1];
    buf[--ind] = null;
    return top;
  }
}
```

Listing 2.12: A base class implementing an unbounded stack

```
coordinator Stack {
   selfex {push, pop};
   mutex {push, pop};
   int len=0;
   condition full=false,empty=true;
   push: requires !full;
   on_exit {
     empty=false;
     len++;
     if(len==buf.length) full=true;
   }
   pop: requires !empty;
   on_entry {len--;}
   on_exit {
     full=false;
     if(len==0) empty=true;
   }
}
```

Listing 2.13: An example of COOL

2.2.5 Join-Point Models

The join point model is a critical element in the design of any AOPL. The ability of a such language to support crosscutting lies in its *Join Point Model (JPM).* A JPM is defined by:

– The set of supported join points.
– A means of identifying join points.
– A means of modifying the execution at join points.

2.2.5.1 Pointcut-Advice Model

The *Pointcut-Advice* (PA) is the general model used for AspectJ. PA model is defined in [57, 77] by defining the three properties of a join point model.

The state of the art shows that there are two types of PA models, depending on the exact definition of a join point:

region-in-time This is the join point model in AspectJ-like languages, where a join point covers the *region in time* covering the whole execution of an instruction. In case of a call, this includes the whole execution until the call returns. In such model, a piece of advice is able to replace the execution of the instruction. Within the piece of advice, the instruction can alternatively be executed through the use of an instruction typically called `proceed`.

point-in-time This is a join point model proposed by Masuhara [76] where a join point represents a *point-in-time* (or an instant of program execution) such as the beginning of a method call or the termination of a method call rather than a region-in-time (or an interval). Whereas the region-in-time model associates an instruction to a single join point, the point-in-time model associates an instruction to two join points, one before the instruction and one after.

2.2.6 Aspect Mechanisms

In order to highlight the intricacies of weaving, basically what are the main issues with join point matching, aspect composition, we will give a short description of the AspectJ compilte-time weaving mechanism.

The AspectJ compiler accepts both AspectJ bytecode and source code and produces pure Java bytecode as a result in two stages. The weaving mechanism starts at the first stage where the front-end produces annotated bytecode (the goal of annotation is to handle non pure Java information as advices and pointcuts). At the end of this stage, aspects are available as annotated Java classes and their advices as Java method. The front-end extracts all the aspects informations and make them available at the next stage. The weaving mechanism continue at the second stage where the back-end of the AspectJ compiler instruments the bytecode by inserting calls to the precompiled advice method. It does this by considering that certain principled places in bytecode represent possible join points. These are called *static shadows* of those join points. The weaving mechanism at the back-end stage applied to every such static shadow checks each piece of advice in the system and determines if the advice's pointcut could match that static shadow. If it is the case, a call the advice's implementation method is inserted at the shadow. When several aspect match the same shadow, the back-end uses the information precedence construct for ordering

It is worth mentioning that the semantics of AspectJ has evolves since the first version ([18] for call and execution semantics). This shows that the implementation were based on target practical efforts rather than theoretical underpinnings while a well-understood model of the weaving process can guide us to better understand AOP and allows the implementation and the extension of new aspect mechanisms. There are two approaches to model aspect mechanisms [60]: *conceptual* and *formal.*

Conceptual models present a conceptual abstraction of the aspect mechanisms. Masuhara and Kiczales [77] model each aspect mechanism as a weaver that combines an aspect program and a base program at runtime. Kojarski *et al.* [63, 65] describe an abstract weaving process (Listing 2.14) for weaving that comprises four subprocesses: **reify**, **match**, **order**, and **mix**. These (sub)processes are found in all *reactive* aspect mechanisms [63]. MetaSpin [26],

```
public void weaveClass(ClassFile cf) {
  Shadow[] shadows = reify(cf);
  for(Shadow shadow:shadows) {
    Advice[] advs = order(shadow, match(shadow));
    mix(shadow, advs);
  }
}
```

Listing 2.14: The weaving mechanism.

```
  applyIntroductions(cf);
  weaveClass(cf);
```

Listing 2.15: The AspectJ weaving mechanism.

a metamodel that summarizes the common language concepts of AOPLs has been conceived accrding the survey and taxonomy of AOPLs and their execution models [25]. The metamodel describes the abstract grammar of AOPL while their execution semantics is expressed with an interpreter.

Formal models present formal semantic descriptions of aspect mechanisms. Lämmel [67] explains an aspect mechanism name Method-Call Interception (MCI). Wand *et al.* [101] give a denotational semantics for a minilanguage that embodies the key features of dynamic join points, pointcut designators, and advice. Djoko Djoko *et al.* [36] present the CASB, for Common Aspect Semantics Base, as a formal semantic descriptions of aspect mechanism.

In the rest of this section, we detail the abstract aspect weaving presented in [63, 65] and the CASB presented in [36].

2.2.6.1 Abstract aspect mechanisms

The abstract weaving process for weaving comprises four subprocesses: **reify**, **match**, **order**, and **mix**.

The **reify** process prepares the class file `cf` to the next steps by constructing a reified version of this class. In the case of AspectJ (`ajc` weaver), the weaver represents a class as a set of computation *shadows* and identifies all the shadows that can possibly be advised. Each shadow references a list of instructions embedded in one of `cf`'s methods (the body of the shadow), and provides static and lexical descriptions of these instructions (to be used by the match process).

The **match** process associates elements of the reified version (shadows) with pieces of advice. In AspectJ, the weaver selects the set of advice pieces by matching the description of the shadow (avaible after the **reify** process) against the static part of the advice pointcuts.

The **order** process sorts and orders all pieces of advice that match the same shadow into a correct application order according to the semantics of the AOPL. The `ajc` weaver orders the pieces of advice according to the rules defined by the AspectJ language semantics (using default ordering rules or `declare precedence` statements).

The **mix** process transforms a shadow by introducing calls to advices that match this shadow. The AspectJ weaver transforms the shadow by sequentially introducing calls to the advice methods before, after, or instead of the original code. If several pieces of advice match this shadow, they are woven in by sequentially transforming the body of the shadow.

Listing 2.14 shows the execution of the 4 subprocesses in the case of AspectJ.

The four processes provide a high-level description of the advice weaving semantics. A concrete weaver may also realize other kinds of transformations. For example, the `ajc` weaver implements *intertype declarations* and advice weaving in two separate steps 2.15. First, the weaver extends and transforms the class `cf` by applying the intertype declarations. Once the declarations are applied, the weaver calls `weaveClass` method, which implements the advice weaving behavior. The additional transformations are normally static in nature, and do not interfere with the dynamic advice weaving behavior.

2.2.6.2 CASB

The CASB model starts from a simple model of a base language, and then considers the necessary extensions for supporting aspects. The simple model of the base language is based on a reduction semantics, where each reduction step maps a *configuration* consisting of a sequence of instructions and an interpreter state, to a new sequence of instructions and a new interpreter state. The aspect language is derived from this, but extends the base configuration and the base language reduction rules in order to handle aspects. A base program C is modeled as a list of base *instructions* i terminated by the empty instruction ϵ:

$$C ::= i : C | \epsilon$$

A configuration is a tuple (C, Σ) where Σ represents the state of the interpreter. Σ contains an environment, a stack, a heap, etc., depending on the semantics of the considered language and the details of its implementation. The semantics of the base language is then defined by a relation $\rightarrow_b$ on configurations, which defines single reduction steps of the form (the second reduction rule defines the semantics of *blocks*):

$$(i : C, \Sigma) \rightarrow_b (C', \Sigma')$$
$$(\{i_1, ..., i_n\} : C, \Sigma) \rightarrow_b (i_1 : ... : i_n : C', \Sigma')$$

In the presence of aspects, the semantics of a program is captured by a new relation $\rightarrow$ on extended configurations (C, Σ, P), where P, the *proceed stack*, is used to deal with around advices [1].

The issue is then to explain this new relation $\rightarrow$ in terms of the base relation $\rightarrow_b$. This requires modeling aspects and introducing new reduction rules for "weaving" these aspects to the base program.

Aspects are modeled by the functions ψ and ϕ. The function ψ captures both static weaving and the execution of the woven code. It determines, based on *static* information (that is, syntactic information, present in the program text) which aspects *statically* match at each instruction. These statically matching aspects are modeled by the functions ϕ. Executing a function ϕ consists of checking whether the aspect *dynamically* matches, in which case the advice is executed. In the CASB, a function ϕ must be passed as a parameter to an instruction `test` in order to be executed. In order to simplify the presentation, we will rather assume that statically matching aspects are directly returned as executable instructions. In the following, ϕ will therefore be an instruction rather than a function.

Let us see what happens when a base instruction i is executed. When no aspect statically matches, $\psi(i)$ returns an empty list of statically matching aspects and the woven program simply behaves as the base program:

$$\text{NoAdvice}\ \frac{\psi(i) = \epsilon \qquad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma, P) \rightarrow (C', \Sigma', P)}$$

Otherwise, ψ returns a non-empty list of instructions ϕ denoted by $\phi : \Phi$:

$$\text{Around}\ \frac{\psi(i) = \phi : \Phi}{(i : C, \Sigma, P) \rightarrow (\phi : \texttt{pop} : C, \Sigma, (\Phi : [\bar{i}]) : P)}$$

1. As before and after advices can be explained in terms of around advices, whose handling is anyway more complex, we only consider *around* advices.

$$\textsc{Advice}\ \frac{\phi(\Sigma) = a}{(\phi : C, \Sigma, P) \rightarrow (a : C, \Sigma, P)}$$

The instruction i is pushed on the top of the proceed stack so that it can be possibly executed by a proceed. The proceed stack is organized as a stack of instructions. The execution of ϕ is followed by an instruction `pop`. When ϕ is the current instruction, the rule ADVICE applies ϕ it to the current state Σ in order to insert the corresponding advice.

The instruction `pop` simply trims the stack:

$$\textsc{Pop}\ \frac{}{(\texttt{pop} : C, \Sigma, \Phi : P) \rightarrow (C, \Sigma, P)}$$

This means that, in the absence of an instruction `proceed` in an advice, the remaining aspects together with the join point itself will never be executed.

Let us now see what happens when an instruction `proceed` occurs in an advice:

$$\textsc{Proceed}\ \frac{}{(\texttt{proceed} : C, \Sigma, \phi : \Phi : P) \rightarrow (\phi : \texttt{push}\ \phi : C, \Sigma, \Phi : P)}$$

An instruction `push` follows the execution of the instruction ϕ. This instruction does not put a new element on top of the proceed stack but rather adds an instruction at the beginning of the stack:

$$\textsc{Push}\ \frac{}{(\texttt{push}\ \phi : C, \Sigma, \Phi : P) \rightarrow (C, \Sigma, (\phi : \Phi) : P)}$$

The instruction `push` is used to rebuild the initial aspect set in order to cater for multiple instructions `proceed` in the same advice.

2.2.7 Approaches for Implementing AOPLs

From the programmer's viewpoint, an AOPL makes it possible to write different modules encapsulating different concerns. From the language designer's viewpoint, some of these modules, the aspects, crosscut other modules and have to be mixed with them at the appropriate point. This can be done at two different times: at compile time or at execution time. As a result, there are two basic ways to implement AOPLs:

1. A combined base program is produced, from the base program and the aspect program.
2. The interpreter are updated to understand and implement AOP features.

Let us formally describe the difference between these two approaches. Suppose that we have a language B, with a defining interpreter for B int_B. We want to study the semantics of an AOPL that integrates a certain aspect extension, A, with the base language, B. This semantics will explains *how* p_B, a program in B, is executed in the presence of p_A, a program written in the aspect extensions A. The association between these two programs is represented as : $p_B \Leftarrow p_A$, or p_A is weaved into p_B.

In the following, we assume that input data is included in the program (think about a Java `main` method).

Transformation Approach The first approach uses a *transformation* approach to introduce the semantics of AOPL. The result is a program in the base language:

$$p_B \Leftarrow p_A = T_A(p_B, p_A) = p'_B \tag{2.1}$$

and the resulting program is executed by int_B:

$$output = int_B(p_B \Leftarrow p_A) = int_B(p'_B) = int_B(T_A(p_B, p_A)) \tag{2.2}$$

The semantics of the considered AOPL is defined by the function T_A where the index of T_A symbolizes the aspect extension A. Note that each AOPL must provide a semantics for composing several aspects (aspects written in this AOPL) to explain the order in which the transformation has to be applied.

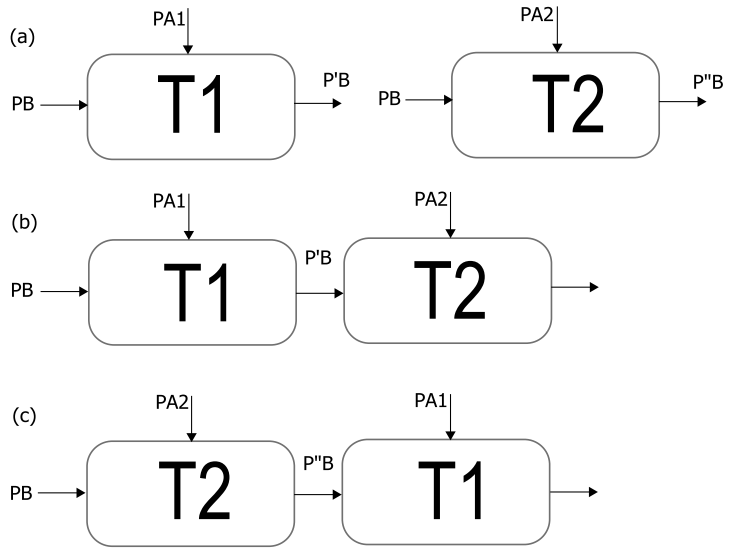

Figure 2.5: Sequential instrumentation

Interpretation Approach The second approach consists of changing the interpreter of the base language according to the semantics of the AOPL:

$$output = int_B(p_B \Leftarrow p_A) = int_B(T(p_B, p_A)) = (T'_A(int_B))(p_B, p_A) \tag{2.3}$$

and this can be written as:

$$output = int_B(p_B \Leftarrow p_A) = int_{B \Leftarrow A}(p_B, p_A) \tag{2.4}$$

where

$$int_{B \Leftarrow A} = T'_A(int_B) \tag{2.5}$$

T'_A is a transformation depending on the semantics of the AOPL which, applied on int_B returns $int_{B \Leftarrow A}$, an interpreter of the extended language.

The interpreter can also be defined as follows:

$$int_B(p_B \Leftarrow p_A) = int_{B \Leftarrow A}(p_B, p_A) = int_B(p_B) \Leftarrow int_A(p_A) \tag{2.6}$$

The operator $\Leftarrow$ applied on $int_B(p_B)$ and $int_A(p_A)$ is meaningful because it describes the interactions between the base interpreter and the aspect interpreter. This issue will be considered in Chapter 5.

To summarize, each time we want to extend an existing base language with AOP features, the *transformation* functions T or T', depending on which approach we use, have to be defined.

2.2.8 Approaches for Prototyping and Composing AOPLs

Now let us see what happens when several aspect extensions have to be composed. There are two approaches to composing aspect languages: *Sequential instrumentation* and *Translation* [60].

Sequential transformation (See Figure 2.5) We have two aspect languages: A_1 and A_2. Suppose that A_1 and A_2 were independently developed, then we have the following equations for A_1 and A_2 respectively:

$$output_1 = int_B(p_B \Leftarrow p_{A_1}) \tag{2.7}$$

$$output_2 = int_B(p_B \Leftarrow p_{A_2}) \tag{2.8}$$

When composing the two extensions with B, we get:

$$output = int_B(p_B \Leftarrow (p_{A_1}, p_{A_2})) \tag{2.9}$$

Let us assume that weaving can be linearized:

$$p_B \Leftarrow (p_{A_1}, p_{A_2}) = (p_B \Leftarrow p_{A_1}) \Leftarrow p_{A_2} \tag{2.10}$$

$$(p_B \Leftarrow p_{A_1}) \Leftarrow p_{A_2} = T_{A_1}(p_B, p_{A_1}) \Leftarrow p_{A_2} \tag{2.11}$$

$$T_{A_1}(p_B, p_{A_1}) \Leftarrow p_{A_2} = T_{A_2}(T_{A_1}(p_B, p_{A_1}), p_{A_2}) \tag{2.12}$$

$$T_{A_2}(T_{A_1}(p_B, p_{A_1}), p_{A_2}) = T_{A_2} \circ T_{A_1}(p_B, p_{A_1}, p_{A_2}) \tag{2.13}$$

Alternatively:

$$(p_B \Leftarrow (p_{A_1}, p_{A_2})) = ((p_B \Leftarrow (p_{A_2})) \Leftarrow (p_{A_1})) \tag{2.14}$$

$$((p_B \Leftarrow (p_{A_2})) \Leftarrow (p_{A_1})) = (T_{A_1}(p_B, p_{A_2}) \Leftarrow (p_{A_2})) \tag{2.15}$$

$$(T_{A_1}(p_B, p_{A_2}) \Leftarrow (p_{A_2})) = T_{A_2}((T_{A_1}(p_B, p_{A_1}), p_{A_2})) \tag{2.16}$$

$$T_{A_2}((T_{A_1}(p_B, p_{A_1}), p_{A_2})) = ((T_{A_1}) \circ (T_{A_2}))(p_B, p_{A_1}, p_{A_2}) \tag{2.17}$$

Here we see that multiple independent aspect extensions can be trivially composed by passing the output of one transformation as the input to another transformation. But this composition leads to several confusions as it was demonstrated by Kojarski *et al.* [70] where a transformation could be erroneously applied into a code introduced by a previous transformation. This problem will be described in Chapter 4. The confusion means also that weaving order matter and let us write that $T_{A_1} \circ T_{A_2} \neq T_{A_2} \circ T_{A_1}$.

Translation (See Figure 2.6) Translation approach means that aspect programs in different aspect extensions (T_1 and T_2) can be translated to a common target aspect extension (T_3). The composition of the two aspects from T_1 and T_2, respectively, is simplified into the composition of two aspects in T_3, where the semantics of the composition of two aspects should be defined.

2.3 Summary

In this chapter we have exposed what is required about AOPLs for our work in the coming chapters. After presenting elements that define programming languages and their implementations, we have switched to the description of three concrete approaches then we have described the aspects mechanisms, mechanisms that differentiate a standard programming language from an AOPL. We have given a short formal description of these mechanisms. We have shown the gap between the formal model, the conceptual model and the implementation of an AOPL, which makes it difficult to extend, prototype and compose AOPLs. We have also shown how we can provide a generalized mechanism in order to define frameworks for prototyping and composing AOPLs, which are the subject of the next chapter where we examine the state of the art of such systems.

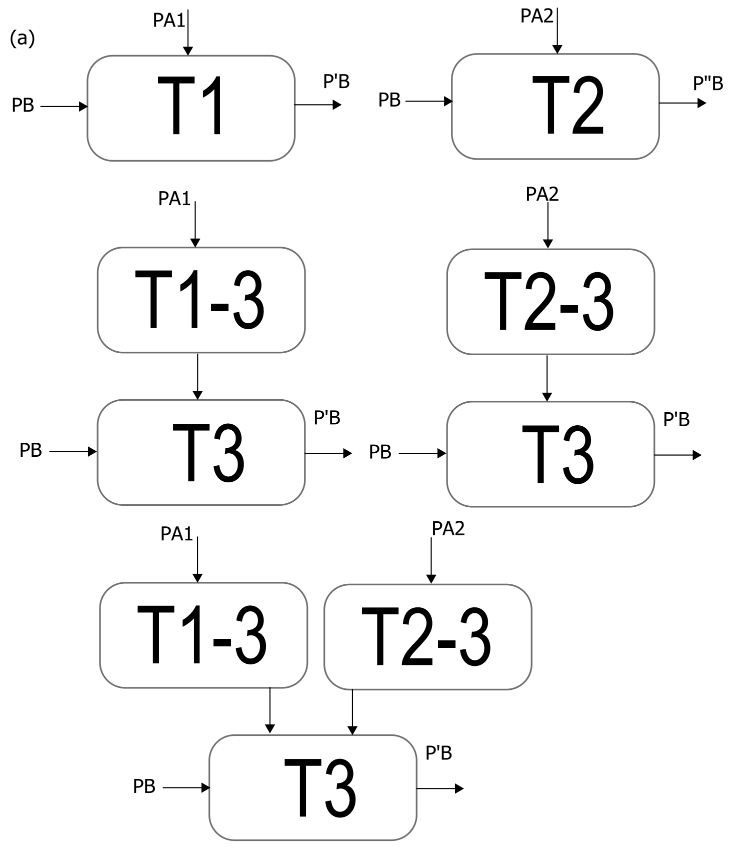

Figure 2.6: Common target transformation

Chapter 3

Prototyping and Composing Aspect Languages

Contents

As we saw in the previous chapter, the two functions T and T' are the core of the semantics of AOPLs. Every time an AOPL is implemented, these functions must be defined, even implicitly. This is a tedious and time consuming job, especially when prototyping a new AOPL from scratch. Once an AOPL is implemented, it is difficult to combine it with other AOPLs, whereas such a combination would be useful, for instance when working with multiple DSALs. A good example of such a combination is when we work with multiple DSALs. To summarize, there are two requirements when developing AOPLs:

- Support for designing and rapid prototyping of an AOPL.
- Support for customized composition of AOPLs.

In this chapter, we are going to explore the state of the art of existing systems for prototyping and composing AOPLs. For each system, we separate the study into two sections, one for discussing prototyping and one for discussing composition. For each framework, we specify whether the implementation technique is based on transformation or interpretation.

3.1 Prototyping and Composing AOPLs

A framework for prototyping AOPLs must provide a convenient refinement of one of the functions T and T'. Today, there are few frameworks for prototyping and composing AOPLs. The majority of these frameworks use a type of general transformation mechanism. The implementation of a new AOPL consists of translating aspects using the general transformation provided by the framework. For example, implementing a new AOPL with Reflex consists of transforming aspect programs into a *configuration class* of Reflex. The transformation mechanism of Reflex is based on partial behavioral reflection [92]. XAspects consists of translating aspects written in DSALs into AspectJ itself. XAspects uses the transformation mechanism of AspectJ. In general, a framework for prototyping AOPLs provides support for composing different AOPLs implemented with this framework by composing different modules written in the same intermediate representation provided by the framework.

3.1.1 Design Space for AOPLs

Defining a programming language, in general, requires to define the language syntax (concrete and abstract) as well as its semantics. Defining an aspect language is in no way different. As far as syntax is concerned, a current practice [25] is to extend the grammar of the base language. To introduce the semantics of aspects, there are two possible approaches:

- Rewriting the base application with respect to the aspects [2, 15]. The rewriting can be done using different techniques like program transformation [2, 15], reflection [20, 61, 92], etc. The result is a program in the base language and its interpretation is performed by the standard interpreter.
- Modifying the mechanisms of the base interpreter [46, 26] to support aspect mechanisms. Weaving is performed dynamically and the aspect is a *runtime entity* in the interpreter. Steamloom [19] is the first VM implementation to natively support dynamic aspects.

The second approach has the advantage to provide an easy access to language semantics, in spite of lower performance. In this respect, a study of the state of the art [25] shows that each aspect language uses specific interpreter modifications. Understanding and extending aspect mechanisms remains difficult. There is a lack of a good design space representation which would provide a common framework for understanding and evaluating existing mechanisms. A well-defined model of the weaving process would guide the designer of new aspect mechanisms. Several papers have focused on the modeling of aspect mechanisms [97, 63, 79, 78, 77]. Most of them do not attack the problem from a language design point of view and remain at a very abstract level. Metaspin [26, 24] is a first attempt at defining a design space for prototyping and testing aspect languages in a common framework and a source of inspiration for this work.

3.1.2 Combining AOPLs

The use of GPALs is now quite common in software development. However, when using aspects to implement different concerns like concurrency, logging, security, etc., DSALs have many advantages over GPALs as discussed in [62, 92].

Different concerns crosscut in a program. Each concern should be expressed using a different DSAL. The aspects written in different languages are composed and woven into the base program. The problem of composing aspect languages results in an *aspect mechanism composition problem* [62]. Consider two new aspect languages L_1 and L_2 merged with Java as the base language. Each of L_1 and L_2 have their own aspect mechanisms. The meaning of composing a base program *base* and an aspect $aspect_1$ written in L_1 is well defined. This is also true of the composition of *base* with a second aspect $aspect_2$ written in L_2. However, the semantics of composing *base*, $aspect_1$, and $aspect_2$ is undefined.

Unfortunately, there is no methodology and support for the integration of distinct aspect mechanisms. For each AOPL, there is an infrastructure that is constructed in an ad hoc manner, aspect mechanisms are differently represented and implemented and as a result, it is difficult to integrate them.

As we will see, the diversity of aspect mechanisms affect the ability to compose AOPLs. The combination of different AOPLs can be facilitated by using the same methodology and design space.

In the following, we look at four representative approaches: Reflex, XAspects, Metaspin and Pluggable AOP by exploring their ability to support the prototyping and the composition of AOPLs as well as the resolution of interactions when multiple AOPLs are used.

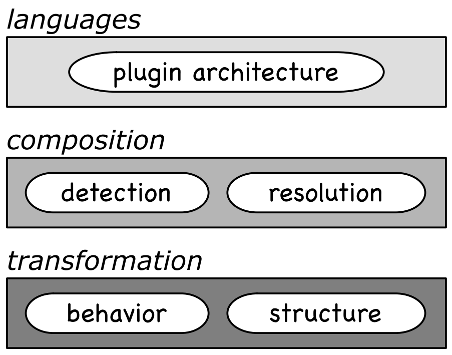

Figure 3.1: The architecture of the Reflex kernel [92].

```java
ClassSelector cut = new ClassSelector() {
  boolean accept(RClass aClass) {
    return aClass.getName().equals("A");
  }
};
```

Listing 3.1: An example of a class selector

3.2 Reflex

Reflex [93, 42, 92] is a Java framework that extends Java with structural and behavioral reflective abilities. The generality of Reflex abilities makes it possible to use it as a suitable *backend* for implementing several AOP frameworks and languages: it provides, in the context of Java, building blocks for facilitating the implementation of different AOPLs so that it is easier to experiment with new AOP concepts and languages, and also possible to compose aspects written in different AOP languages. It is built around a flexible intermediate model, derived from reflection, of (point)cuts, links, and metaobjects, to be used as an intermediate target for the implementation of AOPLs.

Architecture The architecture of Reflex (Figure 3.1) is as follows:

The transformation layer is based on a reflective core [93], which extends Java with behavioral and structural reflective facilities. At load-time, this layer takes the base program and, according to a configuration class, injects hooks into the base program in order to support behavioral reflection. To do this, Reflex relies on Javassist [8] for byte-code manipulation. At runtime, the injected hooks send reifed execution information to metaobjects that are executed in order to modify (*aspectize*) the program execution.

The composition layer ensures automatic detection of aspect interactions at injection, and provides expressive means for their explicit resolution [87].

The language layer consists of translating aspects into the upper interface of the transformation layer [88]. The semantics of an AOPL is expressed by means of this translation.

Links and metaobjects The transformation layer of Reflex relies on the notion of an explicit *link* binding a *cut* to an *action*, similar to PA model. This makes it possible to define aspects as modular units comprising more than one pair cut-action. There are two types of links: *structural* and *behavioral*. A *structural link* binds a structural cut to some action, either structural such as adding a new interface, or behavioral, such as an additional computation before a method call. In Reflex, a structural cut is a *class set*, defined intentionally by a *class selector*.

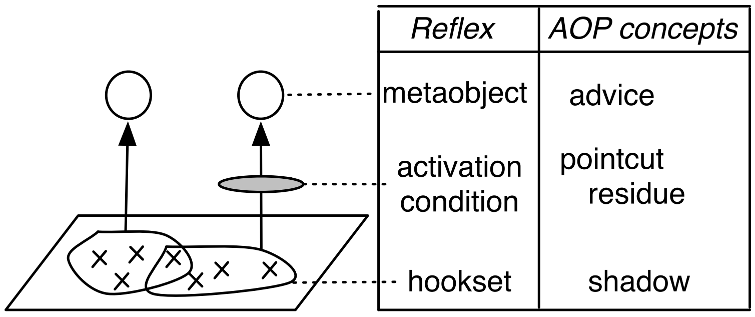

Figure 3.2: Behavioral links and correspondence to AOP concepts according to [93]

```
Hookset theHookset = new PrimitiveHookset(
                    //We want to hook field accesses...
                    FieldAccess.class,
                    // ...on all classes...
                    AllCS.getInstance(),
                    //...but only for field writes, excluding field
                        read
                    FieldWriteOS.getInstance());

                    // Create and add the link to the config
BLink theLink = addBLink(theHookset,
                    new MODefinition.MOClass(MO_CLASSNAME));
```

Listing 3.2: A part of a Reflex configuration class

As an example of cuts, the class selector `cut` in Listing 3.1 defines a cut that selects the class `A` only. An instance of the class `RClass` is the reflective representation of a class in Reflex.

A *behavioral link* binds an element of the execution control flow, such as a method call, to an action (see Figure 3.2). For the majority of the execution points, there is a corresponding program point (shadow) in the source code. Additional code at the source level introduces additional behavioral at runtime. The model of Reflex is based on a standard model of behavioral reflection, where *hooks* are inserted in a program to delegate control to a metaobject (for the additional code) at appropriate places (shadows). A set of hooks can be grouped into a *hookset*. A hookset corresponds to a set of program points, and the metaobject corresponds to the action to be performed at these program points. The link is characterized by several attributes like an activation condition, which may be attached to the link in order to avoid reification when a dynamically-evaluated condition is false. Listing 3.2 shows a part of a Reflex configuration class. A simple AspectJ aspect, consisting of a single piece of advice associated to a simple pointcut, is straightforwardly implemented in Reflex with a metaobject, a hookset and a link. However, most practical AOP languages, like AspectJ, make it possible to define aspects as modular units comprising more than one cut-action pair. In Reflex this corresponds to different links, with one action bound to each cut. Furthermore, AspectJ supports higher-order pointcut designators, like `cflow`. In Reflex, the implementation of such an aspect requires an extra link to expose the control flow information. There is therefore an abstraction gap between aspects and links: a single aspect may be implemented by several links.

An interaction occurs in Reflex when several hooks have to be inserted at the same program point. Reflex supports:

– automatic detection of aspect interactions limiting spurious conflicts;

```
public aspect Logging {
  pointcut p(): call (* *.*(..));

  before(): p(){
    System.out.println("method-call of "+
                       thisJoinPoint.getSignature().getName());
  }
}
```

Listing 3.3: An AspectJ Logging aspect

- aspect dependencies, such as implicit cut and mutual exclusion;
- extensible composition operators for ordering and nesting of aspects;
- control over the visibility of structural changes made by aspects;
- aspects of aspects.

Prototyping AOPL Prototyping an AOPL using Reflex means defining a plugin within the language layer on top of the two other layers. It consists of defining the translation of aspects from this AOPL to Reflex representation (hooksets, links and metaobjects). Hence, Reflex uses a translation approach to translate aspects from different languages. The primitive means to configure Reflex are configuration classes. To raise the level of abstraction, plugins can be implemented: a plugin supports an AOPL, and is in charge of generating the appropriate Reflex configuration. Before talking about the translation, let us see the implementation of an AspectJ aspect using a Reflex configuration class.

Consider the AspectJ `Logging` aspect as shown in Listing 3.3, which logs all the method calls then print the name of the called method. In Reflex, this can be implemented using a `LoggingConfig` configuration class and a `Logger` metaclass:

According to this correspondence between hooksets and pointcuts, metaobject and advice, aspects and links, etc. a plugin implementing a subset of AspectJ was developed [84, 88]. A new version of the plugin uses Metaborg [23, 22, 43, 95] in the language layer by defining the language grammar in SDF and uses Stratego to transform AspectJ aspects into Reflex configuration classes. This approach is called ReflexBorg and it is based on:

- SDF (Syntax Definition Formalism) is an extensible and modular language for defining syntax [98]. Definitions (derivations, lexical restrictions, terminals, keywords, etc.) are done in modules that can be extended and reused.
- Stratego and XT [99] represents a powerful machinery for program transformation. Stratego is a declarative language for transforming trees through the application of rewrite rules, composed by means of rewriting strategies for modular transformation and fine-grained control over their application. Rewrite rules can use the concrete syntax of the host language in their definitions.

  Finally, XT is a toolset which offers a collection of extensible and reusable transformation tools such as the SGLR parser used in conjunction with SDF.

Reflex was also used to implement *KALA*, a DSAL for managing transactions [43, 44], Sequential-Object Monitoring [28] and was extended to support distribution [96].

Composing AOPLs When composing several AOPLs implemented using Reflex (a plugin for each language), aspects from all AOPLs have the same representation (hookset, metaobject, links) and the problem of composing AOPLs is simply turned into a problem of composing Reflex aspects. A designer of multi-AOPLs composition uses Reflex composition rules to resolve the interactions between the translated aspects. Let us explore the rules of composing aspects in Reflex.

Aspect dependencies, actually link dependencies can be of two kinds: implicit cut (*apply l1 whenever l2 applies*) and mutual exclusion (*never apply l1 if l2 applies*).

```
public class LoggingConfig extends ReflexConfig
{
  public void initReflex() {
    // Create a hookset
    Hookset theHookset = new PrimitiveHookset(
                       MsgSend.class,// ...method-call
                       AllCS.getInstance(),   // ...on all classes
                       new DeclaredInOS("*"));// ...all the method
    // Create dynamic link
    BLink theLink = addBLink(theHookset,
                       new Logged());// instantiating a metaobject
    // Further configure the link:
    // There should be one system-wide instance of the metaobject
       ...
    theLink.setScope(Scope.GLOBAL);
    // ...which should be called before the operation occurs
    theLink.setControl(Control.BEFORE);
    // Define the MOP: the metaobject should have its log method
    // called with no arguments.
    theLink.setMOCall(Control.BEFORE,
                       new CallDescriptor(MO_CLASSNAME,
                       "log", new Parameter[0]));
  }
}

public class Logger extends MODefinition {
  public void log() {
    System.out.println("method-call");
  }
}
```

Listing 3.4: The translation of the aspect in Listing 3.3 in a configuration class of Reflex

```
  public AspectPlugin(CompilationEnvironment ce);
  abstract void recieveBody(AspectInfo ai, String aspectID, String
     body);
  abstract File[] generateExternalInterfaces();
  abstract File[] generateCode(File[] classfiles);
  abstract void cleanUp();
}
```

Listing 3.5: The class `AspectPlugin` provided by XAspects

An implicit cut is obtained by sharing the cut specification between two aspects like when two aspects in AspectJ share the same definition of a pointcut. In the following example, `theLink` and `anotherLink` share their hookset because `anotherLink` is instantiated with the hookset of `theLink` (by calling `theLink.getHookset()`):

```
BLink anotherLink = Links.get(theLink.getHookset(), <mo>);
```

Mutual exclusion between two aspects is obtained by declaring that a link should not apply if another one does. The following statement makes `theLink` and `newLink` mutually exclusive:

```
BLink newLink = Links.get(new Hookset(), <mo>);
Rules.declareMutex(theLink, newLink);
```

If links are mutually exclusive, specifying their ordering is not necessary. Otherwise, ordering must be specified. Consider the general case of *around* advice. In Reflex, the composition is done by a set of operators: *seq* and *wrap*. The rule $seq(l1, l2)$ ($l1$ and $l2$ are ordered) uses the seq operator to state that $l1$ must be applied before $l2$. The rule $wrap(l1, l2)$ ($l1$ and $l2$ are nested) means that $l2$ must be applied within $l1$.

3.3 XAspects

XAspects [86, 85] is a plugin mechanism for DSALs, where aspects are translated at compile time, into general AspectJ source code, which serves as a base language from which other AOPLs can be defined. Developing a new AOPL consists of specifying the translation between aspects from this language to aspects in AspectJ. This is similar to what is done in Reflex where plugins translate aspects to Reflex configuration classes.

Differently to Reflex, XAspects does not provide a transformation layer but instead, it uses AspectJ compiler. XAspects modifies the grammar and the compiler (front-end) of AspectJ in order to forward some compilation phases into XAspects plugins. The interaction between the plugins and the XAspects compiler (xajc) is displayed as a 6-phase-compilation process (see Figure 3.3): *Source Code Identification*, *Generation of External Interfaces*, *Initial Bytecode Generation*, *Crosscutting Analysis*, *Generation of Semantics*, *Final Bytecode Generation.*

The grammar extension consists of adding this new rule to the grammar of AspectJ:

```
<Aspect> ::= ["privileged"] [<Modifiers>]
             "aspect" "(" <Type> ")" <Id>
             ["extends" <Type>] ["implements" <TypeList>]
             "{" [<BalancedCurlies>] "}"
```

The *Type* token corresponds to the name of the aspect extension. Each aspect extension must be implemented as an extension of the class `AspectPlugin` (see Listing 3.5) provided by XAspects API. Note that each extension must provide its own grammar and parser, which is used by each plugin.

The parser modification consists of performing the following operations when detecting the above rule:

```
aspect(Traversal) FileSystemTraversals {
    declare strategy: eachFile: "intersect(from CompoundFile to
       File, down)";
    declare traversal: void listAll(): eachFile (FileLister);
    declare strategy: down: "from * bypassing -> *,parent,* to *";
    declare strategy: up: "from * bypassing -> *,contents,* to *";
}
```

Listing 3.6: An aspect written in `Traversal` DSAL

– Finding the class specified by the matched *Type* in the XAspects package (phase of Source Code Identification). As we say, this class corresponds to the aspect extension and it is called the plugin class.
– Instantiating the plugin class with an instance of the class `CompilationEnvironment`. Note that all plugin classes implement the Singleton pattern in order to have only one instance during compilation.
– Invoking the method `receiveBody` on the plugin instance by providing the string representation of the rule *BalancedCurlies*, which represents the body of the aspect and belongs to the grammar of the aspect extension.

The parser applies the above operations on each plugin before *sequentially* calling the method `generateExternalInterfaces` of each plugin (phase of Generation of External Interfaces). This method returns a list of source files into a temporary disk location. All new aspects, classes, methods or fields that the plugin introduces for external use are specified during this phase. This method shows that XAspects follows the translation approach described in Chapter 2. The AspectJ compiler performs an initial bytecode generation by compiling the base program in addition to the generated files and returns an array of classes (phase of Initial Bytecode Generation). The drawbacks of XAspects are that, during this last phase, plugin-specific code could be erroneously matched by the AspectJ aspects generated by other plugins. This problem is discussed in Chapter 4. For each plugin, the method `generateCode` performs, if needed, behavioral changes to the generated program and returns a list of generated AspectJ source files (phase of Generation of Semantics). These files are compiled in order to generate the final bytecode (phase of Final Bytecode Generation).

Prototyping AOPL Prototyping a new AOPL with XAspects consists of providing an extension of the class `AspectPlugin` (see Listing 3.5) as well as an extension of the AspectJ grammar. Note that any extension of the class `AspectPlugin` must implement the three main methods: `receiveBody`, `generateExternalInterfaces` and `generateCode`. To clarify the extension mechanism, we describe two plugins for Traversal and AspectJ, respectively.

Listing 3.6 shows an example from the web site of XAspects [85], where an aspect is defined in the Traversal plugin. The aspect defines different strategies to traverse a graph of classes.

The implementation of the Traversal extension is based on the class `Traversal` (see Listing 3.7) and reuse DAJ [3] (Demeter in AspectJ). DAJ is accessible via the variable `DAJ`. The class `Traversal` extends the abstract class `AspectPlugin` and implement its main methods. For instance, `receiveBody` does not perform any processing in the incoming source code but stores the received code in a `.trv` file with the corresponding `aspectID` as the filename. The method `generateExternalInterfaces` calls the method `DAJ.generateStubs(List, File)` and generates Traversal Stubs. The method `generateCode` is responsible of applying bytecode transformation. It takes the bytecode files and generates *Traversals* using the method `DAJ.generateTraversals(List, File, boolean, File)`. The method `cleanup` deletes temporary files created for storing redirected outputs.

The AspectJ plugin is included for completeness purpose and it is simpler than any other plugin. Listing 3.8 shows the AspectJ plugin. The helper method `generatedFiles` scans the working Directory for all `.java` files, collects them in an array of File objects are returns

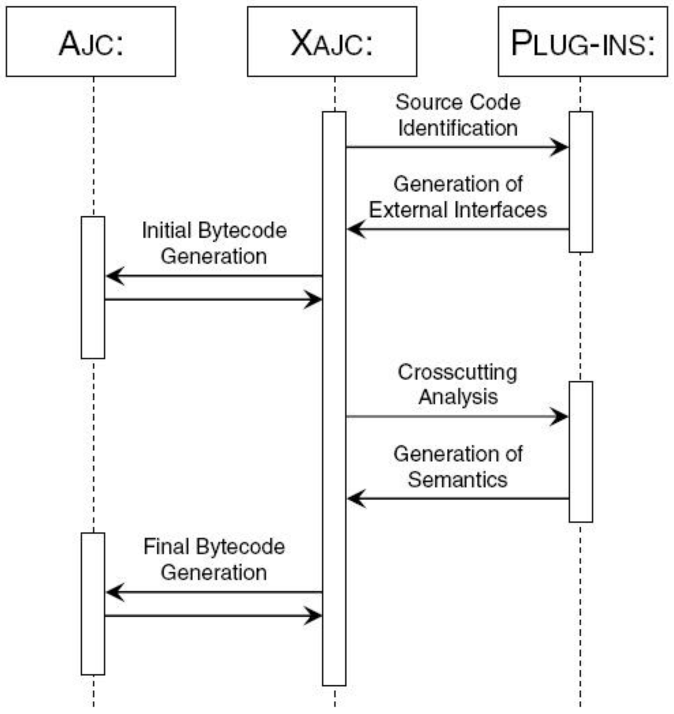
AJC:
XAJC:
PLUG-INS:
Source Code Identification
Generation of External Interfaces
Initial Bytecode Generation
Crosscutting Analysis
Generation of Semantics
Final Bytecode Generation

Figure 3.3: The architecture of XAspects.

```
public class Traversal extends AspectPlugin{
  public Traversal (){ }
  public void init (CompilationEnvironment ce){ .. }
  public void receiveBody(AspectInfo aspectInfo, String aspectID,
     String body){ .. }
  public File[] generateExternalInterfaces(){ .. }
  public File[] generateCode(File[] classFiles){ .. }
  public void cleanup(){ .. }
  private void partialCleanup(){ .. }
  private void printErrors (){ .. }
  private void printErrors2 (){ .. }
  private void redirectOutputs () throws FileNotFoundException{ ..
     }
  private void restoreOutputs (){ .. }
  private File[] generatedFiles(){ .. }
  private void copyFiles(File[] source, File dest){ .. }
}
```

Listing 3.7: The Traversal implementation using XAspects

```java
public class AspectJ extends AspectPlugin{
  public AspectJ (){ }
  public void init (CompilationEnvironment ce){ }
  public void receiveBody(String aspectID, String body){ }
  public File[] generateExternalInterfaces(){ }
  public File[] generateCode(File[] classFiles){ }
  public void cleanup(){ }
  private File[] generatedFiles(){ }
}
```

Listing 3.8: The AspectJ implementation using XAspects

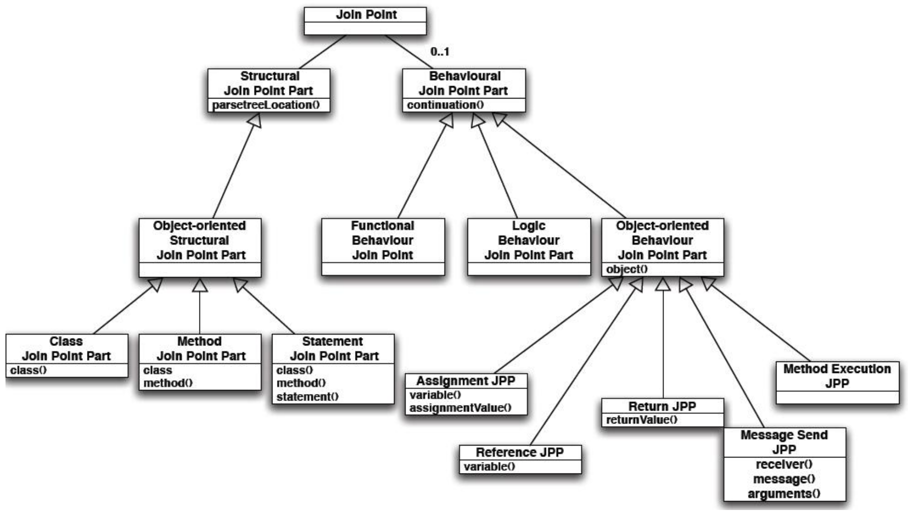

Figure 3.4: The join point metamodel as described by Metaspin [26].

the array. The method `receiveBody` stores all the received code in a `.java` file without any checking. The two methods `generateExternalInterfaces` and `generateCode` simply return the list returned by the method `generatedFiles`.

Composing AOPL Because XAspects plugins translate DSAL to AspectJ, a composition of DSALs is turned into a composition of different AspectJ aspects. The composition is done during the phases of Crosscutting Analysis and Generation of Semantics. There is no support for the configuration of the composition to control aspect weaving. An aspect generated by plugin X can erroneously match join points in the body of an aspect generated by another plugin Y and this can lead to several problems. These problems are described in the section about aspect interaction in Chapter 4.

3.4 Metaspin

Following the survey and taxonomy of AOPLs and and their execution models performed within the AOSD European Network of Excellence [25], a metamodel that summarizes the common language concepts of AOPLs has been conceived. While the metamodel describes the abstract grammar, an interpreter defines operationally the semantics of the metamodel. The Metaspin interpreter is an implementation of the semantics of the metamodel in Smalltalk [26].

- The metamodel has been conceived as an open and extensible framework that makes it possible to describe and represent the essential aspect language features and their relations. It is composed of 4 sub metamodels: *join point*, *pointcut*, *aspect binding* and *advice* metamodels. It can be seen as an abstract grammar of a core aspect language.
  - **Join-Point Metamodel**: The metamodel represents the concept of a join point in the aspect language. It typically depends on the base programming language in which the aspect language is integrated. As an illustration, this model is shown in Figure 3.4.
  - **Pointcut Language Metamodel**: This metamodel describes the concept of a pointcut as a *Join Point Selector* that applies a *Predicate* to the join point.
  - **Advice Binding Metamodel**: This metamodel describes the instantiation, scoping, and modularization of aspects, as well as the binding of advices to pointcuts.
  - **Advice Metamodel**: The actions that can be triggered by aspects at particular join points are described using the advice metamodel. The advice metamodel is also closely related to the choice of a particular language for the advice.
- The interpreter describes the semantics of the core language. It implements the operational semantics of all language concepts. As a result, it can interpret programs that are described using these concepts. This means that the interpreter can interpret aspect-oriented programs whose implementations have been mapped onto implementations in terms of the concepts in the metamodel. The interpreter seemed the most convenient, easiest and quickest way to get working implementations of a semantics, thus making it all a bit more concrete, which facilitates experimenting with prototype languages.

3.4.1 Execution Semantics

3.4.1.1 Base and Metalevel Aspect Interpreter

In Metaspin, the evaluation of all program entities that are expressed using the base language are executed by the *base interpreter*. Similarly, all program entities called *metalevel instructions* and expressed using aspect-oriented language concepts are executed by the *metalevel aspect interpreter*. More specifically, the metalevel aspect interpreter evaluates aspect programs that are expressed using concepts of the metamodel. As a consequence, the semantics of the aspect-oriented language concepts are localized in the definition and the implementation of the metalevel aspect interpreter. The metalevel aspect interpreter is both an observer and a controller of the base language interpreter.

3.4.1.2 Interaction between the two interpreters

The mechanism of interaction between the two interpreters consists of communicating join points to the metalevel aspect interpreter at every discrete evaluation step. The discrete steps correspond to the evaluation of each *atomic* base program expression. Atomic expressions are the basic expressions evaluated by the base interpreter, they are not defined in terms of other expressions.

At each evaluation step, the base language interpreter stops the execution of the program at hand, creates a join point that represents the current execution state and passes control to the metalevel aspect interpreter. The evaluation of a base instruction *expB* gives rise to two join points, one before and another after the evaluation of this instruction. The aspect interpreter can then decide to invoke an aspect at this join point or to let the base interpreter continue its normal evaluation (with the base language semantics). The evaluation of pieces of advice is performed by the base interpreter because they (mostly) contain base instructions.

Consider the following base program where ":" is a right associative operator constructing a program from an expression and another program and ϵ is the empty program:

$$B = exp1 : exp2 : exp3 : \epsilon$$

The program consists of 3 atomic expressions. This program is evaluated by the base interpreter. Figure 3.5 displays 4 *join points*, each one appears *in-between* join points. Each execution point

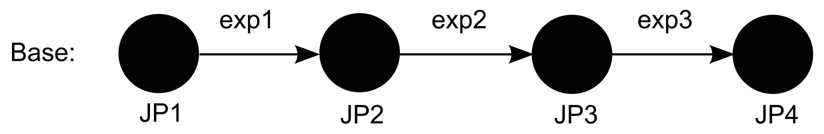

Figure 3.5: Discrete evaluation through join-point stepping [26].

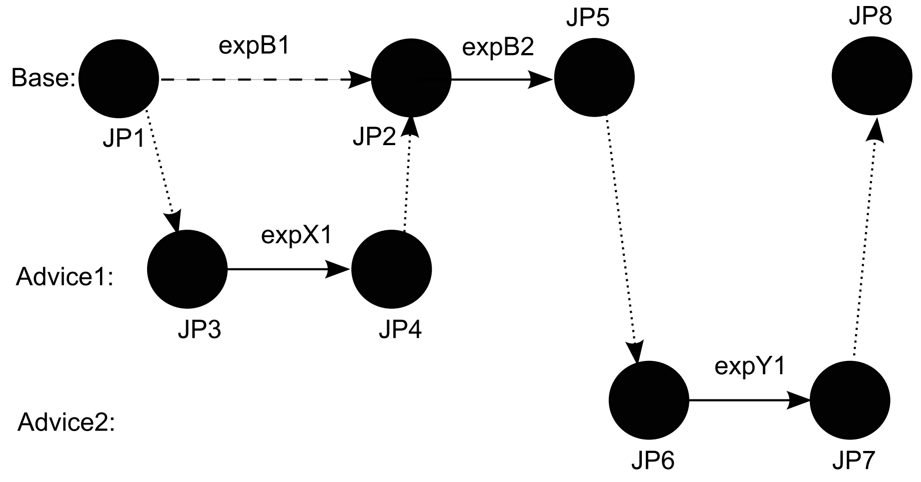

Figure 3.6: Interleaved evaluation of programs.

corresponds to a join point and an entity representing this moment of evaluation must be communicated to the metalevel aspect interpreter.

3.4.2 Woven Execution of Program

Now let us consider a base program B and two pieces of advice: X and Y (in all figures, X and Y are named $Advice1$ and $Advice2$, respectively). Let us see the coordination of the execution between these entities.

$$B = expB1 : expB2 : \epsilon$$
$$X = expX1 : \epsilon$$
$$Y = expY1 : \epsilon$$

We suppose that X and Y do not match the same join points. Figure 3.6 shows the flow of execution between the three entities: B, X and Y. The plain arrows represent normal evaluation steps by the base interpreter. The dotted arrows represent the switch of the evaluation to the metalevel aspect interpreter. Program B is the base program and consists of two atomic expressions $expB1$ and $expB2$. During its evaluation, the base interpreter creates join points JP1, JP2, etc. At join

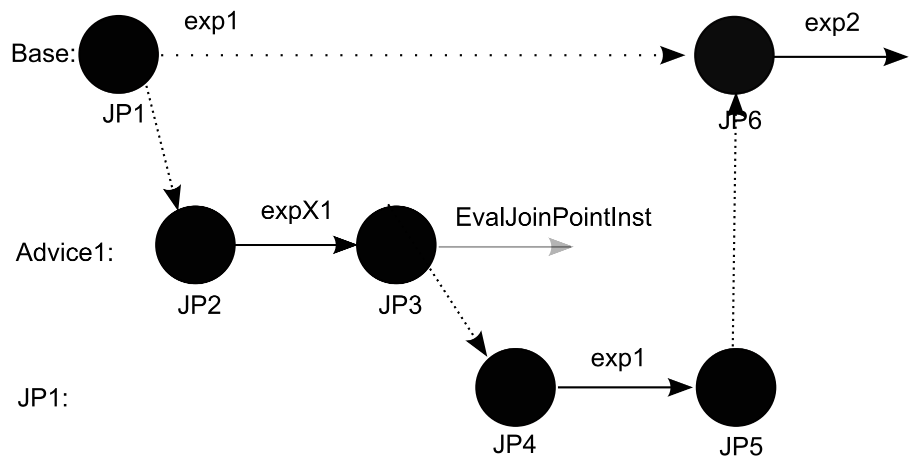

Figure 3.7: Evaluation of Advised Instruction.

point JP1 (before the execution of $expB1$), the metalevel aspect interpreter halts the execution of program B and starts the evaluation of program X. This happened because join point JP1 is matched by a join point selector in the aspect program and, therefore, the metalevel aspect interpreter schedules the execution of the corresponding advice (program X) in the base interpreter after saving the current state of the computation as a continuation pushed on a continuation stack maintained by the metalevel aspect interpreter. The base interpreter thus starts evaluating program X that consists of expression $expX1$. The execution of $expX1$ generates the join point JP2 (before the execution) and JP3 (after the execution), which are not matched by any aspect. When finishing the evaluation of $expX1$, the base interpreter restarts the halted program B (which was stored as a continuation) by generating a join point before $expX2$ is evaluated by restoring the continuation from the top of the continuation stack.

The instruction associated to the join point JP1 is never executed and the execution of program B continues. The execution of program Y happens in exactly the same way as the execution of program X. The only difference is that it is triggered by the join point JP5, which occurs *after executing* $expB2$. X is an *around* advice, which replaces $expB1$, whereas Y is an *after* advice.

3.4.3 Evaluation of Advised Instruction

In the example of Figure 3.6, $expB1$ is never executed. This is because there is no metalevel instructions in the X program to say that $expB1$ needs to be executed. Similarly to proceed in AspectJ, the metalevel instruction $EvalJoinPointIns$ can be used within (in Figure 3.7) advice code to execute the advised instruction. The execution of $EvalJoinPointIns$ is performed at the join point before the instruction. The instruction is then skipped by the base interpreter (since it does not understand it). Instead, the metalevel interpreter creates a new continuation that contains only the instructions of the join point (visualized as program JP1) and activates it. After this, the normal execution of continuations proceeds. $EvalJoinPointInst$ basically works as proceed. If there is no such instruction, the advised instruction is replaced and not further aspect applies. Otherwise, the advised instruction is potentially executed, which may trigger a new advice.

In order to avoid matching JP4, which corresponds to the already matched join point JP1 (the join point before the execution of $exp1$), Metaspin has a mechanism for saving the history of

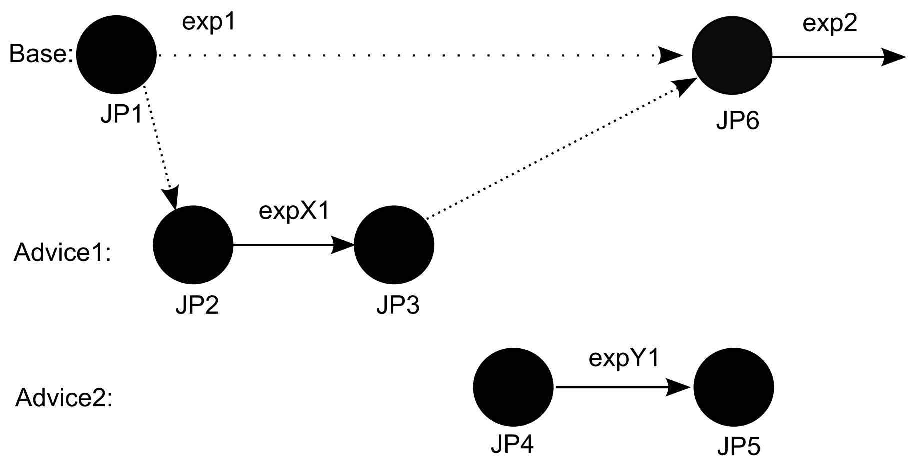

Figure 3.8: X takes priority on Y.

matched join points and recognizing already matched join points. Using this mechanism, JP4 is not matched by X.

3.4.4 Dealing with Aspect Interactions

In the examples of Figures 3.6 and 3.7, the aspects do not have any common join point; their selectors do not have intersections. When aspects match the same join point, the corresponding pieces of advice must be scheduled. In fact, the Advice Binding Metamodel defines the notion of `BindingSelector` which represents the composition of advices when multiple aspects and/or advices share the same join point. This selector defines the method `select(JP,Advices)`, which should be implemented in a way to choose the suitable advice at a shared join point.

3.4.4.1 Interacting Advices at a Join Point without EvalJoinPointInst

Let us consider the example of Figures 3.8 and suppose that X and Y both match the join point JP1. Each (piece of) advice has only one instruction, $expX1$ for X and $expY1$ for Y. The program X is chosen to execute first based on a precedence relation. After the execution of program X, the join point JP3 is created. This means that program Y is no longer considered. This represents exactly the semantics that the activation of program X replaces the execution of expression $exp1$. As a result, if X does not schedule the execution of the join point instruction $exp1$, all other advices are discarded together with the expression $exp1$.

3.4.4.2 Interacting Advices at a Join Point with EvalJoinPointInst

In Figure 3.9, X contains the metalevel instuction $EvalJoinPointIns$ after the expression $expX1$. This leads to execute the second advice (Y) just after the first (X). In fact, after executing $expX1$, the execution of $expB1$ is scheduled. A new join point is created at this moment. However, this join point exactly corresponds to the join point JP2, which has been matched by X and therefore, the new join point is recognized by the history mechanism. As a result, the method `select` of `BindingSelector` returns the second advice Y in order to start its execution. After

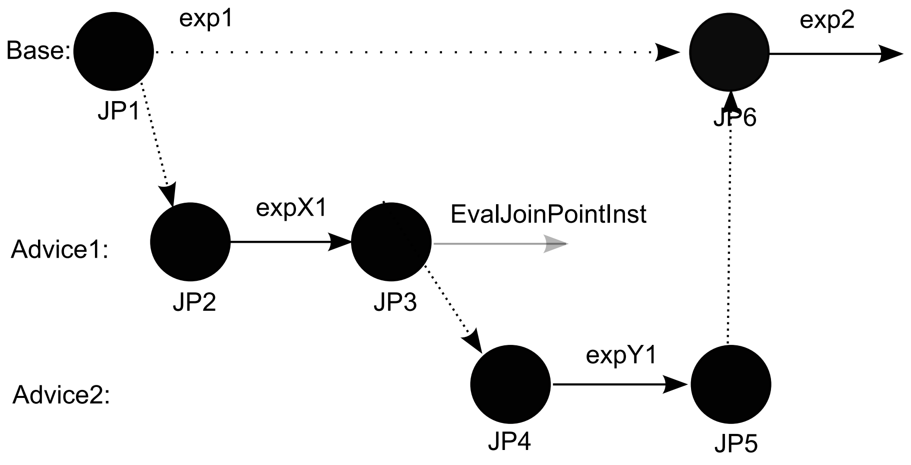

Figure 3.9: Execution of pieces of advice at a shared join point.

executing $expY1$, no advice is returned by the method `select BindingSelector`. Therefore, if Y contains $EvalJoinPointIns$, $exp1$ will be executed, otherwise, the execution of $exp1$ is skipped.

3.5 Pluggable AOP

Pluggable AOP [62] addresses the problem of integrating and using a base language with a set of third-party [1] aspect extensions for that language. It provides a semantical framework to compose, independently developed, dynamic aspect mechanisms. It does not provide support for prototyping AOPL but it defines an architecture for integrating different AOPLs. The base semantics is expressed as an interpreter evaluating expressions. A new AOPL is implemented as expression evaluation transformers. Each mechanism collaborates by delegating or exposing the evaluation of expressions. The base mechanism serves as a terminator and does not delegate the evaluation further.

This framework addresses the problem of integrating and using a base language with a set of third-party extensions. It consists of a framework in which independently developed, dynamic aspect mechanisms can be subject to third party composition and work collaboratively. A *base mechanism* denotes an implementation of the base language semantics, an aspect mechanism denotes an implementation of an aspect extension semantics. The contribution of this work is a general method for implementing the base mechanism and the aspect mechanisms in a way that multiple aspect mechanisms can be subject to third-party composition. This method consists of defining the base mechanism as an expression interpreter. A new aspect mechanism, which is an extension of the base mechanism, is implemented as a *mixin* [21, 27], which extends the base mechanism. The following code implements the mechanisms described in Figure 3.10:

```
public class BMechanism extends Mechanism {
  public void self_eval(){
    // do some thing for the base evaluation
    this.self_eval();
```

1. Third-party extensions means extensions that have been independently developed but have the possibility to be composed together with the base system.

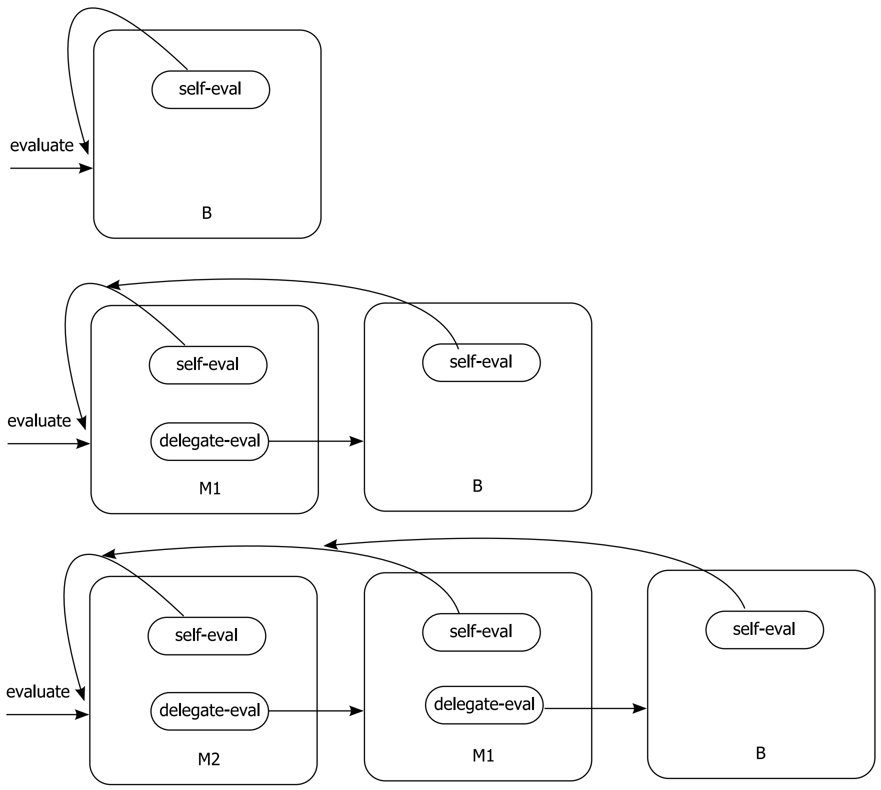

Figure 3.10: The interpreter composition in Pluggable AOP [62]

```
  }
}
public class A1Mechanism extends BMechanism {
  public void self_eval(){
    // do some thing for the aspectual evaluation
    delegate_eval();
  }
  public void delegate_eval(){
    super.self_eval();
  }
}
```

3.6 Summary

This chapter has shown four representative approaches for prototyping and composing AOPLs. For each approach, we have described the provided supports for prototyping and composing AOPLs and we have also identified the provided support for resolving interactions between the composed AOPLs.

In the next chapter, we describe the important criteria for evaluating existing approaches: Simplicity of use, preserving aspect behavior and aspect interactions. We evaluate each of the four approaches against the above criteria and then we study the difference between the use of a transformation or an interpretation approach and the effect on the interactions between composed AOPLs.

Chapter 4

Evaluation

Contents

In this chapter, we evaluate the ability of the four representative proposals described in the previous chapter to prototype and to compose AOPLs. Our evaluation consists of three principal criteria: the first is the *simplicity* of the prototyping, the second is preserving the aspect behavior when prototyping an AOPL and the third is the support for *resolving interactions* when composing AOPLs.

In Section 4.1, we classify whether the aspect mechanisms used in each of the reviewed proposals are implemented using a transformation or an interpretation approach. Then we classify whether the prototyping approach is a *translation* or *sequential*. Section 4.2 describes the effect of using these two approaches on the facility and the flexibility of prototyping. Section 4.3 shows how the translation approach can lead to unexpected behavior while Section 4.4 studies the ability of resolving aspect interactions like *co-advising* and *foreign advising*. Section 4.5 specify the problems that we attack and describe our approach.

4.1 Classification of Existing Work

Let us return to the ideas discussed in 2.2.7. A framework for prototyping AOPL using a transormation approach (define a function T to transform the code), must provide a generic transformation function T that can be specified each time we want to prototype an AOPL. In order to clarify this idea, let us consider, two AspectJ compilers, *ajc* and *abc* then show how abc facilitates the task of defining a new generic transformation in order to make AspectJ more extensible.

ajc The AspectJ compiler *ajc* has a front-end and a back-end. The front-end translates aspects written in AspectJ (`.java` and `.aj` files) to annotated classes in Java bytecode. An aspect is translated to a Java class with the same name; an advice declaration is transformed into a method declaration with the same body. The compiled advice method is also annotated with attributes that store its aspect-specific data (e.g., pointcut declarations). The annotations distinguish aspect classes from other Java classes, and provide pointcut designators for advice methods.

The back-end implements the semantics of the aspect extension (knowing that AspectJ extends Java). It is based on a transformation approach. It takes the classes generated by the frontend, and transform the byte code by including call to the aspects in the corresponding places.

abc The AspectBench Compiler for AspectJ *abc* [15] is another complete implementation of AspectJ. It has beed designed to make it easy to implement both *extensions* and *optimisations* of the core language. Its font-end in the first versions has been built on the *Polyglot* framework for extensible Java compilation (new version of the front-end is based on the *JastAddJ extensible Java compiler*). Its backend is built on the *Soot* framework for code generation, analysis and optimisation.

The AspectJ grammar developed for *abc* is specified as an extension of the Java grammar, and the grammars for extensions are in turn specified as modifications to the AspectJ grammar. The front-end takes `.class` and `.java` (the aspect have `.java` extension) and generate Java AST and an aspect information structure called *AspectInfo*. The weaving in *abc* is performed on the Jimple intermediate representation of Soot. A join point is defined as a single Jimple statement. The backend takes the *AspectInfo* and weave aspects at the level of Jimple representation. A final phase of analyses and optimizations is performed before the bytecode generation. The extension of the AspectJ syntax is based on the feature of Polyglot that allows a new grammar to be specified as a collection of modifications to an existing grammar, where these modifications are given in a separate specification file, not in the original grammar file. AspectJ semantics is extended by introducing new transformation rules supported by Soot. New join point can be added by defining a new factory class that can recognise the relevant statements, and registering it with the global list of join point types.

There are two reasons that make *abc* more extensible than *ajc*:

1. *ajc* performs its transformation at the level of the bytecode and generate optimized bytecode to be executed on the JVM while *abc* performs its transformation at the Jimple level, giving more abstraction to the transformation and making it more extensible than *ajc*.
2. Soot tools for writing new analyses and transformations, such as control flow graph builders, definition/use chains, a fixed-point flow analysis framework, and a method inliner, are useful for implementing extensions such as pointcuts describing specific points in the control flow graph.

We can say that *abc* is characterized by its generic transformation which can be specified in order to extend AspectJ. Using *abc* to extend AspectJ requires the knowledge of all the machinery of abc like Polyglot, Soot, Jimple, etc. while a lightweight extensible implementation of AspectJ suffices, allowing rapid prototyping. In addition, *abc* was not designed to allow the composition of AspectJ with other AOPLs.

Like *abc*, Reflex and XAspects use a transformation approach to implement their aspect mechanisms. Instead of having a generic transformation, Both Reflex and XAspects define an intermediate transformation. Each time a new AOPL has to be defined using Reflex, a mapping from the semantics of the AOPL to the configuration and metaobjects classes, which are the parameter of the code transformation performed by Reflex. XAspects maps AOPL aspects to AspectJ aspects where the transformation is done by an AspectJ weaver.

Pluggable AOP, in contrast to other reviewed proposals, does not use a transformation approach. It represents a semantical framework in which independently developed, dynamic aspect mechanisms can be subject to third-party composition and work collaboratively. Aspect mechanisms are defined as expression evaluation transformers. The mechanisms can be composed like mixin layers in a pipe-and-filter architecture with delegation semantics. Each mechanism collaborates by delegating or exposing the evaluation of expressions. The base mechanism serves as a terminator and does not delegate the evaluation further. In Pluggable AOP, an AOP interpreter involves the base and aspect semantics, and each new aspect mechanism is a wrapper of this interpreter. Pluggable AOP permits the independent development of aspect mechanisms and their

	Transformation	Interpretation
Prototyping	Reflex, XAspects	
Composing	Reflex	Pluggable AOP

Table 4.1: Classification of existing works

composition but the evaluation order of aspect mechanisms is imposed statically by the application sequence of plugins.

4.1.1 Prototyping and Composing AOPLs

Table 4.1 classifies the existing proposals into transformation or interpretation approaches. When a framework for prototyping and composing AOPLs uses a transformation for its aspect mechanism, two alternatives for prototyping and composing AOPLs exist: *translation* and *sequential instrumentation*.

Translation In this approach, aspect programs in different aspect extensions are translated to a common target aspect extensions. Both Reflex and XAspects are using this approach. The impact of this approach is described in the rest of the chapter.

Sequential Instrumentation Different independent mechanisms can be composed sequentially while the output of one mechanism is considered as the input of the next one. This approach is not common because it leads to unexpected result because each mechanism introduces synthetic join points which will be erroneously advised by other mechanisms.

4.2 Simplicity of Prototyping

The main purpose of a framework for prototyping AOPLs is to allow the experimentation of new AOP features in a simple way. The framework must use a *general* aspect mechanism in order to simplify the *mapping* of AOPLs into this mechanism. For example, the aspect mechanism in Reflex is represented as the supported model of partial behavioral reflection while the aspect mechanism used in XAspects is the one of AspectJ. The translation approaches suffer of two problems:

1. The abstraction gap between the semantics of the implemented AOPL and the aspect mechanism of the prototyping framework complicates the construction and the extension of the prototype.
2. Relying on the aspect mechanism of the framework makes it difficult to implement several alternative semantics of the implemented AOPL. For example, the AspectJ Reflex plugin represents `call` designator as a `MsgSend` instance which has, in Reflex, a well-defined semantics and it is difficult to propose alternative for `call` as it was proposed in [18].

These problems existing with the translation approach let us think about designing an extensible aspect mechanism and build AOPLs by extending this mechanism instead of translating different AOPL to an existing one.

4.3 Preserving Aspect Behavior

In the previous section, we described some problems when we rely on a translation approach. This section completes the list of drawbacks by exploring the problems of aspect interactions. We start by showing how the existing work, like Reflex and XAspects, cannot correctly translate aspects from an AOPL to their own representation.

Let us see how Reflex does not properly translate AspectJ aspect to a Reflex configuration class and metaobjects.

```
public class A {
  public void foo () {
    System.out.println("foo");
  }
}
public aspect Logging {
  pointcut p(): call (* *.*());
  void around(): p() {
        log(thisJoinPoint.getSignature().getName());
        proceed();
  }
  void log(String methodName){
    System.out.println("call of "+methodName);
  }
}
public class Main {
  public static void main(String args[]){
    new A().foo();
  }
}
```

Listing 4.1: An AspectJ logging aspect.

Directly using AspectJ, the result of the execution of `Main` in Listings 4.1 is the following:

```
call of foo
foo
```

The aspect of Listing 4.1 is transformed, using the AspectJ plugin of Reflex [95], to a configuration class and a metaobject class shown in Listing 4.2.

The Reflex core takes the two plugin outputs with the base class (`A`).

Now using the generated class, the result of the execution of `Main` is the following:

```
call of foo
call of foo
foo
```

We see that the result of the execution is different from the AspectJ one, which is a problem. The root of the problem is that in Reflex an aspect advises another aspect in the same manner as it advises a base class. In the example, the hooks are inserted in `Logging` as well as in `A`. In Reflex, any class can be instantiated as a metaobject, associated with a hookset, while choosing one of its methods as the advice. In some cases, this causes an infiniteloop.

To generalize the problem, we can say that the translation introduces implementation-specific code into the resulting aspects. The implementation-specific code which is not explicit in the source aspect becomes a part of the resulting aspect and represents unexpected join points. Because the aspect mechanism of the implementation framework cannot distinguish between the synthetic (unexpected) from the genuine (expected) join points, it can erroneously advise the synthetic points and lead to incorrect behavior of the program.

The solution in this case is the quantification of the aspect code, where we could differentiate between the advice code and the specific implementation code as we have seen with the AspectJ plugin. In the `ajc` compiler, the quantification is done using *annotations*: an aspect is translated by the front-end into an annotated Java class (bytecode), and the back-end (weaver) can distinguish, using the annotations, between different parts of code in the aspect to make the right transformation of the byte code. So we can generalize this point to make the first requirement on

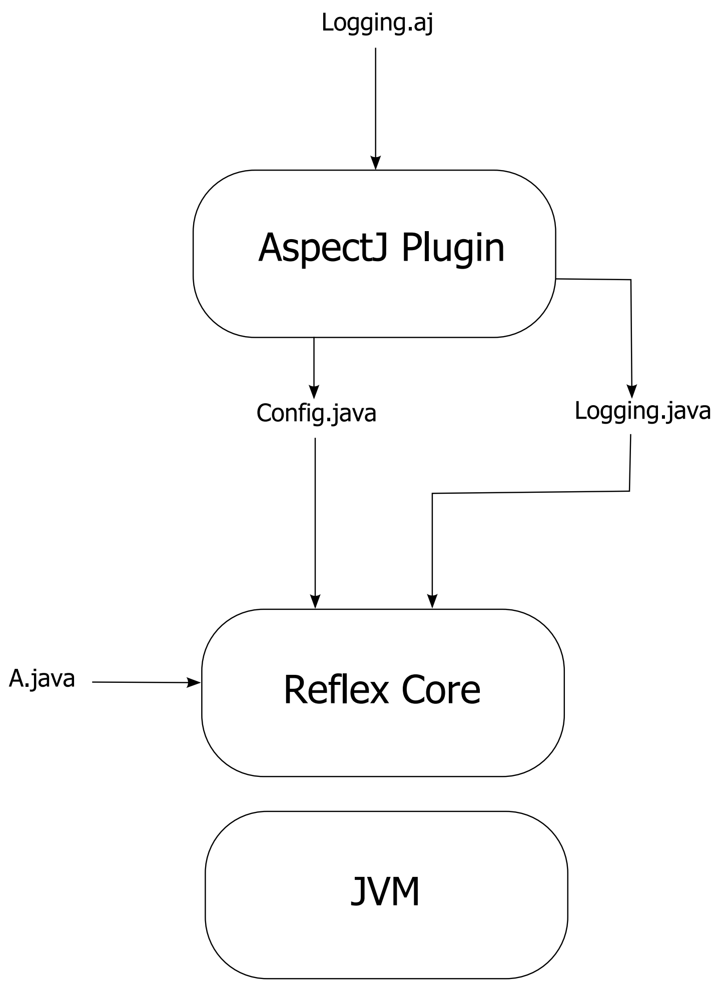

Figure 4.1: The architecture of AspectJ plugin on top of Reflex

```
public class Logging extends MODefinition{
  public void aroundMethod {
  log();
  proceed();
  }
  public void log() {
    System.out.println("call of ..");
  }
}
public void Config extends ReflexConfig {
  public void initReflex() {
  Hookset p = new PrimitiveHookset(MsgSend.class,
                                           AllCS.getInstance(),
                                           AllOP.getInstance());
  BLink theLink = addBLink(theHookset, new Logging());
  theLink.setCall(new CallDescriptor(
        Logging.class.getName(), "aroundMethod", Parameter.CLOSURE)
  );
  }
}
```

Listing 4.2: The Reflex representation of the logging aspect.

the framework to prototype AOPL:

The aspect representation must provide a categorization to identify the implementation-specific code from the code that can be advised

4.4 Aspect Interactions

In our work, we study the design of a framework helping us in prototyping and composing AOPLs. When composing AOPLs, aspect interactions appear as interactions between foreign aspects, i.e., aspects written in different AOPLs. We rely on the work of Kojarski *et al.* [69, 65, 70] to identify the different types of interactions and how to resolve them. This study is very important when we design our framework and, as we will see, there is a strong relationship between these interactions and the representation of aspects. There are two types of aspect interactions:

Co-advising means coordinating pieces of advice selected from aspects in the various AOPLs applying to the same join point.

Foreign advising means the control of the matching join points that occur in foreign aspects (aspects written in different AOPLs).

When composing AOPLs, these interactions must be resolved. We distinguish between two levels of resolution:

Language level requires the language designer to specify the semantics of how aspects in each AOPL interacts with aspects in all the other AOPLs.

Program level requires the aspect programmer to resolve how a concrete set of aspects interact.

We start by explaining both co-advising and foreign advising because a good understanding of the problem that occurs when composing AOPLs is central to the design of our framework.

4.4.1 Co-advising

Co-advising appears when several aspects match the same join point. Generally, co-advising can be resolved in any conceivable way or even arbitrarily because the ordering of aspects does not matter. Practically, however, co-advising is usually an advice scheduling problem that requires the various AOPLs to coordinate the execution of advice code at the same join point.

Let us consider two AOPLs, A_1 and A_2, where we use a transformation approach for each language. The weaver must be scheduled to run one after the other sequentially (sequential instrumentation) advice that is applied later will always wrap around a piece of advice that is applied earlier. This would result in a very restrictive behavior that does not support the flexible ordering needed in general for resolving co-advising.

4.4.1.1 Discussion

In a framework for composing different AOPLs, co-advising interactions should be controlled at both the program and the language levels [70]. A framework should specify default co-advising rules at the language level, and provide means for overriding the default behavior at the program level. For example, the framework could define default advice ordering rules that correspond to the order of the aspects definition.

4.4.2 Foreign advising

In AspectJ, an aspect $aspect_1$ can advise join point in the execution flow of an advice in other aspects, but when composing AOPLs, an aspect advises not only base language classes and, possibily, aspects written in the same AOPLs, but also *foreign* aspects that are written in other AOPLs. The resolution of foreign advising interactions controls how aspects advise foreign aspects. In a composition of COOL and AspectJ, foreign advising controls the weaving of AspectJ advice into COOL coordinators, and the weaving of COOL advice into AspectJ aspects. To understand the problem, let us describe both the AspectJ compiler and COOL compiler.

The AspectJ compiler *ajc* has a front-end and a backend. The front-end translates aspects written in AspectJ (`.java` and `.aj` files) to annotated classes in Java (bytecode). An aspect is translated to a Java class with the same name; an advice declaration is transformed into a method declaration with the same body. The compiled advice method is also annotated with attributes that store its aspect-specific data (e.g., pointcut declarations). The annotations distinguish aspect classes from other Java classes, and provide pointcut designators for advice methods.

The back-end implements the semantics of the aspect extension. The semantics define the meaning of advice weaving in terms of computations. It take the classes generated by the frontend, and transform the byte code by including call to the aspects in the corresponding places.

As we see, the front-end introduces implementation specific points into the bytecode. The semantic of the backend takes in consideration these points when weave aspects. Now, let us describe the COOL compiler [64].

The `Stack` class of Listing 4.3 defines two public methods: `push` and `pop`. An attempt to pop objects off an empty stack or push objects onto a full stack throws an exception. A coordinator in COOL (see Listing 4.4) imposes synchronization over `push` and `pop` methods. The synchronization policy is expressed in COOL using declarations (`mutex`, `selfex`, `condition`), expressions (`requires`), and statements (`on_exit`, `on_entry`). The `selfex` declaration specifies that neither push nor pop may be executed by more than one thread at a time. The `mutex` declaration prohibits `push` and `pop` from being executed concurrently. `full` and `empty` are condition boolean variables. The `requires` expressions further guard `push` and `pop` executions. If the guard is false, a thread suspends, even if the `mutex` and `selfex` conditions are satisfied. The execution resumes when the guard becomes true. The `on_entry` and `on_exit` blocks update the aspect state immediately before and immediately after the execution of an advised method body, respectively.

The front-end of COOL translates the coordinator to a Java class (see Listing 4.5), `StackCoord`, which implements the synchronization logic via special synchronized methods and instance variables. The class provides a pair of `lock_` and `unlock_` methods and an instance variable for every

```
public class Stack {
  private Object[] buf;
  private int ind = 0;
  public Stack(int capacity) {
    buf = new Object[capacity];
  }
  public void push(Object obj){
    buf[ind] = obj;
    ind++;
  }
  public Object pop() {
    Object top = buf[ind-1];
    buf[--ind] = null;
    return top;
  }

}
```

Listing 4.3: A non-synchronized stack

```
coordinator Stack {
  selfex {push, pop};
  mutex {push, pop};
  int len=0;
  condition full=false,empty=true;
  push: requires !full;
  on_exit {
    empty=false;
    len++;
    if(len==buf.length) full=true;
  }
  pop: requires !empty;
  on_entry {len--;}
  on_exit {
    full=false;
    if(len==0) empty=true;
  }
}
```

Listing 4.4: A coordinator in COOL

```
public class StackCoord {
  private boolean empty = true, full= false;
  private List pushState = new Vector(), popState = new Vector();
  private int len = 0;
  public synchronized void lock_push(Stack target) {
    while (!(!full) ||
        isRunByOthers(pushState) ||
        isRunByOthers(popState))
      try { wait(); }
      catch (InterruptedException e) {}
    pushState.add(Thread.currentThread());
  }
  public synchronized void unlock_push(Stack target) {
    pushState.remove(
    Thread.currentThread());
    empty = false;
    len++;
    if (len == target._buf().length)
    full = true;
    notifyAll();
  }
  public synchronized void lock_pop(Stack target) { .. }
  public synchronized void unlock_pop(Stack target) { .. }
  private synchronized boolean isRunByOthers(List methState) {
    return (methState.size() > 0 &&
                      !methState.contains(Thread.currentThread()));
  }

}
```

Listing 4.5: A translated COOL coordinator class

```
public class Stack {
  public Object[] _buf() {return buf;}
  private Object[] buf;
  private int ind = 0;
  private StackCoord _coord;
  public Stack(int capacity) {
    buf = new Object[capacity];
    _coord = new StackCoord();
  }
  public void push(Object obj) {
    _coord.lock_push(this);
    try{
      buf[ind] = obj;
      ind++;
  } finally {_coord.unlock_push(this);}
  }
  public Object pop()
}
```

Listing 4.6: A synchronized bounded stack

method that is advised by the coordinator. Specifically, the synchronization for the `Stack.push` method is realized by `lock_push` and `unlock_push`. Similarly, the synchronization logic for `Stack.pop` is realized by `lock_pop` and `unlock_pop`. At any point of the execution, the `pushState` (`popState`) instance variable stores all threads that are currently executing the `push` (`pop`) method on the coordinated object. The coordinator class also includes all fields of its coordinator. The `lock_` methods implement the semantics for `mutex`, `selfex`, and `requires`, and run `on_entry` blocks. A while loop suspends the execution of the current thread if a guard condition is violated.

The backend transforms the coordinated methods by introducing calls to the coordinator's `lock_` and `unlock_` methods before and after the original body. To ensure invocation of the `unlock_method`, the weaver also introduces a `try finally` block around the original body. The result of the transformed `Stack` can be seen in Listing 4.6.

4.4.2.1 Discussion

We see that both the AspectJ and COOL compilers introduce in the aspect representation (Java class generated by the front-ends) implementation specific code like `lock_` and `unlock_` methods for the coordinator.The problem is that this code can be further erroneously advised by anotheraspect written in another language. This is more explained in 4.4.3.

4.4.3 Composition of AspectJ and COOL

Let us see what will happen if we want to compose an AspectJ aspect with the stack coordinator. Consider the aspect of Listings 4.1: `Logging` logs all join points in a program execution, including join points within executions of the stack coordinator like `lock_` and `unlock_` methods and other implementation specific code, where we found the problem.

Foreign advising is not solvable by merely using a weaver for COOL (AspectJ) to weave the foreign aspects (coordinators), because one language does not recognize the syntax or semantics of the other. Even though the weavers for COOL and AspectJ may both use Java classes as their intermediate representation, applying the COOL (AspectJ) weaver to the Java representation of foreign aspects (coordinators) will not do the job either. This is because the classes embed synthetic code that is generated during the translation to the intermediate representations, e.g., calls to `wait` and `notifyAll` in the coordinator class `StackCoord` of Listing 4.5.

4.4.4 Discussion

Let us discuss co-advising and foreign advising by reasoning in terms of abstract mechanisms described in Chapter 3. An abstract weaving mechanism consists of four subprocesses: **reify**, **match**, **order** and **mix**.

1. Co-advising: requires a proper specifying the meta behavior that coordinates the **match**, **order** and **mix** processes of the individual weavers.
2. Foreign advising: requires a proper representation of foreign aspects correctly, which is the responsibility of the integrated **reify** process.

4.4.5 Requirements

We summarize what we have seen by concluding with some requirements for a good framework for composing AOPLs.

1. The aspect representation must provide a categorization to differentiate implementation specific code from advice code. This categorization should be used by the **reify** process.
2. The framework must provide support for aspect scheduling of each implemented language and support for generic aspect scheduling.

4.5 Contribution

According to the evaluation of existing proposals discussed in this chapter, we deduce that is very difficult to deal with source transformation to implement AOPLs because it requires the deal with a large set of all the machinery related to the compiler world. Studying the foundations of aspect languages can be done by considering what changes have to be made to conventional language interpreters (*base* interpreters) to introduce aspect semantics [46]. In general, writing an interpreter is much less work than writing a back-end. Also it helps for rapidly prototyping, testing and extending the language.

Our approach consists of making an hierarchy for the abstraction levels of aspect mechanisms and describing relations between these levels: *conceptual model*, *semantical model* until the *implementation* level as an interpreter. These relations make it easy the understanding of AOPLs semantics, their prototyping and composition.

We consider the model defined in [63] as our conceptual model. The four subprocesses discribed in this model are at the top of the hierarchy. The second level is the semantical model defined by the CASB [36]. For each subprocess, we describe the semantics according to the CASB.

We reuse the metamodel of AOPL defined in [26] as an Abstract Syntax Tree (AST) while the semantics is implemented according to the conceptual and the semantical models. The concepts that are modeled in the metamodel represent the essential aspect language features and their relations. We obtain a general AOPL that can be the subject of extension and specialization in order to model concrete AOPL. Any aspect language should be defined as a mapping of its own language features to the concepts in the metamodel.

We study the extension of MetaJ according to our approach. MetaJ plays the role of the base interpreter and interacts with the aspect interpreter implementing the AOPL semantics. We achieve a clear separation between the base and the aspect interpreter, while defining the interaction between the two interpreters. The aspect interpreter explicitly shows the separation between the four subprocesses. Each subprocess is implemented according to the goal of facilitating the understanding and the extension of the interpreter. This separation facilitates the understanding and the extension of the interpreter and is also a step towards third-party composition of AOPLs. Third-party composition means that different AOPLs, which were independently implemented, can be assembled without modifying the individual implementation. In such a framework approach, the concrete language features of particular aspect languages can be partially described as specializations of the concepts described in the common metamodel. The framework approach also guarantees that all aspect languages are described with respect to the framework. As proof of concepts, we realize a prototype of lightweight version of AspectJ by implementing several pointcut designator as an extension of an essential part of the metamodel, *JoinPointSelector*.

After implementing the framework for a part of Java, we apply the same design but for the whole of Java by reusing a mechanism called two-step weaving which simplifies the construction of our framework on top of the whole of Java without dealing with problems of efficiency and performance but focusing on the design and the extensibility of the interpreter. We name this framework CALI for *Common Aspect Language Interpreter*.

Part II

Contributions

Table of Contents

Chapter 5

Modifying an Existing Java Interpreter: MetaJ

Contents

A good way of understanding the foundations of AOPLs on top of Java is to take a Java interpreter and consider what changes we need to make to add/introduce aspects. In this chapter, we describe our experience with MetaJ, a Java interpreter written in Java [39, 40], as our base interpreter. We show the modifications (instrumentation and extension) to get an interpreter for AOPLs. We will keep these changes modular as much as possible by separating the base and the aspect interpreter. The aspect interpreter will, in turn, be split into two parts: one called `Platform`, for the common features and another for features specific to a given AOPL. The specific part should provide an *interface* to be called from the common one. The structure of this chapter is as follows:

Section 5.1 shows the modifications that we made on MetaJ itself to support an AOP extension then displays how the AOP extension is performed as an independent component. Section 5.2 presents the aspect instantiation mechanism as an extension of the class instantiation mechanism of MetaJ. Section 5.3 reviews the previous section in order to define how to implement a concrete AOPL by extending the aspect interpreter. Section 5.4 instantiates our approach and describes an implementation of a lightweight version of AspectJ. Section 5.5 concludes the chapter.

5.1 Modifying MetaJ to support AOP

In this section, we consider the modifications that we need to make on MetaJ in order to support AOP. The modifications are done according to the semantics of the CASB (see Section 2.2.6.2). In the CASB, the base interpretation is described by the following rule:

$$(i : C, \Sigma) \rightarrow^b (C', \Sigma')$$

In a Java setting we will talk about expressions, instances of the class `Exp`, instead of instructions. The reduction rule is then implemented by a method `void eval(Environment)` of `Exp`, where the class `Environment` explicitly implements a part of Σ and the other part is implicitly implemented in the runtime of the interpreter.

The challenge now is to find how to represent the function ψ of the CASB, which should take an `Exp` object as a parameter. In the CASB, aspects are introduced by complementing the base rule with a number of additional rules:

$$\textsc{NoAdvice}\ \frac{\psi(i) = \epsilon \qquad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma, P) \rightarrow (C', \Sigma', P)}$$

$$\textsc{Around}\ \frac{\psi(i) = \phi : \Phi}{(i : C, \Sigma, P) \rightarrow (\phi : \texttt{pop} : C, \Sigma, (\Phi : [\bar{i}]) : P)}$$

$$\textsc{Advice}\ \frac{\phi(\Sigma) = a}{(\phi : C, \Sigma, P) \rightarrow (a : C, \Sigma, P)}$$

The introduction of the AOP semantics consists of applying ψ to each instruction before performing the base evaluation. When $\psi(i) = \epsilon$, the base rule is applied and the instruction is normally evaluated. Otherwise, the interpreter has to evaluate, according to the rules, the first ϕ corresponding to the aspects that statically matched the instruction. A direct implementation of these rules consists of applying a `functionPsi` on each instance of the class `Exp` *before* executing the method `eval`. The method `functionPsi` must determine the list of aspects that *statically* match the current expression. The word *statically* signifies that the matching is based on the static information attached to the instance of `Exp` like method name, static type, etc. and not on the information given in the instance of `Environment`. We can deduce that the representation of an aspect must provide a `staticTest` method which is invoked by `functionPsi` in order to determine if the aspect statically matches the current expression. The return type of the method `staticTest` should provide a method that perform an additional (*dynamic*) test on the current context (corresponds to $\phi(\Sigma) = a$) in order to return the corresponding advice a. This idea is reviewed in the body of this chapter.

Requirement 1 *For any AOPL implementation, the aspect interface must contain a `staticTest` method, which should be invoked by the function `functionPsi` of the aspect interpreter.*

5.1.1 Interpreter-based two-step weaving

As we said in the state of the art, in AspectJ, weaving is static. The weaver implements the semantics statically by transforming the base and aspect classes. Another possibility would be to write an interpreter from scratch. Our approach is different. We start from an existing Java interpreter and weave aspects in two steps instead of one:

1. The first step consists of reifying the evaluation of an expression within a given environment, using AspectJ. A join point is then a pair (expression, environment).
2. The second step consists of applying the implementation of the CASB functions on the reifed expression.

The introduction of these two steps is done by instrumenting the base interpreter $int(p, d)$ (where p and d represents program and data respectively), which becomes $int_i(p, d)$.

5.1.2 Implementation

We see that the aspect interpreter can be considered also as an aspect that is applied on the base interpreter in order to add AOP capabilities. This aspect contains a pointcut that matches the execution of all the types of expression and an `around` advice, which will `proceed` if there is no aspect statically matching the instruction or start the evaluation of a `test`ϕ instruction otherwise. The evaluation of this instruction performs an additional dynamic test which, if successes, launches the evaluation of the piece of advice. The aspect interpreter must have a `proceedStack` in order to represent the proceed stack P of the CASB. Implementing the AOP semantics using an aspect keeps the modifications on the base interpreter modular.

```
public aspect Platform {
  pointcut reify(Environment env, Exp exp) :
                    execution(Data (!(aop..*)).eval(..) )
                    && args(env)
                    && this(exp);
  Data around(Environment env, Exp exp) : reify(env,exp) { .. }
  public static List<Phi> match (Exp exp){ .. }
  public static List<Phi> order (Exp exp){ .. }
  public static List<Phi> mix (Exp exp, Environment env, Phi phi){
      .. }
}
```

Listing 5.1: The aspect interpreter as an aspect `Platform`

Requirement 2 *The implementation of the aspect interpreter must be modular (separated from the base interpreter) to increase the understanding of AOP foundations as well as the extensibility of AOP implementation.*

The aspect interpreter is then represented by the aspect `Platform` (see Listing 5.1). Proposition 3 states the condition of the advice return-type:

Requirement 3 *Since the return type of the expression evaluation by the base interpreter is `Data`, the advice of `Platform` must return the same type (see Listing 5.1).*

5.1.3 Join-Point Model

We consider that the execution of the method `eval(Environment)` of any subclass of the class `Exp` is a join point. The static part of this join point is the instance of `Exp` and the dynamic part is the instance of `Environment` given as the parameter. The pointcut of the aspect `Platform` should test these two parts to decide to match or not the join point. An intuitive question naturally arises here:

Is the list of expressions Exp provided by MetaJ sufficient to express at least the known join points?

To answer this question, we must study how to support method-call and method-execution join points. Below, we review how MetaJ implements call and execution of methods then we discuss how it is possible to represent the corresponding join points.

5.1.3.1 Original MetaJ

Listing 5.2 shows the class `ExpMethod` which is the implementation of a method-call in MetaJ. An instance of this class represents the AST of a method-call. It has three attributes: `exp` is the *receiver* expression on which we call the method `methodId` with the list of arguments `args`. This is a static view of a method call in the source code. The evaluation of an instance of `ExpMethod` (execution of its `eval` method) consists of:

– Evaluating the receiver expression,
– Evaluating the list of arguments and creating an `argE` environment,
– Looking up the method `methodId` and getting the corresponding `Method` object,
– Finally calling `apply` of the proper method with, as parameters, the proper environment and receiver.

Listing 5.3 shows the class `Method` which is the representation of a method in MetaJ. It has an instance variable of type `Exp` to represent the *body* of the method and an instance variable of type `StringList` to represent the formal parameter names. The execution of `apply(Environment, Instance)` associates the names of the parameters (in addition to `this` object which is associated

```java
public class ExpMethod extends Exp {
  private Exp exp;
  private String methodId;
  private ExpList args;

  ExpMethod(Exp exp, String methodId, ExpList args) {
    this.exp = exp;
    this.methodId = methodId;
    this.args = args;
  }

  Data eval(Environment localE) {
    // evaluate the lhs (receiver)
    Instance i = (Instance) this.exp.eval(localE).read();
    // evaluate the arguments to get a new local environment
    Environment argsE = new Environment(null, null, null);
    this.args.eval(localE, argsE);
    // lookup and apply method
    Method m = i.lookupMethod(this.methodId);
    return m.apply(argsE, i);
  }

  public String toString() {
    return "(" + this.exp.toString() + ")." + this.methodId
            + "(" + this.args.toString() + ")";
  }
}
```

Listing 5.2: The representation of a method call in MetaJ

```java
public class Method {
  private StringList args;
  private Exp body;

  Method(StringList args, Exp body) {
    this.args = args;
    this.body = body;
  }

  Data apply(Environment argsE, Instance o) {
    // name each argument
    argsE.zipWith(this.args);
    // add the first implicit argument
    argsE.add("this", new Data(o));
    // eval the body definition of the method
    return this.body.eval(argsE);
  }

  public String toString() {
    return "(" + this.args.toString() + ") {\n\t\t"
             + this.body.toString() + "; \n\t}";
  }
}
```

Listing 5.3: The representation of a method in MetaJ

with the name “this”) with the argument list then evaluates the expression of the method (body) with this association list.

5.1.3.2 Discussion

When capturing a method-call join point, the receiver object should have been evaluated and exist in the environment. It is not the case in MetaJ, because when evaluating an instance of `ExpMethod`, the receiver expression has not yet been evaluated and is not yet in the environment. For this reason, an intermediate evaluation step has to be inserted between the evaluation of the `ExpMethod` instance and the lookup of the method.

Capturing a method-execution join point consists of capturing the evaluation of the method body. This could be done by capturing the call of the method `Method.apply` within the class `ExpMethod` or capturing the call of the method `Exp.eval` within the body of `Method.apply`. The first choice is not conform to what we have mentionned at the head of this section about supporting the CASB semantics by capturing the call of the method `Exp.eval`. The second choice does not allow the differentiation between the expression of the method body from another expression type (see Listing 5.1).

5.1.3.3 Modification on MetaJ

Method call The modification on MetaJ consists of adding an intermediate evaluation step between the `ExpMethod` evaluation and the method lookup. Listings 5.4 and 5.5 show the modification on `ExpMethod` and the new expression type `ExpMethodCall`. The method `eval(Environment)` of `ExpMethod` is modified by adding a `target` variable in the arguments environment (`argsE`) then creating an instance of `ExpMethodCall` and evaluating it with the new `argsE` containing the target instance. The two variables `exp` and `args` of the class `ExpMethod` is also passed to the

```
public class ExpMethod extends Exp {
  ...
  Data eval(Environment localE) {
    // evaluate the lhs (receiver)
    Instance i = (Instance) this.exp.eval(localE).read();
    // evaluate the arguments to get a new local environment
    Environment argsE = new Environment(null, null, null);
    this.args.eval(localE, argsE);
    // add the receiver to the environment
    argsE.add("target", new Data(i));
    // create an instance of ExpMethodCall
    ExpMethodCall methodCall = new ExpMethodCall(exp, methodId,
       args);
    // evaluate the instance of ExpMethodCall
    return methodCall.eval(argsE);
  }
}
```

Listing 5.4: The modified class `ExpMethod`

```
public class ExpMethodCall extends Exp {
  private Exp exp;
  private String methodId;
  private ExpList args;

  ExpMethodCall(Exp exp, String methodId, ExpList args) {
    this.exp = exp;
    this.methodId = methodId;
    this.args = args;
  }

  Data eval(Environment localE) {
    // lookup for the receiver instance
    Instance thisObj = ((Instance)localE.lookup("target").value);
    // lookup and apply method
    Method m = thisObj.lookupMethod(this.methodId);
    return m.apply(localE, thisObj);
  }
}
```

Listing 5.5: The intermediate representation of a method call

instance of `ExpMethodCall` in order to be used when matching method-call join points (call of `ExpMethodCall.eval`).

Method execution In order to identify the method-execution join point, we just create a new type of expression called `ExpBodyMethod`. In the `apply` method, we create an instance of this class to encapsulate the body of the method then we evaluate it besides evaluating the body of the method.

5.1.3.4 Possible join points

After making the above changes on the base interpreter, we can capture, using the pointcut of Listing 5.1, the execution of all expression types of interest and decide to match or not the corresponding join point depending on the existing aspects. This leads to a very rich join-point model. For example, the instantiation join point is captured when testing if the expression being evaluated is of type `ExpNew`. Also a `if-then-else` join point can be captured by matching the evaluation of `ExpIfThenElse` instances. This is the role of a pointcut to test the current expression (by the pointcut of Listing 5.1) and the corresponding `Environment` and decide to match or not the join point. The main possible join point are:

– method call: `ExpMethodCall`
– method execution: `ExpMethod`
– field set: `ExpAssign`
– field get: `ExpData`

More details about pointcuts will be explained later in this chapter.

5.1.4 Processes of Aspect Interpreter

Now, we are going to identify the mechanisms that must be in the aspect interpreter sketched in Listing 5.1 and to make the collaboration between the mechanisms existing in a weaver (see Section 2.2). This work help us to properly identify what is common and what is specific between different aspect interpreters.

In Section 2.2, the reify process was explained in terms of a compile-time weaver. Within a runtime weaver, the reify process does not produce a list of shadows, which correspond to program points, but directly produces join points. It is implemented by the pointcut `pointcut reify(Environment env, Exp exp): execution(Data (!(aop..*)).eval(..))`, where the aspect interpreter accesses the expression being evaluated `exp` in the environment `env`. This mechanism should be common to all aspect interpreters.

The **match** process of a weaver takes a *shadow*, looks for all the aspects that match this shadow, and returns a list of *residues*. For the interpreter, it is very similar: the **match** process takes the expression captured by the pointcut and returns a list of `Phi` expressions. The **match** process of the interpreter implements the application of the function ψ, which returns a list of functions ϕ. This process should also be common between different aspect interpreters while the semantics of matching an expression by an aspect and the returned list of expressions `Phi` are specific to each language.

In terms of the CASB, the **order** process order the functions ϕ, which are implemented as residues in a compile-time weaver and here as `Phi` expressions.

The **mix** process transforms the *shadow*s in the weaver while it launches the interpretation of the first `Phi` resulting from the **order** process in the aspect interpreter.

The identification of the 4 subprocesses in the aspect interpreter leads us to define what should be common between different aspect interpreters and what should be specific.

Requirement 4 *An aspect interpreter contains two parts: one is shared with other aspect interpreters and one is specific to each implemented AOPL. The mechanisms of matching an expression is specific to each aspect interpreter.*

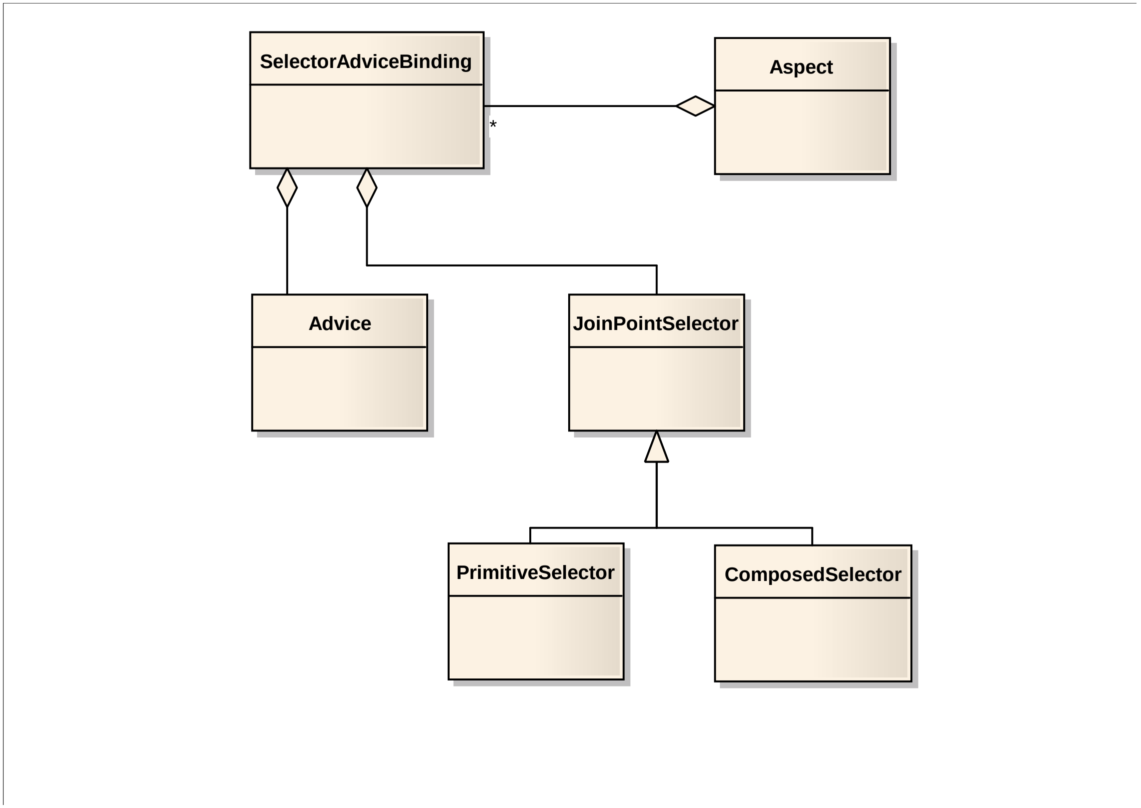

Figure 5.1: The class diagram representing the abstract aspect language

```java
public abstract class JoinPointSelector {
  public abstract boolean staticTest(Exp exp);

  public abstract boolean dynamicTest(Environment env);
}
```

Listing 5.6: The `JoinPointSelector` class

5.1.5 Implementing an Abstract Aspect-Oriented Language

The metamodel of AOPLs represents an *abstract* AOPL (see Figure 5.1). In this part, we are going to give the semantics of this abstract language by implementing it as an interpreter by respecting the specification given in Propositions 1, 2 and 4. This abstract language should be used to implement concrete AOPLs. The common semantics will be implemented as a part of this interpreter while the specific semantics will be implemented when implementing concrete AOPLs.

5.1.5.1 Join-Point Selector

The role of a join-point selector is to select or not a join point. As we mentioned before, a join point always has both a *static part* and a *dynamic part*. The join-point model of AspectJ conforms to this notion [55].

For these reasons, we implement the notion of join-point selector as a class `JoinPointSelector` (see Listing 5.6). This class has two methods: `staticTest(Exp)` for matching the static part of a join point and `dynamicTest(Exp)` for matching its dynamic part. Figure 5.1 shows the class

```
public class Advice extends ExpS { }
```

Listing 5.7: The class `Advice` implementing an advice

```
public class ExpProceed extends Exp {
  public Data eval(Environment localE) {
    Exp exp = (Exp)ProceedStack.pop();
    Data data = exp.eval(localE);
    ProceedStack.push(exp);
  return data;
}
```

Listing 5.8: The implementation of the expression `proceed`

diagram representing the join-point selectors in the abstract AOPL. A `JoinPointSelector` can be either primitive or composed according to the *Composite* design pattern [49]. Composed selectors are combined via the usual boolean operators.

5.1.5.2 Advice

An advice possesses a statement (advice body) to be executed when matching a join point. In the general case, it is not necessary that this statement be a statement of the base language. In AspectJ, an advice body is a Java statement except that it may contain the specific expression `proceed` and a reference to the current join point (`thisJoinPoint`) but we suppose that the advice is more general. Listing 5.7 shows the class `Advice`, which inherits from `ExpS`. This last class represents a statement in MetaJ.

5.1.5.3 Proceed

The specific expression `proceed` is implemented as an expression class `ExpProceed` (see Listing 5.8). The evaluation of this expression implements the PROCEED rule described in Section 5.1. The execution of the method `eval` access a proceed stack implemented by the class `ProceedStack`, pops the expression at the top of the stack and returns result of its evaluation. The expression is re-pushed again in the stack, according to the rule PROCEED, in order to be reused by the incoming aspects matching the current join point and containing the expression `proceed`.

5.1.5.4 The class `Phi`

An instance of the class `Phi` (Listing 5.9) represents a function ϕ in the CASB. According to the semantics described above, the evaluation of this instance must return the advice if the evaluation of the dynamic test is positive (the residue matches the environment) and `proceed` if the result is negative. For this reason, an instance `Phi` encapsulates an object, a selector/advice binding, which contains the dynamic test and the advice.

5.1.5.5 Selector/Advice Binding

A selector/advice binding associates a join-point selector and an advice. Listing 5.10 shows the implementation of selector/advice bindings by the class `SelectorAdviceBinding`.

Again, the methods `staticTest` and `dynamicTest` correspond to the two phases of matching. The test itself is the responsibility of the join-point selector. During the second phase, in case the join point is selected, the advice is returned.

```java
public class Phi extends Exp {
  SelectorAdviceBinding sadb;

  public Phi(SelectorAdviceBinding sadb) {
    this.sadb = sadb;
  }

  public Data eval(Environment localE) {
    Advice result = sadb.dynamicTest(localE);
    if (result == null)
      return new ExpProceed().eval(localE);
    else
      return result.eval(localE);
    }
}
```

Listing 5.9: The `Phi` class

```java
public class SelectorAdviceBinding {
  protected Advice advice;
  protected JoinPointSelector jps;

  public boolean staticTest(Exp exp) {
    return this.jps.staticTest(exp);
  }

  public Advice dynamicTest(Environment env) {
    if (jps.dynamicTest(env)) {
      return this.advice;
    } else
    return null;
  }
}
```

Listing 5.10: The `SelectorAdviceBinding` class

```java
public interface Aspect {
  public List<Phi> staticTest(Exp exp);
}
```

Listing 5.11: The `Aspect` interface

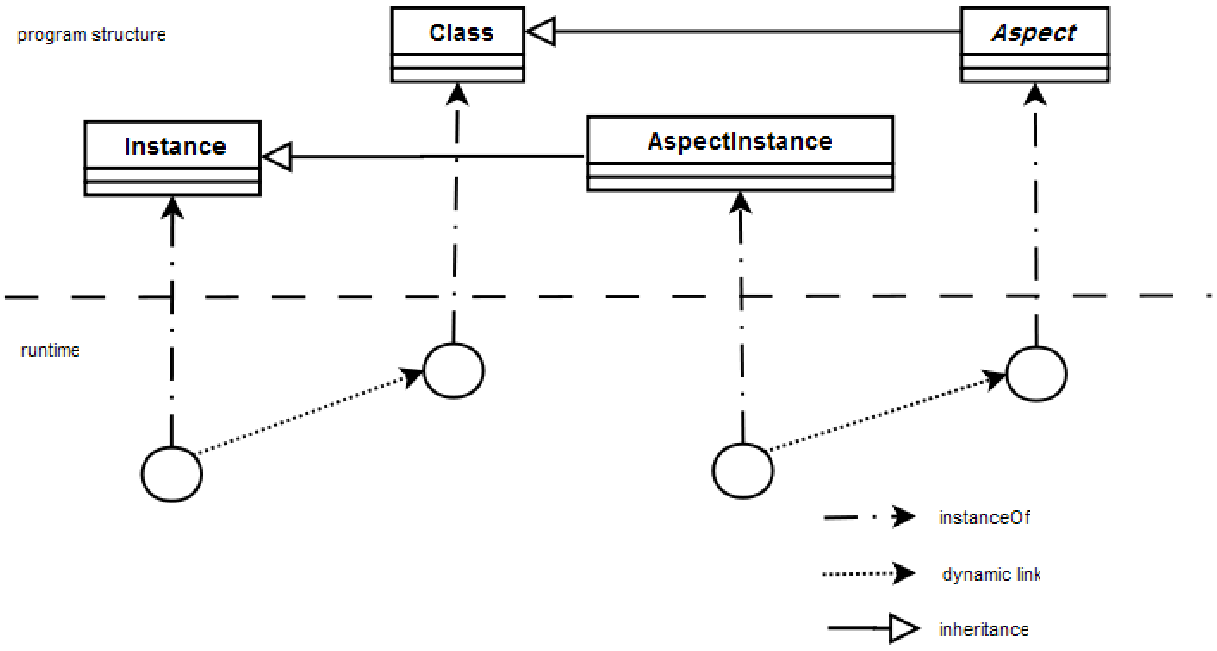

Figure 5.2: The description of the link between instances and definitions (aspects and classes)

5.1.5.6 Aspect

According to Proposition 1, the aspect must provide, in its interface, the method `staticTest`. Listing 5.11 shows the representation of an aspect as an interface containing the method `staticTest(Exp)`. This method should return a list of instances of `Phi`. In addition, the aspect should have a list of selector/advice bindings. The *mechanism* implementing how the aspect calculates the list of instances of `Phi` using its bindings is specific to each AOPL.

5.2 Aspect instances

5.2.1 Link between aspect instances and aspect definition

In MetaJ, an instance of the class `Class` represents the definition of a class while an instance of the class `Instance` represents an instance of the class and is linked to the corresponding `Class` instance (see Figure 5.2). The aspect instances and definitions should be linked in the same manner. In addition, an aspect could define methods and variables in the same manner than a normal class (this is for example the case of AspectJ). For this reason, we change the implementation of an aspect from the interface `Aspect` to the abstract class `Aspect`, which inherits from the class `Class` of MetaJ and defines the method `staticTest`. Listing 5.12 shows this new implementation.

Similarly to the pattern Aspect/Class, we define the class `AspectInstance` (see Listing 5.13) as a subclass of `Instance`.

```java
public abstract class Aspect extends Class {
  public abstract List<Phi> staticTest(Exp exp);
}
```

Listing 5.12: The abstract class `Aspect`

```java
public class AspectInstance extends Instance {
  AspectInstance(Aspect instanceLink, DataList dataList) {
    super(instanceLink,dataList);
  }
}
```

Listing 5.13: The `AspectInstance` class

The class `AspectInstance` inherits the field `instanceLink`, which is used to link its instances to the corresponding aspect definitions. The access to an aspect definition from an aspect instance is done by this *dynamic link* (see Figure 5.2).

An aspect deployment mechanism defines when an aspect is instantiated and used in the program. There are two types of aspect deployment mechanisms: *static* and *dynamic* deployment. However, when an aspect is instantiated and deployed, its *scope* must be precisely specified. In the following, we describe these two deployments types as well as their possible *scoping strategies*.

5.2.2 Static Deployment

In this type of deployment, the parts of the program that are affected by the aspect are always affected, from the start to the end of the execution [89]. The possible scopes found for this type of deployment are **issingleton** (the default strategy in AspectJ), **perthis**, **pertarget** and **percflow**. For the sake of simplicity, we only consider the first one (**issingleton**) where a single aspect instance has to be deployed in a specific environment. In Listing 5.14, the attribute `aspectEnv` is used to store the aspect instances.

5.2.3 Dynamic Deployment

In this type of deployment, the aspect is applied or not depending on the program being in a certain scope. To support a *dynamic deployment*, the language must provide two expressions for explicit (un)deployment.

Let us consider the example of Listing 5.15. The aspect `MyAspect` contains a pointcut matching the call of `foo`. The role of the instruction `deploy` is to instantiate and deploy the aspect `MyAspect`. But with this type of deployment, we can imagine different types of aspect scope (and then different semantics of `deploy`) like having a lexical scope over a specific statement, a dynamic scope or a global scope [90].

Global scope The implementation of global scope with dynamic deployement is very similar to the implementation of static deployment. When an aspect is instantiated and deployed, it should be added to the aspect environment which can be the one described in Listing 5.14. The semantics of `deploy` consists of simply adding the aspect instance to the attribute `aspEnv` of the aspect `Platform`.

Lexical scope An example of lexical scoping is program text-based pointcuts of AspectJ such as `within` and `withincode`. Since the lexical deployment cannot be done dynamically [90], this strategy is not supported here.

Dynamic scope This imposes to take care about the aspect instances and their location (environment).

```
public aspect Platform {
  Environment aspectEnv;
  static List<AspectInstance> getAspects() {
    return ..;
  }

  static List<Phi> functionPsi(Environment env, Exp exp) {
    List<Phi> list = new ArrayList<Phi>();
        for (AspectInstance anAspect : getAspects()) {
      list.addAll(((Aspect)anAspect.getInstanceLink()).staticTest(
         exp));
        }
      return list;
  }

  pointcut reify(Environment env, Exp exp) :
                    execution(Data (!(aop..*)).eval(..) )
                    && args(env)
                    && this(exp);

  Data around(Environment env, Exp exp) : reify(env,exp) {
        // match subprocess
        List<Phi> phiList= match(env, exp);
        if (phiList.size() == 0)
          return proceed(env, exp);
        else {
          // order subprocess
          order(phiList);
          // mix subprocess
          return mix(exp, env, phiList);
        }
  }

  public static List<Phi> match(Environment env, Exp exp) {
    return functionPsi(env, exp);
  }

  public static void order(List<Phi> list) { .. }

  public static Data mix(Exp exp, Environment env, List<Phi>
     phiList) {
    // prepare the proceed stack
    Proceed.proceedstack.push(exp);
        for (Phi phi: phiList) {
      Proceed.proceedstack.push(phi);
    }
        return phiList.get(0).eval(env);
  }
}
```

Listing 5.14: Platform with static deployment

```
MyClass c = new MyClass();
deploy(MyAspect)
{
  c.foo();
}
c.foo();
```

Listing 5.15: An example dynamic aspect deployment

```
public aspect Platform {
  ..
  static List<Phi> functionPsi(Environment env, Exp exp) {
    List<Phi> list = new ArrayList<Phi>();
      for (AspectInstance anAspect : getAspects(env)) {
        list.addAll(((Aspect)anAspect.getInstanceLink()).staticTest
            (exp));
      }
    return list;
  }
}
```

Listing 5.16: Platform with dynamic deployment

Supporting this type of scoping strategy consists of two parts. The first one is the modification of the method `getAspects` in the aspect `Platform` in order to return the aspect instances existing in the environment in which the captured expression is evaluated (parameter of the method `eval`). Listing 5.16 shows these changes. The second is the implemenation of `deploy` as an expression class `ExpDeploy`. This class contains two variables: `statement` and `aspectExp` (see Listing 5.17). This class represents a deployment of `aspectExp` over the statement represented by `statement`. Evaluating an expression of type `ExpDeploy` consists of (see Listing 5.17):

- Creating the aspect instance relative to `aspectExp`.
- Adding this instance to the environment in which this expression is evaluated (line 8).
- Launching the evaluation of the statement (line 9). Because the environment will be used to evaluate all statement sub-expressions, the aspect instance is *propagated* and will be available during the evaluation of all these sub-expressions hence the scope is dynamic.
- Removing the aspect instance from the environment once the evaluation of the statement is finished (line 11).

5.2.4 Collaboration between subprocesses

Listing 5.14 details the interpretation of a join point (*exp*, *env*). The flow of interpretation and the collaboration between the 4 subprocesses are the following:

- The **reify** pointcut captures the execution of an expression *exp* in an environment*env*.
- The advice takes the expression and the environment and launches the execution of the subprocesses **match**, **order** and **mix**.
- The **match** subprocess (represented by the static method `match`) takes the expression and returns a list of instances of `Phi` corresponding to the aspects that statically match the expression.
- The **order** subprocess (represented by the static method `order`) orders the result of the **match** subprocess. The ordering depends on the semantics of the desired aspect-oriented language. For example, when implementing AspectJ, we can imagine that this method takes

```java
public class ExpDeploy extends Exp {
  protected ExpS statement;
  protected Exp aspectExp;
  public Data eval(Environment localE){
    // create an aspect instance
    AspectInstance asp = (AspectInstance) aspectExp.eval(localE).
       read();
    // put the aspect instance in the aspects environment
    localE.add("MyAspect", new Data(asp));
    Data data = statement.eval(localE);
    // remove the aspect from the aspect environment
    localE.remove("MyAspect");
    return data;
  }
}
```

Listing 5.17: The deployment expression `ExpDeploy`

the `declare precedence` statements into account to order the aspects (the corresponding instances of `Phi`).

- The **mix** subprocess initializes the proceed stack according to the CASB semantics then starts the execution of the first instance of `Phi`.

5.3 Generalization

In this section, we overview what we did in this chapter in order to define the methodology for implementing concrete AOPLs using our approach.

5.3.1 Separation between base and aspect interpreter

The separation between the base and the aspect interpreter is very effective in order to improve the understanding and the extensibility of each interpreter. The general architecture shown in Figure 5.3 displays each interpreter as an independent component and displays the interaction between the two interpreters. The interaction between the two components consists of two types of messages:

- A join point emitted by the base interpreter at each step of expression evaluation.
- Data resulting from the evaluation of the join point by the aspect interpreter with the existing aspects.

Advice of the Aspect-Oriented language can be written or not in part in the base language. The aspect language can be an extension of the base language or not. As an architecture, there are three possibilities:

1. The aspect interpreter is a full extension of the base interpreter and the advice actions are interpreted in the aspect interpreter. The two interpreters share the object *heap*.
2. The aspect interpreter delegates the evaluation of the base expressions (in the advice) to the base one. This case is only an optimization of the first one.
3. The aspect interpreter has nothing to do with the base interpreter. This is the case where the aspect language is independent from the base one (the advice body does not contains any base level expression).

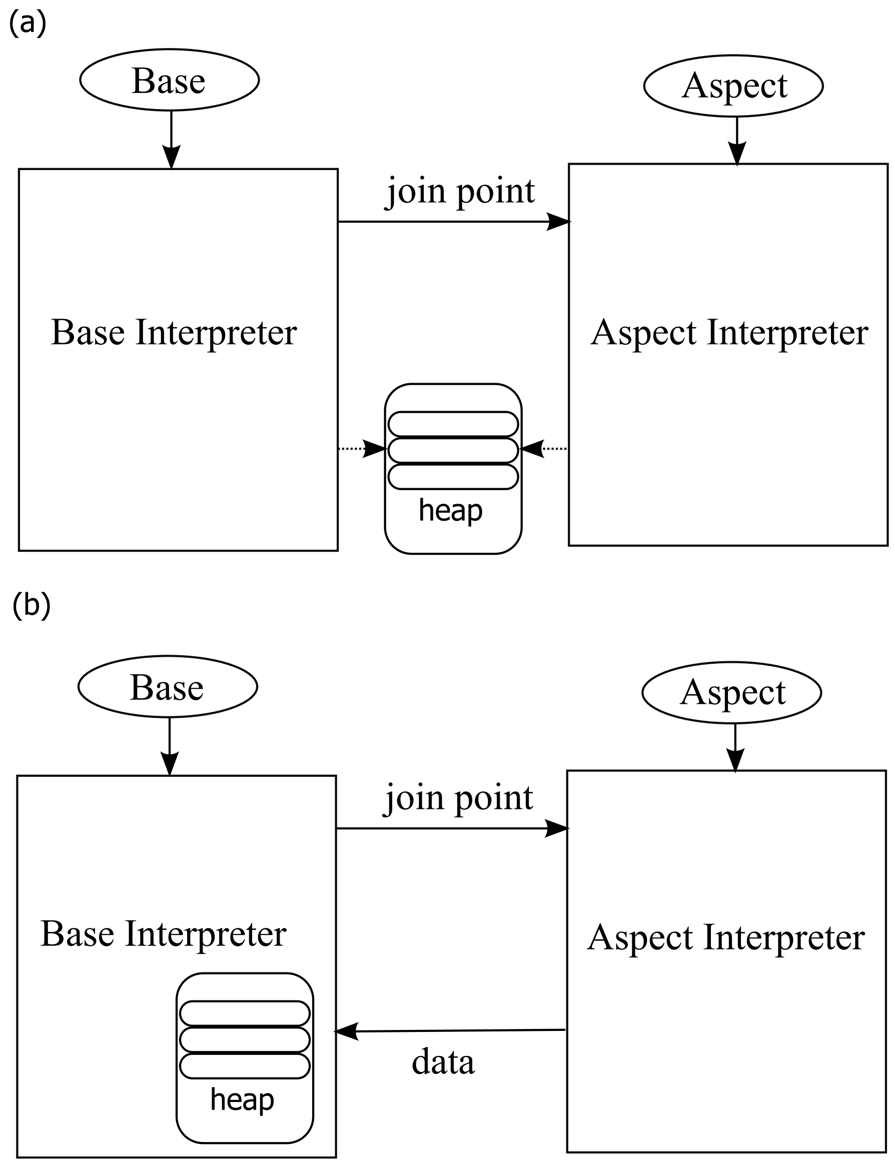

Figure 5.3: A general architecture of the base and the aspect interpreter

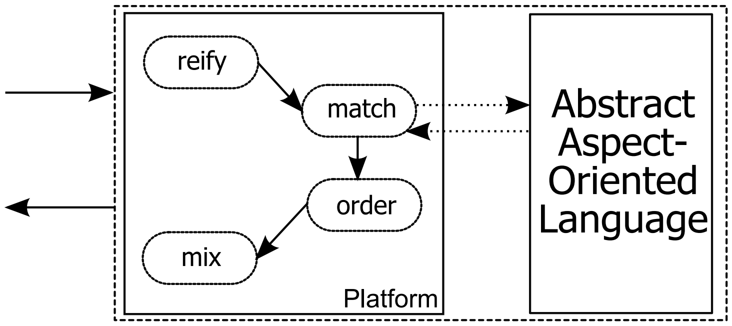

Figure 5.4: The decomposition of the aspect interpreter in two parts

5.3.2 Separation between what is common and what is specific in the aspect interpreter

After separating the base and the aspect interpreter, we have to explore the aspect interpreter in order to ensure its extensibility to concrete aspect-oriented languages. Figure 5.4 shows that the aspect interpreter is decomposed into two parts, one consists of the aspect `Platform` and another contains the core semantics of the abstract aspect language. The aspect `Platform` implements the four subprocesses defined in Listing 5.1.4 while the abstract language interpreter consists of the implementation of the Abstract Syntax Tree with its associated semantics as described above (Section 5.1).

5.3.3 Prototyping and composition AOPLs

Figure 5.5 shows how the abstract AOPL can be extended in order to prototype a concrete one. The aspect `Platform` is not modified but the abstract aspect language interpreter is extended. As an example, Section 5.4 discusses the prototyping/implementation of a lightweight version of AspectJ. Once we have several languages implemented in the above way (extending the abstract aspect-oriented language), they can be composed as we compose several aspects in the abstract language.

5.4 Lightweight AspectJ

Lightweight AspectJ is an AOP extension of MetaJ. It consists of implementing several AspectJ-like pointcut designators as well as intertype declaration. Since the semantics of AspectJ pointcuts is deeply described in the following chapters, the prototype of Lightweight AspectJ is a proof of concept and hence the semantics of pointcut designators might be slightly different from the AspectJ one.

5.4.1 Pointcut designators

With the methods `staticTest` and `dynamicTest`, a join-point selector deals with static matching (as performed in the semantics by the function ψ) as well as dynamic matching (as performed by the function ϕ). It does so by aggregating the results coming from its constituent primitive selectors, instances of the class `PrimitiveSelector`, which implements the notion of pointcut designators of AspectJ. Each pointcut designator is implemented as a subclass of `PrimitiveSelector` with an appropriate definition of the methods `staticTest` and `dynamicTest`, as specified by the semantics of AspectJ (see [94]). While the method `staticTest` returns true if the pointcut matches the static information contained in the join point: method name, static type, field name, etc., the

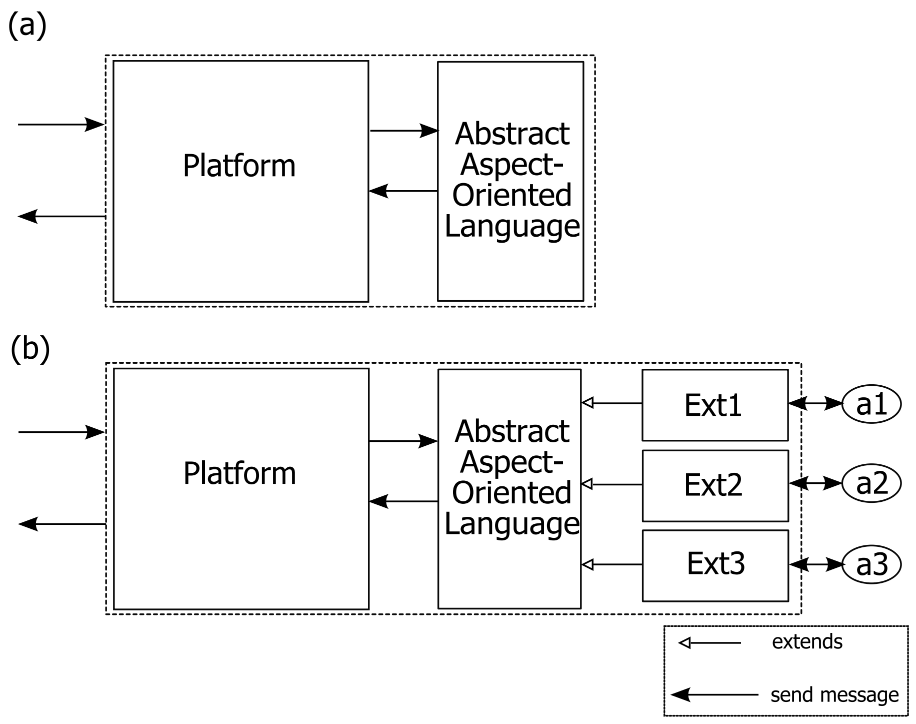
(a)
Platform
Abstract Aspect-Oriented Language
(b)
Platform
Abstract Aspect-Oriented Language
Ext1
Ext2
Ext3
a1
a2
a3
extends
send message

Figure 5.5: (a) General architecture (b) Implementation and composition of different aspect-oriented languages

```
public class AspectJImpl extends Aspect {
  List<SelectorAdviceBinding> bindings;
  public AspectJImpl(List<SelectorAdviceBinding> bindings) {
    this.bindings = bindings;
  }
  public List<Phi> staticTest(Exp exp) {
    ArrayList<Phi> phis = new ArrayList<Phi>();
    for (SelectorAdviceBinding binding: bindings) {
      if (binding.staticTest(exp)) phis.add(new Phi(binding));
    }
    return phis;
  }
}
```

Listing 5.18: The implementation of a lightweight AspectJ aspect

method `dynamicTest` deals with the runtime information contained in the join point: dynamic type, receiver type, etc.

5.4.1.1 Call

The static information of the join point is sufficient for the designator call to select a join point. Listing 5.19 gives the implementation of the corresponding primitive selector subclass `Call`. The method `staticTest` tests the type of expression to choose: `ExpMethodCall` (Line 11). This type was added to the AST of MetaJ because the class `ExpMethod` is not sufficient to capture the method-call join point as explained before. To perform the matching, we compare the static type (`e.getStaticType()`) of the evaluated expression and the `className` attribute (Line 13), the method name of the `ExpMethodCall` (`e.method`) and the `methodName` attribute (Line 14) while the method `dynamicTest` always returns true.

5.4.1.2 Execution

The static information of the join point is also sufficient for the designator `execution` to select or not a join point. Listing 5.19 gives the implementation of the corresponding primitive selector subclass `Execution`. The method `staticTest` tests the type of expression to choose `ExpMethodBody` (Line 34). This type was also added to the AST of MetaJ to capture the method-execution join point. To perform the matching, we compare the static type (`e.getStaticType()`) of the evaluated expression and the `className` attribute (Line 36), the method name of the `ExpMethodBody` (`e.method`) and the `methodName` attribute (Line 37) while the method `dynamicTest` always returns true.

5.4.1.3 This

The designator `This` (see Listing 5.20) tests whether the current object (*this*) at the current join point is of a specified type. The method `staticTest` of the corresponding primitive selector subclass always returns true whereas the type *this* is determined at runtime by the method `dynamicTest`. This method looks the environment up for the value having the name `"this"` then checks whether the type of the resulting instance is from the type given in the designator or a subclass of it.

5.4.1.4 Target

The implementation of the designator `Target` is very similar to the implementation of the designator `This` (see Listing 5.20) except that we look for the variable `"target"` instead of `"this"`.

5.4.2 Inter-type declarations

Whereas pointcuts and advice make it possible to alter the behavior of the program, inter-type declarations make it possible to alter its structure (see Chapter 2). Let us sketch how we can implement this feature in our lightweight version of AspectJ. The creation of a class in MetaJ is the result of evaluating an instance of `ExpClass` (see Listing 5.21). If we capture the evaluation of this type of expression, inter-type declaration can be implemented as (behavioral) aspects applied to the base interpreter. This consists of adding a new instance of `Exp` to `expMethodList` (Line 5 in Listing 5.21) to add a new method or adding a new instance of `Exp` to `expDataList` (Line 4 in Listing 5.21) to add a new field.

5.4.3 Parsing

The LL(k) grammar of Java used in MetaJ is extended to support the extension. While the extension of the MetaJ interpreter to support AOP was modular, unfortunately the extension

```java
public class Call extends PrimitiveSelector {
  String className;
  String methodName;

  public Call(String classname, String methodname) {
    this.className = classname;
    this.methodName = methodname;
  }

  public boolean staticTest(Exp exp) {
    if (exp.getClass().getName().equals("ast.ExpMethodCall")) {
      ExpMethodCall e = (ExpMethodCall) exp;
      return e.getStaticType().equals(this.className)
             && e.method.equals(this.methodName);
    } else
      return false;
    }

  public boolean dynamicTest(Environment env) {
    return true;
  }
}

public class Execution extends PrimitiveSelector {
  String className;
  String methodName;

  public Execution(String classname, String methodname) {
    this.className = classname;
    this.methodName = methodname;
  }

  public boolean staticTest(Exp exp) {
    if (exp.getClass().getName().equals("ast.ExpMethodBody")) {
      ExpMethodBody e = (ExpMethodBody) exp;
      return e.getStaticType().equals(this.className)
             && e.method.equals(this.methodName);
    } else
      return false;
  }

  public boolean dynamicTest(Environment env) {
    return true;
  }
}
```

Listing 5.19: Lightweight AspectJ - implementation of the selectors `Call` and `Execution`

```
public class This extends PrimitiveSelector {
  String classname;

  public This(String classname) {
    this.classname = classname;
  }

  public boolean staticTest(Exp exp) {
    return true;
  }

  public boolean dynamicTest(Environment env) {
    try {
      Instance ins = (Instance) env.lookup("this").value;
      ast.Class cls = (ast.Class) Main.globalE.lookup(this.
         classname).value;
      return ins.instanceLink.equals(cls)
             || ins.instanceLink.isSubClassOf(this.classname);

    } catch (NullPointerException nullpointer) {
      return false;
    }
  }
}

public class Target extends PrimitiveSelector {
  String classname;

  public Target(String classname) {
    this.classname = classname;
  }

  public boolean staticTest(Exp exp) {
    return true;
  }

  public boolean dynamicTest(Environment env) {
    try {
      Instance ins = (Instance) env.lookup("target").value;
      ast.Class cls = (ast.Class) Main.globalE.lookup(this.
         classname).value;
      return ins.instanceLink.equals(cls)
             || ins.instanceLink.isSubClassOf(this.classname);

    } catch (NullPointerException nullpointer) {
      return false;
    }
  }
}
```

Listing 5.20: Lightweight AspectJ - implementation of the selectors `This` and `Target`

```java
public class ExpClass extends Exp {
  private String name;
  private String upId;
  private Exp expDataList;
  private Exp expMethodList;

  ExpClass(String upId, String name,
              Exp expDataList, Exp expMethodList) {
    this.upId = upId;
    this.name = name;
    this.expDataList = expDataList;
    this.expMethodList = expMethodList;
  }

  public Data eval(Environment localE) {
    // get father class
    Class c1;
    if (this.upId != null)
      c1 = (Class)(Main.globalE.lookup(this.upId).read());
    else
      c1 = null;
    // create a new data list
    DataList dataList = (DataList)this.expDataList.eval(localE).
        read();
    // create a new method list
    MethodList methodList = (MethodList)this.expMethodList.eval(
        localE).read();
    // create a new class
    Class c2 = new Class(c1, dataList, methodList);
    // add the class to the global environment
    Main.globalE.add(this.name, new Data(c2));
    // dummy return value
    return null;
  }

  public String toString() {
    return "class " + this.name
        + " extends " + this.upId + " {\n"
        + this.expDataList.toString()
        + this.expMethodList.toString() + "}\n";
  }
}
```

Listing 5.21: The ExpClass implementation

is not modular for the parser (due to some limitations of JavaCC: it does not permit the reference of a grammar file from another one). The `Java2ExpVisitor` is also extended to visit the AspectJ nodes and generate the corresponding runtime entities. For example, when it visits `ASTAspectDeclaration`, it returns an instance of `AspectJImpl` and adds it to the list of aspect instances in the platform (with default scope).

5.5 Conclusion

In this chapter, we have defined an aspect interpreter for an abstract aspect-oriented language. The aspect interpreter is also split into two parts, one is general and it will be found in different aspect-oriented languages, the other is specific to each aspect-oriented language. The implementation of a concrete aspect language consists of extending the abstract interpreter without changing the common part. We have presented a general architecture for the modular composition of the aspect and the base interpreter.

We have also described a framework that can be used to prototype aspect-oriented languages but on top of an interpreted subset of Java. One way of generalizing this work to Java needs access to the JVM and to modify it, which is a complex task with portability issues. The next chapter shows how the implementation of the framework for Java can be realized, without changing the JVM, with a compiler-based two-step weaving mechanism that instruments the program instead of instrumenting the interpreter.

Chapter 6

CALI: Common Aspect Language Interpreter

Contents

In the previous chapter, we have seen how we can construct aspect languages on top of an incomplete version of Java by extending MetaJ, the interpreter of this Java version written in Java itself. Implementing an effective framework imposes the use of the whole of Java as base language. Implementing the approach of the previous chapter with a full Java interpreter (or a JVM) is complex and far from our goals which are: implementing a framework for easy prototyping, composing and testing of AOPLs.

In this chapter, we use a *compiler-based two-step weaving* mechanism which facilitates the construction of our framework on top of the whole of Java. Note that we do not need to care about performance because it is not our objective but we will focus on the design and the extensibility of the interpreter. We name this framework CALI for (Common Aspect Language Interpreter).

CALI reuses AspectJ to perform a first step of static weaving, which we complement by a second step of dynamic weaving, implemented through a thin interpretation layer. This can be seen as an interesting example of reconciling interpreters and compilers, the dynamic and the static world.

This chapter is organized as follows: Section 6.1 describes the compiler two-step weaving mechanism. Section 6.2 explains the semantics of join point interpretation using the CASB. Section 6.3 overviews the architecture of CALI and describes its implementation as an (AspectJ) aspect `Platform`. The abstract language interpreted by `Platform` is described in Section 6.4. Finally, Section 6.6 describes how to use CALI to implement concrete aspect languages.

6.1 Compiler-based two-step weaving

In the previous chapter, we have seen the interpreter-based two step weaving approach. In this chapter, the approach is slightly different: instead of instrumenting the interpreter, the program itself is instrumented. We call this a *compiler-based two-step weaving* approach. The first step takes place at compile time, the second step in CALI at runtime. At compile time, we weave a generic aspect, as in the previous chapter, called `Platform` into the base-level class code to introduce an indirection to our interpreter. Each generated join point is evaluated by the interpreter in the context of existing aspects in order to determine which aspects match it.

In fact, a Java interpreter can be represented by $int : p_{Java} \times d$ where int is a program that takes a program p (static information) and some data d (dynamic information). Using this notation, the relation between a *compiler-based two-step weaving* and an *interpreter-based two-step weaving* can be explained as follows:

1. Partially evaluating int with p_{Java} returns a program equivalent to p_{Java} in the implementation language of the interpreter int.
2. *Interpreter-based two-step weaving* consists of instrumenting int in order to introduce AOP semantics.
3. Partially evaluating the instrumented int with p is equivalent to partially evaluate int with an instrumented version of p.

Instead of weaving the aspect `Platform` into the base interpreter as in Chapter 5, it is woven into the base program. The pointcut of the aspect `Platform` implements the **reify** process described in [63] (first step of weaving). At runtime, the advice of the aspect `Platform` evaluates the current join point with aspects, selects the set of aspects that match this join point, orders and executes them (second step of weaving). AspectJ is used to capture join points then, the framework makes it possible to:

– Proceed with these join points. This issue is detailed in 6.4.3.
– Access to the information reified by AspectJ. This issue is detailed in 6.4.6.

The difference between a weaver and our interpreter is that, when using a weaver, the four subprocesses (**reify**, **match**, **order** and **mix**) are executed at compile time while in our approach the **reify** process is still executed at compile time but the three others are executed at runtime.

6.2 Semantics of CALI

We base the semantics of CALI (see Section 2.2.6.2) on a modified version of CASB to introduce the notion of aspect group.

Let us reconsider what happens when a base instruction i is executed. When no aspect statically matches, $\psi(i)$ returns an empty list of statically matching aspects and the woven program simply behaves as the base program:

$$\textsc{NoAdvice}\ \frac{\psi(i) = \epsilon \qquad (i : C, \Sigma) \rightarrow_b (C', \Sigma')}{(i : C, \Sigma, P) \rightarrow (C', \Sigma', P)}$$

Otherwise, ψ returns a list of instructions ϕ, which is denoted by Φ:

$$\textsc{Around}\ \frac{\psi(i) = \Phi \qquad \alpha(\Phi, \Sigma) = (\phi, \Phi')}{(i : C, \Sigma, P) \rightarrow (\phi : \texttt{pop} : C, \Sigma, (\Phi' :: [\bar{i}]) : P)}$$

The rules that we show here are actually a slightly generalized version of the rules that have been presented in the state of the art, thanks to the introduction of the function α. We could consider that Φ is a set but, in the case of AspectJ, it is important to consider Φ as a list. This corresponds to the fact that some static scheduling takes place. This ordering will not be questioned by the other rules, although it could by appropriately defining the function α. We have introduced this function, not present in the initial CASB rules, to show that there would be the possibility here to perform *dynamic scheduling* that can be needed in some AOPLs implemented on top of CALI. Indeed, in our extended rules the function α chooses, possibly based on Σ, the first instruction ϕ to consider as well as the instructions Φ' to consider if the aspect *proceeds*.

For example, this choice is static in AspectJ: α simply returns the head ϕ and tail Φ' of Φ. The instruction ϕ replaces the advised instruction/join point i and the list Φ', with the *tagged* instruction i, $\bar{i}$, added at its end, is pushed on the proceed stack.

In the initial CASB rules, the proceed stack is organized as a stack of instructions, we have chosen a different organization here: the proceed stack is a stack of *aspect groups*, where an *aspect group* is a list of instructions for statically matching aspects terminated by the corresponding advised instruction. All along the execution, the *current* aspect group, at the top of the proceed stack, holds the current join point together with the pending aspects.

The execution of ϕ is followed by an instruction pop, the corresponding rule Pop is not modified:

$$\textsc{Pop}\ \frac{}{(\texttt{pop} : C, \Sigma, \Phi : P) \to (C, \Sigma, P)}$$

Let us now see what happens when an instruction `proceed` occurs in a piece of advice:

$$\textsc{Proceed}\ \frac{\alpha(\Phi, \Sigma) = (\phi, \Phi')}{(\texttt{proceed} : C, \Sigma, \Phi : P) \to (\phi : \texttt{push}\ \phi : C, \Sigma, \Phi' : P)}$$

The current aspect group is retrieved on top of the proceed stack. The same scheduling function α as in the Around rule is used. A new instruction ϕ (a statically matching aspect or the join point) replaces `proceed` and the current aspect group is updated. An instruction `push` follows the execution of the instruction ϕ. This instruction does not put a new element on top of the proceed stack but rather adds an instruction at the beginning of the current aspect group:

$$\textsc{Push}\ \frac{}{(\texttt{push}\ \phi : C, \Sigma, \Phi : P) \to (C, \Sigma, (\phi : \Phi) : P)}$$

The instruction `push` is used to rebuild the initial aspect group in order to cater for multiple instructions `proceed` in the same advice.

These rules will be used in the next sections in order to build our framework for prototyping and composing AOPLs.

6.3 Architecture of CALI

CALI represents a good example of mixing together compilation and interpretation. The base program runs as a normal Java program on a standard JVM while aspects are partially interpreted. All the aspects (from any language implemented with CALI) must be translated, using a parser, to a specific structure (in Java) defined by the abstract aspect language, and they are represented at runtime by Java objects conforming to this specific structure. As we said, two-step weaving [33] is used to weave the interpreted aspects into the compiled Java code:

1. The first step takes place in AspectJ at compile time, through an aspect called `Platform` (see Listing 6.1), which results in the instrumentation by AspectJ of all possible join points in the base program by defining a pointcut `call(* *.*(..))|| execution(* *.*(..))|| set(* *)...;`
2. The second step takes place at runtime, when the advice of the aspect `Platform`, which behaves as an *interpretation layer*, evaluates the current AspectJ join point (accessed through `thisJoinPoint`) by looking for matching aspects before executing them. The "base" Java parts of the selected advices are then again executed as plain Java code.

The **reify** process is here implemented as a pointcut which, at runtime, produces all possible join points in the base program (`pointcut reifyBase`) which should be associated with another pointcut (`pointcut reifyAdvice`) in the case of a language that allows the reifying of its advices. The rest of the process is very similar to static weaving, except that we are dealing with runtime entities. The around advice of `Platform` directly implements the rule Around of the semantics, where the combination of the calls to `match` and `order` implements the function Ψ.

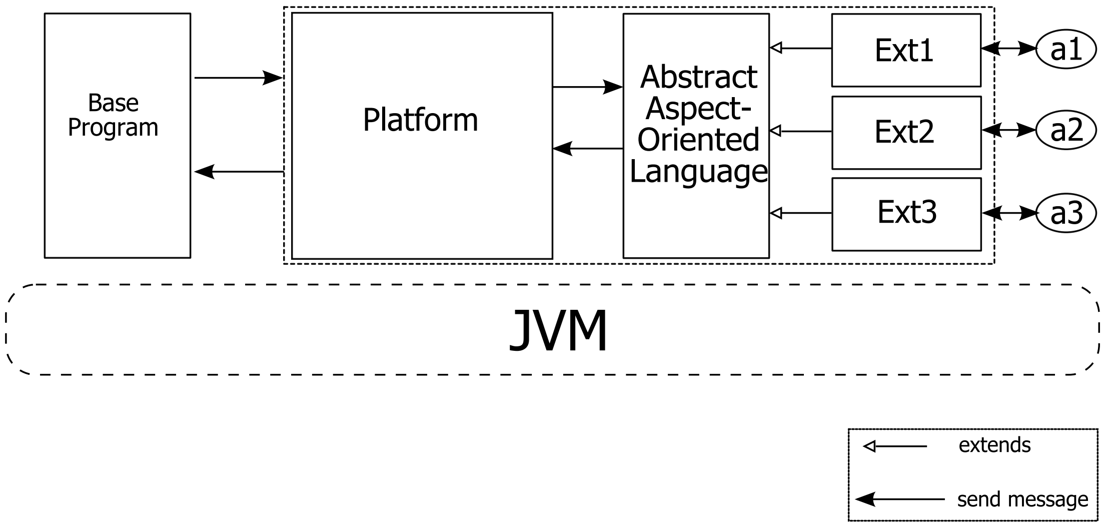

Figure 6.1: The architecture of CALI

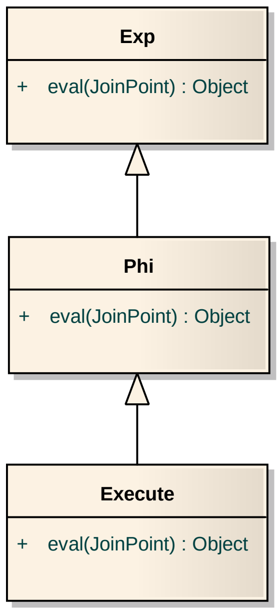
Exp
+ eval(JoinPoint) : Object
Phi
+ eval(JoinPoint) : Object
Execute
+ eval(JoinPoint) : Object

Figure 6.2: The class diagram of expressions used in CALI

```
public aspect Platform {
  ...
  pointcut reifyBase():
    (call(* *.*(..)) || execution(* *.*(..)) || set(* *) ...)
      && !within(java..*)
      && !within(org.aspectj..*)
      && !within(aspectj..*);
  pointcut reifyAdvice(): ...
  pointcut reify(): reifyBase() || reifyAdvice();
  Object around(): reify() {
    List<Phi> phis = match(thisJoinPoint);
    if (phis == null)
      return proceed();
    else {
      phis = order(thisJoinPoint, phis);
      Execute jp = new Execute() {
        public Object execute() {
          return proceed();
        }
      };
    AspectGroup aspectGroup = new AspectGroup(phis, jp);
      return mix(thisJoinPoint, aspectGroup);
    }
  }

  public static List<Phi> match(JoinPoint jp) { ... }
  public static List<Phi> order(JoinPoint jp, List<Phi> phis) {
    List<Phi> pseudoAspectGroup = aspectJOrder(phis);
    return pseudoAspectGroup;
  }
  public static Object mix(JoinPoint jp,
                           AspectGroup aspectGroup) {
    Phi phi = aspectGroup.remove(0);
    ProceedStackManager.push(aspectGroup);
    Object o = phi.eval(jp);
    ProceedStackManager.pop();
    return o;
  }
}
```

Listing 6.1: The aspect `Platform`

```java
public class ProceedStackManager {
  public static HashMap<Thread, Stack<AspectGroup>> map = new
      HashMap<Thread, Stack<AspectGroup>>();
  public synchronized static void push(AspectGroup group) {
    Stack<AspectGroup> s = map.get(Thread.currentThread());
    try {
      s.push(group);
    } catch (NullPointerException e) {
      map.put(Thread.currentThread(), new Stack<AspectGroup>());
      s = map.get(Thread.currentThread());
      s.push(group);
    }
  }
  public synchronized static void pop() {
    map.get(Thread.currentThread()).pop();
  }
  public synchronized static AspectGroup peek() {
    return map.get(Thread.currentThread()).peek();
  }
}
```

Listing 6.2: The proceed stack manager

The method `match` takes the current join point as a parameter and returns a list of `Phi` instances. This list is built by calling the method `staticTest` on each available aspect instance and by creating a `Phi` instance for each matching aspect. The class `Phi` is a subclass of a general class of expressions `Exp` (see Figure 6.2).

The method `order` orders the list according to the semantics of aspect precedence in AspectJ and adds the join point at the end of the ordered list, as an instance of a subclass of `Phi`.

The method `mix` pushes the aspect group onto the proceed stack and evaluates the first aspect. As a first step, this evaluation will call the method `dynamicTest` of the aspect. The proceed stack is then trimmed and the result of the evaluation returned. Each instance of `Phi` is basically a pair dynamic test/advice, as seen in Chapter 5. except the last one, which is the join point wrapped as an instance of `Phi`. The wrapping is done inside the constructor of the class `AspectGroup`, which takes as parameters the list of instances of `Phi` and an instance of the class `Execute` representing the advised join point. This class will be detailed later when we explain how to proceed with the advised join point. The evaluation starts then with the first instance of `Phi`. The dynamic test is performed. In the case of a positive result, the corresponding advice is executed, otherwise a special aspect-level instruction, `proceed`, is executed and the execution proceeds with the next instance of `Phi`.

Figure 6.1 shows the architecture of CALI. Due to the compiler two-step weaving, the join points are directly communicated from the instrumented base program instead of the base interpreter as we have seen with the interpreter two-step weaving in Chapter 5.

6.3.1 Proceed stack management

The `ProceedStackManager` class (Listings 6.2) implements the proceed stack of CALI. Its interface contains three methods: `push`, `pop` and `peek`.

In a concurrent context, each proceed stack corresponds to a thread. For this reason, our proceed-stack manager is thread-aware. We define a *stack* for each thread. This is implemented by the `HashMap<Thread, Stack<Exp>> map` attribute. This attribute links the thread to its stack. Each time one of the methods `push`, `pop` and `peek` is called, we must get the current thread in

```
abstract class JoinPointSelector {
  abstract boolean staticTest(JoinPoint jp);
  abstract boolean dynamicTest(JoinPoint jp);
}
```

Listing 6.3: The class `JoinPointSelector`

```
public abstract class Advice<G> extends Exp {
  public JoinPoint thisJoinPoint;
  public Object eval(JoinPoint jp) {
    thisJoinPoint = jp;
    return this.adviceexecution();
  }
  public G proceed() throws java.util.EmptyStackException {
    Exp exp = ((AspectGroup)ProceedStackManager.peek()).pop();
    G result = (G) exp.eval(thisJoinPoint);
    ((AspectGroup)ProceedStackManager.peek()).push(exp);
    return result;
  }
  public abstract G adviceexecution();
  public void deploy(Aspect anAspect){ .. }
  public void undeploy(Aspect anAspect){ .. }
}
```

Listing 6.4: Generic part of the implementation of a piece of advice in CALI

order to retrieve the proper stack.

6.4 Abstract Aspect-Oriented language

Similarly to the approach described in Chapter 5, each aspect is represented by an instance of a class `Aspect`, which encapsulates a list of selector/advice bindings, instances of `SelectorAdviceBinding`. Each selector/advice binding contains a join-point selector, instance of a class `JoinPointSelector`, and an advice, instance of a class `Advice`. Even the structure is the same than the one used in the previous chapter, the implementation of some classes is slightly different as we will see in the following.

6.4.1 Join-point selector

As we said before, the join point selector selects a join point in two steps, a *static* and a *dynamic* step, implemented through the methods `staticTest` and `dynamicTest` (see Listing 6.3). The difference between the class `JoinPointSelector` in this chapter and the one in Chapter 5 is due to the representation of a join point as an instance of `JoinPoint` instead of as a pair (`Exp`, `Environment`).

6.4.2 Advice

A piece of advice in CALI is represented using Java *closures* in the form of anonymous inner classes. Each advice is an instance of the generic class `Advice` (see Listing 6.4). The type `G` defines the return type of the advice. The class `Advice` contains an abstract method called `adviceexecution()`, which must be implemented when defining a concrete subclass. This method defines the advice body. The return type of this method is the generic type given to the class `Advice`. For example, the following AspectJ piece of advice:

```
MyClass around(): p(){
  System.out.println(thisJoinPoint.getSourceLocation());
  return proceed();
}
```

is translated as follows:

```
new Advice<MyClass>(){
  public MyClass adviceexecution(){
    System.out.println(thisJoinPoint.getSourceLocation());
    return proceed();
  }
}
```

Here we define a piece of advice which returns an instance of `MyClass`. As a result, the return type of `adviceexecution()` is `MyClass`. The body of this method encapsulates the code of the AspectJ advice. The execution of this method will generate an `adviceexecution` join point which will be captured by an `adviceexecution` pointcut, a primitive AspectJ pointcut, when implementing AspectJ on top of CALI in the next chapter. The matching will be done by verifying that the join point is of type: an execution join point of `Advice.adviceexection()`.

The method `proceed` of the class `Advice`, which should be called within `adviceexecution`, implements the rule PROCEED in the semantics. It retrieves the expression to be executed from the proceed stack (`ProceedStackManager.peek().pop()`), and launches the evaluation (see Listing 6.4 - note that the casts to `AspectGroup` are only used for documentation purposes). The expression could be an instance of `Phi` or `Execute` which will be detailed in Section 6.4.3.

The class `Advice` also has `deploy` and `undeploy` methods in order to be called within `adviceexecution` to deploy or undeploy the corresponding aspect. This will be detailed later in the chapter.

6.4.3 Proceed

We say that all our aspects are of type around because an around aspect can be used to implement the two other types of aspect (*before* and *after*). When an aspect proceeds, the next aspect that matches the join point will be executed or the captured join point will be executed when there are no more aspects. Let us explain how our framework supports this feature.

Calling proceed within the advice of the aspect `Platform` is a simple way to execute the matched join point. However, according to the semantics of CALI, it is up to the mix process to perform this execution. The main key that we use to execute the advised join point is that we instanciate the class `Execute` (see Figure 6.2) in the advice of `Platform` as a Java *closure* (in the form of anonymous inner classes). A closure is a function that captures the bindings of free variables in its lexical context. Consider the following code which is an excerpt from `Platform` advice:

```
new Execute() {
  public Object execute() {
    return proceed();
  }
}
```

The method `execute` (constituting the closure) is dynamically defined and refers to proceeding with the join point. Calling the method `execute` will execute the advised join point. In CALI, when a join point is captured, all the returned instances of `Phi` (except the first one which is directly

```java
public class SelectorAdviceBinding {
  JoinPointSelector jps;
  Advice advice;
  public SelectorAdviceBinding(JoinPointSelector jps,
                               Advice advice) {
    this.jps = jps;
    this.advice = advice;
  }
  public boolean staticTest(JoinPoint jp) {
    return jps.staticTest(jp);
  }
  public Advice dynamicTest(JoinPoint jp) {
      return (jps.dynamicTest(jp))? advice : null;
  }
}
```

Listing 6.5: The implementation of selector/advice bindings

```java
public abstract class Aspect {
  public abstract List<Phi> staticTest(JoinPoint jp);
  public void deploy() { .. }
  public void undeploy() { .. }
}
```

Listing 6.6: The class `Aspect`

executed) are encapsulated within an instance of `AspectGroup` and pushed on the proceed stack (this is done in the arround advice of `Platform`). In addition, the instance of `Execute` is added (according to the CASB to represent the captured join point). When the last aspect proceeds, the framework pops the instance of `Execute` and calls its method `execute()` (as all the instances of `Phi`).

6.4.4 Selector/Advice Binding

The class `SelectorAdviceBinding` (See Listing 6.5) implements the selector/advice binding notion of the abstract language. This is similar to the class `SelectorAdviceBinding` of Chapter 5 except it uses, as mentioned before, a different join point representation.

6.4.5 Aspect

The interface of an aspect (Listing 6.6) provides the method `staticTest(JoinPoint)` which returns a list of `Phi` instances. As we said when extending MetaJ, the *mechanism* implementing how the aspect calculates the list of `Phi` instances using their selector/advice bindings will be implemented for each concrete AOPL.

In order to allow *dynamic deployement*, the interface of an aspect also provides two other methods: `deploy` and `undeploy`. When calling the method `deploy` (`undeploy` resp.) on an instance of `Aspect`, the instance will be added (removed resp.) to the *aspect environment*.

6.4.6 Reflective access to the join point

As the `thisJoinPoint` field of the class `Advice` contains the reference to the advised join point, access to this field within the advice body (i.e. from the body of `Advice.adviceexecution()`)

gives access to the advised join point, as a result, gives access to all the reflective information of the join point provided by AspectJ itself.

6.5 Principles of matching a join point by Join-point Selectors

Each join point encapsulates the corresponding signature (defined by the AspectJ weaver) in addition to corresponding dynamic information (all the associated information, signature and dynamic, is accessible at runtime using `thisJoinPoint`). At runtime, the advice `Platform` takes the current join point and send it to be evaluated by all the join-point selectors of the existing aspect instance. The role of join-point selectors will be to compare the signature of the join point with the signature specified in each join-point selector.

Requirement 5 *The principle of matching in any Aspect-Oriented language implemented using CALI is based on the join-point selector comparing the information associated with the join point captured by* `Platform` *with the signature specified in the join-point selector.*

This feature will be clarified in Chapter 7 where the pointcut designators of AspectJ are implemented as join-point selectors.

6.6 Implementing a concrete AOPL with CALI

In this section, we describe how to use CALI for implementing concrete AOPL. The language designer must provide:

– The implementation of aspects as a class that implements the aspect interface, in particular the method `staticTest(JoinPoint)`.
– The implementation of selectors that inherit from `JoinPointSelector`. Each selector must implement the two methods `boolean staticTest(Joinpoint)` and `boolean dynamicTest(Joinpoint)`. The first one defines whether the selector should match the static part of the join point, this means the parts that can be determined at compile time even if this matching of the join point is done by interpretation at runtime because the instance of `JoinPoint` contains several pieces of static information (accessible by the `getStaticPart()` provided by the interface of the class `JoinPoint`). The second one defines whether the selector should match the dynamic part of the join point by accessing the runtime information using methods like `getThis(), getTarget(), etc.` provided by the `JoinPoint` interface.
– The implementation of advice. There are two cases: the first is that the advice language is also Java, the designer can use advice as predefined in CALI. The second is that the advice language is not Java, the designer implements the method `adviceexecution` as an evaluator of the advice-language expression.

The implementation of selectors as modular and independent entities improves their reusability. We will see in the next chapters how EAOP (for Java) and AspectJ aspects share the same selectors and advice but differ in the method `staticTest(JoinPoint)` of the aspects. For AspectJ, this method matches all the selector/advice bindings that statically match the join point. For EAOP, it matches a single selector/advice binding depending on the aspect state. Still, the two languages share the selectors `call`, `execution`, `this`, `target`, etc. Once AspectJ is implemented, the implementation of EAOP is straightforward.

Chapter 7

AspectJ plugin on top of CALI

Contents

This chapter describes an implementation of a significant subset of AspectJ as a proof of concept, taking into account some non trivial features of AspectJ, like the use of static types in the call and execution pointcuts. In Section 7.1, we overview the different elements of this implementation. Section 7.2 presents by an example how an aspect is represented in this implementation before explicitly describing the different elements in the following sections. Section 6.5 reviews the principle of matching a join point in AspectJ. Section 7.3 presents an in-depth analysis of AspectJ pointcut semantics before describing their implementation with CALI as join-point selectors. The aspect representation in detailed in Section 7.4. Section 7.5 discusses the implementation of the advice precedence feature. Section 7.6 shows how concrete AspectJ syntax is translated to the prototype representation. Section 7.7 concludes this chapter.

7.1 AspectJ on top of CALI

Following Section 6.6, the implementation of AspectJ on top of our interpreter consists of:

1. Defining the class `AspectJ` (see Listing 7.1) that inherits from the class `Aspect` of CALI. The class `AspectJ` contains a selector/advice binding (instance of the class `SelectorAdviceBinding`) and defines the method `staticTest`. The `staticTest` method forwards the call to the `staticTest` method of its selector/advice bindings.
2. Defining the different *pointcut designator*s (`call`, `execution`, etc.) as join-point selectors implemented as classes that inherit from the class `JoinPointSelector` defined in CALI.
3. Using the class `Advice` defined in CALI to implement a piece of advice. This is because AspectJ uses Java to define the pieces of advice. Advice execution will be reified because, in AspectJ, there is a special join point corresponding to advice execution, which can be matched by the pointcut designator `adviceexecution`. This pointcut is usually used in order to exclude, using the not operator (`!` in concrete syntax), join points that are a result of advice execution. We also need to reify all types of join points within the AspectJ aspect

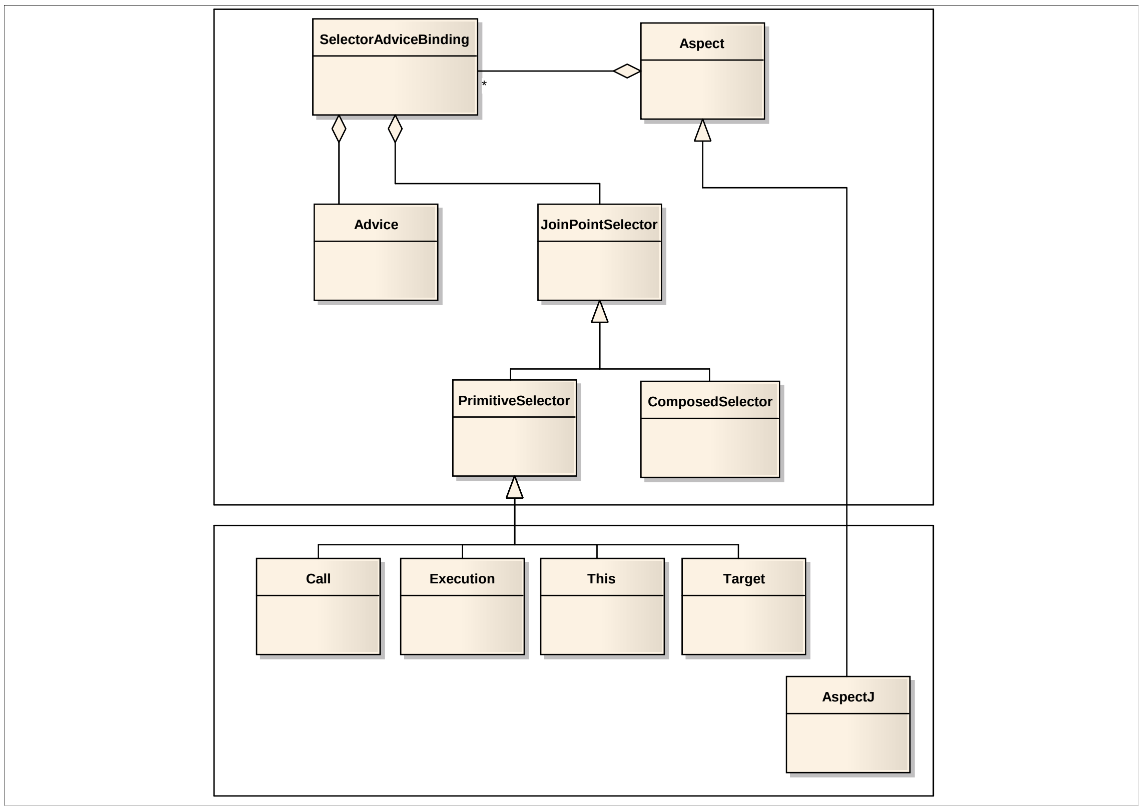

Figure 7.1: AspectJ pointcuts on top of CALI

```
public class AspectJ extends Aspect {
  List<SelectorAdviceBinding> bindings;
  public AspectJ(List<SelectorAdviceBinding> bindings) {
    this.bindings = bindings;
  }
  public List<Phi> staticTest(JoinPoint jp) {
    ArrayList<Phi> phis = new ArrayList<Phi>();
    for (SelectorAdviceBinding binding : bindings) {
      if (binding.staticTest(jp))
        phis.add(new Phi(binding));
      }
    return phis;
  }
  public void addSelectorAdviceBinding(SelectorAdviceBinding
     binding)
  {
    bindings.add(binding);
  }
}
```

Listing 7.1: The AspectJ aspect representation

```java
public class A {
  public String f(String s){
    System.out.println(s+" in A");
    return s;
  }
  public void g(){
    System.out.println("g in A");
  }

  public static void main(String[] args){
    A a = new A();
    a.f("");
  }
}
```

Listing 7.2: A base program

```java
public aspect MyAspect {
  pointcut p(): call(String A.f(String));
  String around(): p(){
    System.out.println("Hello");
    return proceed();
  }
}
```

Listing 7.3: `MyAspect` aspect written in AspectJ

except the join points generated by the implementation level code: the method `staticTest` and the field `list`. To clarify this feature, let us consider the following piece of code:

```
pointcut reifyAspectJ():
  (
    !call(AspectJ.staticTest) &&
    !execution(AspectJ.staticTest) &&
    !set(AspectJ.list) &&
    !get(AspectJ.list) &&
  ) &&
  within(AspectJ) &&
  cflow(execution(* AspectJAdvice.adviceexecution(..)));
```

This pointcut will intercept the execution of `AspectJAdvice.adviceexecution` in addition to every join point in the aspect except the implementation-level ones. In order to be able to match AspectJ advice-related join points, the `reify` pointcut in the platform becomes:

```
pointcut reify(): reifyBase() || reifyAspectJ();
```

7.2 Example

Before proceeding with the implementation of AspectJ features, let us consider an example of AspectJ and see how it is implemented using the CALI AspectJ plugin. Listing 7.2 shows a class `A` with two methods `String f(String s)` and `void g()`. In Listing 7.3, we see an aspect `MyAspect` with a pointcut that intercepts calls to `A.f(String)` where an around piece of advice should be executed.

```
public class MyAspect extends AspectJ {
  public MyAspect() {
    Class[] l = { String.class };
    addSelectorAdviceBinding(new SelectorAdviceBinding(
            new Call(A.class, "f", String.class, new Class[] {
               String }),
            new Advice<String>() {
              public String adviceexecution() {
                System.out.println("Hello");
                return proceed();
              }
            }
    ));
  }
}
```

Listing 7.4: The traduction of `MyAspect` into CALI

Listing 7.4 shows the result of a translation of AspectJ aspects to the AspectJ prototype: each pair *pointcut, advice* is translated into an instance of `SelectorAdviceBinding`. The pointcut is translated to an instance of a subclass of `JoinPointSelector`. In the example, the pointcut expression:

```
pointcut p(): call(String A.f(String))
```

is translated into:

```
new Call(A.class, "f", String.class, new Class[] { String })
```

and the advice:

```
String around(): p() {
  return proceed(‘‘MyAspect’’);
}
```

is translated into:

```
new Advice<String>() {
  public String adviceexecution() {
    return proceed(‘‘MyAspect’’);
  }
}
```

7.3 Pointcut Designators as Join-Point Selectors

The role of join-point selectors is to compare the signature of the join point with the signature specified in each join-point selector. It does so by aggregating the results coming from its constituent primitive selectors, instances of the class `PrimitiveSelector`, which implements the notion of *pointcut designators* of AspectJ. Here, each pointcut designator is implemented as a subclass of `PrimitiveSelector` with an appropriate definition of the methods `staticTest` and `dynamicTest`. While the method `staticTest` returns `true` if the pointcut matches the static information contained in the join point: method name, static type, field name, etc., the method `dynamicTest` deals with the runtime information contained in the join point: dynamic type, receiver type, etc.

```java
public class Super {
  public void f(){ }
}
public class Middle extends Super {
}
public class Sub extends Middle {
  public void g(){ }
}
```

Listing 7.5: An example used to analyse `call` and `execution` semantics.

In the following, we start by giving a background about the signature of the join point generated by the platform in order to use these information in the implementation of the methods `staticTest` and `dynamicTest` of each pointcut designators. We analyse the semantics of method-related pointcuts designators (`call` and `execution`) with respect to inheritance [12, 16, 18]. For each of (`call` and `execution`), we give an implementation of the methods `staticTest` and `dynamicTest` according to this analysis. After that, we proceed to the other pointcut designator types.

7.3.1 Background

Before starting to explain the implementation of AspectJ pointcuts with CALI, we show a set of important features used in the rest of the chapter. Note that the semantics of some AspectJ pointcuts is not straightforward and needs to be looked at.

As mentionned before, AspectJ provides a special reference variable, `thisJoinPoint`, accessible from the body of any advice, that contains reflective information about the current join point. The advice of the `Platform` aspect passes this information to all the reified aspect selectors, which can then access this information for matching. In the following, we will use:

- `String JoinPoint.getKind()` returns a string representing the *kind* of the join point. In particular, it returns `method-call` and `method-execution` for method-call and method execution join point, respectively;
- `Signature JoinPoint.getSignature()` returns the signature of the join point. For method-call join points, it returns the signature of the method *existing* in the static type. For method-execution join points, it returns the signature of the method being executed.

The principle of matching a join point in CALI needs to access information contained within this join point. For this reason, we will use the reflection API of Java, in particular:

- `Class Class.asSubClass(Class c)` tests whether `this` is a subclass of `c`. It returns `this` if this is the case and throws an exception otherwise.
- `boolean Class.isAssignableFrom(Class<?> cls)` is the inverse of the previous method and it can be used as an alternative. It determines if the class represented by this Class object is either the same as, or is a superclass of, the class or interface given as parameter.
- `Method Class.getMethod(String name, Class[] parameterTypes)` returns the instance of `Method` (inherited or defined) in `this` with the name and parameter types given as parameters. An exception `NoSuchMethodException` is thrown if no such method is found, an exception `NullPointerException` if the value of `name` is `null`, and an exception `SecurityException` in case of a security exception in the presence of a security manager.

7.3.2 Method-related pointcuts

AspectJ offers two ways to intercept method invocation: one for intercepting the invocation at the caller side with `call(`$T\ P.m(A)$`)`, and another for intercepting the execution of the method body, at the callee side, with `execution(`$T\ P.m(A)$`)`, where T is the return type of the method of interest, P its *declaring* type, and A the types of its arguments.

```java
class Service implements Runnable {
    public void run() { ... }
    public static void main(String[] args) {
        ((Runnable) new Service()).run();
    }
}
```

Listing 7.6: Another example to analyse `call` and `execution` semantics.

A first issue is to understand what is exactly meant by *declaring* type. A first clue is given by considering Listing 7.5 together with:

– the calls `new Sub().f()` and `new Sub().g()`;

– the pointcuts `call(void Middle.f())` and `call(void Middle.g())`.

With the current version of AspectJ (1.6), the only selected join point is the call `new Sub().f()`. The only difference between the calls is that the method `g` is initially defined in a subclass of `Middle`. We cannot just expect `call(void Middle.g())` to select calls to `g` with receivers of type `Middle` (which is actually impossible) or `Sub`, a subtype of `Middle`, the method has also to *exist* in the declaring type `Middle`. Another clue that there is no straightforward choice of semantics is that in earlier versions of AspectJ, the call to `f` was not selected either: the method had to be *defined* in the declaring type given in the pointcut.

A second issue is to understand against which *qualifying* type, on the join point side, the declaring type is matched. It is simple in the previous example because the static and dynamic types of the receivers are the same, but it is not always the case. We shall also see that the story is different on the caller side and on the callee side with the counter-intuitive result that, in the presence of the pointcuts `call(void Middle.f())` and `execution(void Middle.f())`, a call to `Middle.f()` will be captured by the pointcut `call` but the execution of the method `f` will not be captured by the pointcut `execution`. Of course, with respect to a given call, a call join point and an execution join point are different join points: with respect to a given call, the call join point occurs within the context of the caller object, whereas the execution join point occurs within the context of the receiver. Still, the intuition is that two pointcuts `call` and `execution` using the same declaring type should select both [18, 16].

In the following, we revisit the semantics of the pointcuts `call` and `execution` in the current version of AspectJ (1.6).

7.3.2.1 Call

Semantics According to the AspectJ Programming Guide (Appendix B, Semantics) [94], *when matching method-call join points, the declaring type is the static type used to access the method.*

This leads to the common pitfall, signalled in the AspectJ Programming Guide, that the pointcut `call(void Service.run())` does not capture the call to the method `run` of Listing 7.6. As explained in the AspectJ Programming Guide on this very simple example, this is because the declaring type `Service` given in the pointcut is a subtype of the *qualifying* type `Runnable` of the join-point signature. In case of call join point, this *qualifying* type is the *static type used to access the method.*

But what is exactly the *static type used to access the method*? Let us consider the call `((Middle) new Sub()).f()` in the context of Listing 7.5. Is the qualifying type of the join point `Middle`, the static type of the receiver expression, or `Super`, the type where the method `f` of the class `Middle` is defined? Experimenting with AspectJ shows that the qualifying type of the join point is actually the static type of the receiver expression.

Let us go back to Listing 7.6. Of course, the pointcut `call(void Runnable.run())` would capture the call to `run` because the declaring type of the pointcut and the qualifying type of the join point are then the same. But this is hardly enough to understand the semantics of the pointcut `call`. Some more analysis is necessary.

It is also interesting to consider what happens if there is no cast. In that case, the call join point for `new Service().run()` is captured by both pointcuts. If we consider the pointcut `call(void Runnable.run())`, we can see that the join-point qualifying type `Service` is a subtype of the pointcut declaring type `Runnable`. Getting a different result may come as a surprise as this variant base program is not semantically different.

A second experiment consists of replacing the method `run` by a method `myrun` in both the class `Service` and the pointcuts. In that case, the call join point `new Service().myrun()` is captured by the pointcut `call(void Service.myrun())` but not by the pointcut `call(void Runnable.myrun())`, even though the join-point qualifying type (`Service`) is a subtype of the pointcut declaring type (`Runnable`). At first sight, it may look like it is because, unlike in the initial case, the method `myrun` is not defined in `Runnable`. Further experiments would show that the problem is more precisely that the method `myrun` does not *exist* in `Runnable`, *i.e.*, it is neither defined in `Runnable` nor in any of its supertypes.

To summarize, there are two conditions for a pointcut `call(`P`.`m`())` to capture a call join point e`.`m`()`, where J is the qualifying type of the join point (the static type of e):

– $J <: P$ (J is a subtype of P);
– m exists in P.

Implementation Listing 7.7 gives the implementation of call selectors, as a subclass of `PrimitiveSelector`.

All the information necessary to a call selector in order to determine whether a join point matches or not is actually static (it can be obtained from the join point shadow). As a result, the conditions for matching are implemented in its `staticTest` method, whereas its `dynamicTest` method always returns true.

The implementation makes use of two boolean variables: `exists` and `basicMatch`.

– The boolean variable `exists` is set to true if the method exists in the pointcut declaring class `pointcutClass`. This directly corresponds to the condition m exists in P in the semantics. The value of `exists` is computed once and for all in the constructor. When statically matching a join point, `exists` is tested first. If it is false, the selector directly returns false. Another possibility would be not to capture the `NoMethodException` in the constructor but rather throw an exception at the level of the constructor. In that case [1], it would not be possible to create a call selector for which the method would not exist in the pointcut declaring type: the boolean variable `exists` is not necessary.
– The boolean variable `basicMatch` is set to true if the join point is indeed a method-call join point for the method specified in the pointcut. This requires to compare the name and parameter types specified in the pointcut and the ones obtained from the join point.

If both `exists` and `basicMatch` are true, the last step consists of checking, using the method `asSubClass` of the Java reflection API, that the declaring type of the join point, `jp.getSignature().getDeclaringType()`, is a subclass of the pointcut declaring class `pointcutClass`. This directly corresponds to the condition $J \subseteq P$ in the semantics.

7.3.2.2 Execution

Semantics According again to the AspectJ Programming Guide, *when matching method-execution join points, if the execution pointcut method signature specifies a declaring type, the pointcut will only match methods declared in that type, or methods that override methods declared in or inherited by that type.*

This is illustrated, in the AspectJ Programming Guide, by the application of the pointcut `execution(public void Middle.*())` to the example of Listing 7.8, a variant of Listing 7.5, which is said to capture any method-execution join point for `Sub.m()`. Indeed, the method `m` in `Sub` overrides the definition of `m` inherited by the pointcut declaring type `Middle`.

1. When using the AspectJ Eclipse plugin, a warning is emitted when the method does not exist, but this is not considered as an error.

```java
public class Call extends PrimitiveSelector {
  public Class pointcutClass;
  public String methodName;
  public Class[] parameterTypes;
  public boolean exists;

  public Call(Class pointcutClass, String methodName,
              Class[] parameterTypes)
      throws NullPointerException, SecurityException {

    try {
      pointcutClass.getMethod(methodName, parameterTypes);
      exists = true;
    } catch (NoSuchMethodException) {
      exists = false;
    }
  }

  public boolean staticTest(JoinPoint jp) {
    boolean basicMatch;
    if (exists) {
      boolean basicMatch =
        jp.getKind().equals("method-call")
        && jp.getSignature().getName().equals(methodName)
        && Arrays.equals(((MethodSignature)jp.getSignature()).
            getParameterTypes(), parameterTypes);
      if (basicMatch) {
        try {
          jp.getSignature().getDeclaringType().asSubclass(
              pointcutClass);
          return true;
        } catch (Exception e) {
          return false;
        }
      } else
        return false
    } else
      return false;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return true;
  }
}
```

Listing 7.7: The `Call` selector

```
class Super {
    protected void m() { ... }
}
class Middle extends Super {
}
class Sub extends Middle {
    public void m() { ... }
}
```

Listing 7.8: Example.

```
public class Execution extends PrimitiveSelector {
  ...
  basicMatch = jp.getKind().equals("method-execution");
  ...
}
```

Listing 7.9: The `execution` selector

It is actually interesting to be systematic on this example and check what happens when running `new Sub().m()`, `new Middle().m()`, and `new Super().m()`. As we have just said, the pointcut `execution(public void Middle.*())` captures a join point in the first case. It does not in the second case because, although the method `m` is inherited by `Middle`, it is not overridden in `Middle`. It does not in the third case either because the method `m` of `Super` cannot be inherited from `Middle`. The second case may look surprising if we consider that there is a (virtual) copy of the method `m` in `Middle`. It is less surprising if we look at the execution in terms of dynamic lookup. There is indeed no definition of the method `m` in `Middle` and it is the definition of the method `m` in the superclass that is executed.

To summarize, there are three conditions for a pointcut `execution(`P`.`m`())` to capture a method-execution join point m!()!, where *this* is of dynamic type D:

– $D <: P$;
– m exists in P;
– m is (re)defined in J such that $D <: J <: P$ and not redefined between D and J.

Here, the qualifying type of the join point J is the declaring type of the method m. As $D <: J$ these conditions can then be reformulated as follows:

– $J <: P$, where J is the join-point *qualifying type*;
– m exists in P.

We then get the same conditions as with the pointcut `call` but with a specific definition of the join-point qualifying type.

Implementation The implementation of the execution selector is exactly the same as the implementation of the call selector with the difference that we test if the join point kind is `"method-execution"` (see Listing 7.9). This similarity directly follows from the fact that the semantics of the corresponding pointcuts are exactly same, modulo the definition of the join-point declaring types, and from the fact that we use AspectJ join points.

When calling `getSignature.getDeclaringType()`, we get the static type of the receiver object for method calls and the type which defines the executed method for method executions.

```java
public class Get extends JoinPointSelector {
  public Class fieldType;
  public Class fieldClass;
  public String fieldName;
  public Get(Class fieldType, Class fieldClass, String fieldName) {
    this.fieldType = fieldType;
    this.fieldClass = fieldClass;
    this.fieldName = fieldName;
  }
  public boolean staticTest(JoinPoint jp) {
    if (jp.getKind().equals("field-get")) {
      boolean fn, ft, fdt;
      Signature sig = jp.getSignature();
      fn = sig.getName().equals(fieldName)
         || sig.getName().equals("*");
      fdt = sig.getDeclaringType().equals(fieldClass);
      ft = ((FieldSignature) sig).getFieldType().equals(fieldType);
      return fn && fdt && ft;
    } else
      return false;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return true;
  }
}
```

Listing 7.10: The `get` selector

7.3.3 Field-related pointcuts

7.3.3.1 Get

Semantics The `get(FieldPattern)` primitive pointcut intercepts each join point associated with the access to a field that conforms to the pattern given in the pointcut.

Implementation Listing 7.10 presents the implementation of the `get` selector as a `Get` class. An instance of this class defines three attributes: `fieldName` for the name of the field of interest, `fieldType` for the field type and a `fieldClass` for the defining class of the field. The `staticTest` method test if the join point is of type `"field-get"` then, using reflection determines if the field name, field type and defining class of the current join point match the attributes of the `Get` instance. The `dynamicTest` method always returns `true`.

7.3.3.2 Set

Semantics The `set(FieldPattern)` primitive pointcut intercepts each join point associated with the change of a field that conforms to the pattern given in the pointcut.

Implementation The implementation is similar to the implementation of `get` except that we test if the current join point is of type `"field-set"`.

```
  public boolean staticTest(JoinPoint jp) {
    boolean type = jp.getKind().equals("method-execution");
    boolean className = jp.getSignature().getDeclaringType().equals
        (AspectJAdvice.class);
    boolean methodName = jp.getSignature().getName().equals("
        adviceexecution");
    return type && className && methodName;
  }

  public boolean dynamicTest(JoinPoint jp) {
    return true;
  }
}
```

Listing 7.11: The `adviceexecution` selector

7.3.4 Advice execution-related pointcuts

7.3.4.1 Advice execution

Semantics AspectJ provides the primitive pointcut designator `adviceexecution` to capture advice execution. An example of using this follows.

```
class A {
  public void foo (){
  }
}
aspect MyAspect {
  pointcut p(): execution(* *(..));

 before():  p() {
   new A().foo();
  }
}

aspect YourAspect {
  pointcut myAdvice(): adviceexecution() && within(MyAspect);

  before(): call(* *(..)) && !cflow(myAdvice) {
  // do something
  }
}
```

The advice of `YourAspect` will be executed at every `method-call` join point except if it is in the control flow of the advice execution of `MyAspect`. For example, `new A().foo()` will not be intercepted.

Implementation To implement the pointcut designator `adviceexecution`, we create a class `AdviceExecution` (Listing 7.11), which inherits from `PrimitiveSelector` and we redefine the `staticTest` and `dynamicTest` methods. To intercept the execution of an advice, we must capture the execution of the method `adviceexecution` of `Advice`. The `staticTest` method returns true if the the join point is of kind `"method-execution"` of `Advice.adviceexecution()`.

```java
public class This extends PrimitiveSelector {
  Class pointcutClass;
  public This(Class pointcutClass) {
    this.pointcutClass = pointcutClass;
  }
  public boolean staticTest(JoinPoint jp) {
    return true;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return pointcutClass.isAssignableFrom(jp.getThis().getClass());
  }
}
```

Listing 7.12: The `this` selector implementation

7.3.5 State-based pointcuts

7.3.5.1 This

Semantics In AspectJ, the pointcut designator `this` takes the form of `this`(*Type*). It matches all join points with a `this` object of the specified type. In other words, if `this`(*Type*) will match the join points where the declaring type of the pointcut is the same or a super class of the qualifying type of the join point. According to this semantics, we implement `this` as a `This` class (see Listing 7.12), which inherits from `PrimitiveSelector`.

Implementation The join point selector `This` (see Listing 7.12) determines if the class represented by `pointcutClass` is either the same as, or is a superclass of, the class of `this` at the current join point. It is very simple because we use the capability provided by the AspectJ join points to access the `this` object and to compare its type with the pointcut declaring class. Note that the designator `this`, with the current semantics of AspectJ, deals only with dynamic information (this type). For this reason, the method `staticTest` always returns `true`.

7.3.5.2 Target

Semantics The designator `target` pointcut is similar to the designator `this`, but uses the target of the join point instead of its `this` object.

Implementation The implementation of the designator `target` (see Listing 7.13) is very similar to the implementation of the designator `this` where the only difference is that the method `getThis` is replaced by the method `getTarget` to access the target object.

7.3.5.3 Args

Semantics The `args(Type1, Type2, ..., TypeN)` designator picks out each join point having N arguments of types `Type1`, `Type2`, . . . , `TypeN`, respectively.

Implementation The implementation of the `args` pointcut consists of a class `Args` (see Listing 7.14), which inherits from `PrimitiveSelector`. The method `staticTest` always returns true. The method `dynamicTest`, in turn, determines if each class defined in the array `pointcutClsTab` is either the same as, or is a superclass of, the class of the join point parameters at the same index.

```java
public class Target extends PrimitiveSelector {
  Class pointcutClass;
  public Target(Class pointcutClass) {
    this.pointcutClass = pointcutClass;
  }
  public boolean staticTest(JoinPoint jp) {
    return true;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return pointcutClass.isAssignableFrom(jp.getTarget().getClass()
        );
  }
}
```

Listing 7.13: The `target` selector implementation

```java
public class Args extends PrimitiveSelector {
  Class[] pointcutClsTab;
  public Args(Class[] pointcutClsTab) {
    this.pointcutClsTab = pointcutClsTab;
  }
  public boolean staticTest(JoinPoint jp) {
    return true;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return areSubclasses(pointcutClsTab,typesOfArgs(jp.getArgs()));
  }
  public Class[] typesOfArgs(Object[] objTab) {
    Class[] jpClsTab = new Class[objTab.length];
      for(int i=0; i < objTab.length; i++){
        jpClsTab[i] = objTab.getClass();
      }
    return jpClsTab;
  }
  public boolean areSubclasses(Class[] pointcutClsTab, Class[]
      jpClsTab) {
    boolean result = true;
    for(int i=0; i < pointcutClsTab.length; i++){
        result = result && pointcutClsTab[i].isAssignableFrom(
            jpClsTab[i]);
    }
  }
}
```

Listing 7.14: The `args` selector implementation

```
public abstract class If extends PrimitiveSelector {
  public abstract boolean booleanExpression(JoinPoint jp);
  public boolean staticTest(JoinPoint jp) {
    return true;
  }
  public boolean dynamicTest(JoinPoint jp) {
    return booleanExpression(jp);
  }
}
```

Listing 7.15: The `if` selector implementation

7.3.6 Expression-based pointcuts

7.3.6.1 If

Semantics The pointcut expression `if`(*BooleanExpression*) picks out join points based on a boolean expression *BooleanExpression* which is dynamically evaluated. Within this expression, the `thisJoinPoint` object is available. So one (extremely inefficient) way of picking out any join points would be to use this pointcut with `true` as expression (`if(true)`).

Implementation The implementation of the `if` pointcut designator consists of implementing an abstract method `booleanExpression(JoinPoint)` in the class `If` (see Listing 7.15). A concrete implementation of the method `booleanExpression` is provided for each conditional pointcut expression. For example,

```
if(thisJoinPoint.getKind().equals("call"))
```

is translated into:

```
new If(){
  public boolean booleanExpression(JoinPoint jp) {
    return jp.getKind().equals("call");
  }
}
```

The method `staticTest` always returns `true` and the method `dynamicTest` returns the result of the evaluation of its method `booleanExpression`.

7.4 Aspects

In AspectJ, aspects are not instantiated with new expressions and aspect instances are automatically created to cut across programs.

7.4.1 Aspect declaration

As we saw in the example of Section 7.2, an aspect has to be into a class that inherits from the class `AspectJ`. In the default constructor of the aspect instance, we create the instances of `SelectorAdviceBinding` and we add them to the list bindings of the aspect by calling the method `addSelectorAdviceBinding`, which adds the instance of `SelectorAdviceBinding` given as a parameter.

Each class (inheriting from `AspectJ`) can possess methods and fields that are accessible from advice defined in the default constructor as an AspectJ piece of advice can access fields and methods defined in the aspect.

7.4.2 Aspect instantiation

AspectJ has different types of instantiation policies. We describe the semantics of each policy then we show how we implement it for the AspectJ plugin.

Singleton aspects This policy is the default one. It is implemented directly by having one instance of the class representing the aspect in the `AspectLoader`.

Per-object aspects – `perthis`(*pointcut*) instantiation:

Let us consider an aspect `A` defined with `perthis`(*pointcut*). We use two aspects to implement this policy for the aspect `A`. The first aspect is represented as:

```
public class A extends AspectJ {
  public Object thisObject;
  ..
}
```

The second aspect `PerthisAFactory` is used to manage the instanciation of the aspect `A`:

```
public class PerthisAFactory extends AspectJ {
  List<Object> boundedObjects = new List<Object>();
  ..
}
```

The class `PerthisAFactory` contains the selector/advice binding constructed by the selector corresponding to *pointcut* associated with the following advice:

```
Advice advice1 = new Advice() {
  public Object adviceexecution() {
     Object thisObject = jp.getThis();
    if(!boundedObjects.Contains(thisObject)) {
      A aspect = new A();
      aspect.thisObject = thisObject;
      aspect.deploy();
      boundedObjects.add(thisObject);
      return proceed();
    }
  }
};
```

This advice binds an instance of the aspect `A` to the current object of the join point (matched by *pointcut*) then deploys this aspect instance. The current object is then added to the `boundedObjects` list.

The following `if` selector will select the join point where the current object is bound to the aspect:

```
If instantiationTest = new If() {
  public boolean booleanExpression(JoinPoint jp) {
    return thisObject==jp.getThis();
  }
};
```

The selector `instantiationTest` will be associated to every selector in the aspect `A`. In case an aspect includes several pointcuts, several composed selectors are introduced, one for each pointcut with the `instantiationTest`:

```
ComposedSelector p1 = new ComposedSelector(new And(),
                                      selector1, instantiationTest);
ComposedSelector p2 = new ComposedSelector(new And(),
                                      selector2, instantiationTest);
```

– `pertarget`*(pointcut)* instantiation:

To implement `pertarget`, we proceed as for `perthis` with the difference that we replace `thisObject` by `targetObject` and `getThis()` by `getTarget()`

7.4.3 Aspect extension

The possibility for an aspect to extend a class (or implement an interface) or to be abstract aspect are straightforwardly implemented by applying these extension and implementation relationships to the translation of the aspects.

7.5 Advice precedence

AspectJ uses the statement `declare precedence: Aspect1, Aspect2,..` to define the order of advice execution at shared join points. This statement defines the order at the aspect level. This means that all the pieces of advice of `Aspect1` will be executed before the pieces of advice of `Aspect2`. This feature is implemented by giving this order to the platform which will use it when the order process to order all the `SelectorAdviceBinding` returned by the static test of aspects.

7.6 Transformation of AspectJ syntax to CALI representation

To transform and AspectJ aspect to the aspect representation in our AspectJ prototype, we reuse AspectJ-front of ReflexBorg (see Section 3.2). Instead of transforming aspect code in Reflex, the back-end of our approach transforms aspect code to CALI respresentation.

7.6.1 Implementation

Let us describe how we apply this approach:

AspectJ-front The application of AspectJ-front provides:

- Modular syntax definition for AspectJ 5.0 in SDF. The AspectJ syntax definition is an extension of the modular syntax definition of Java provided by Java-front.
- Hand-crafted pretty-printer for AspectJ. The pretty-printer is an extension of the pretty-printer for Java provided by Java-front.

AspectJ-front can be used to parse AspectJ programs and pretty-print the abstract syntax tree back to an AspectJ source file. The parse result is in the ATerm format (see Figure 7.2a).

Assimilation Now, using Stratego, we write the transformation rules which take the ATerms and produce Java files. These files contain the classes representing the aspects in the AspectJ CALI plugin (Figure 7.2b).

7.7 Conclusion

The AspectJ plugin has the following properties:

Direct extension of CALI The implementation of the AspectJ plugin conform to the methodology for defining AOPLs on top of CALI discussed in the previous chapter.

Syntax extensibility The use of SDF for defining the parser of AspectJ (we will use SDF for other implemented languages) improves the extensibility of implemented languages.

Semantics extensibiliy Our AspectJ plugin makes it easy to play with variants of the AspectJ semantics. For instance, small changes in the implementation give us a variant semantics for some pointcut designators, for aspects scheduling and for deployment. These features are the subject of Chapter 8.

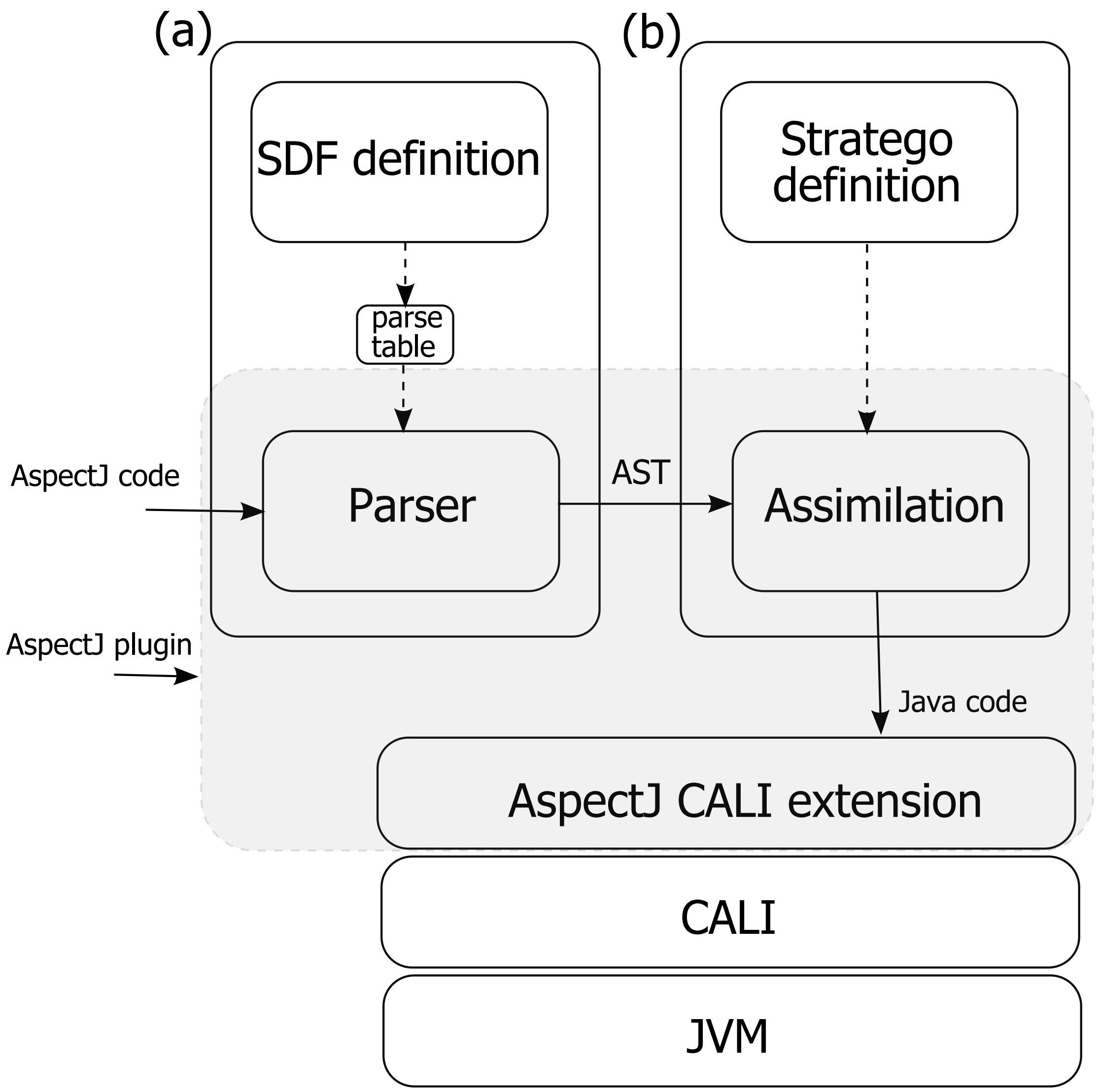

Figure 7.2: The architecture of the AspectJ plugin

Chapter 8

AspectJ Variants with CALI

Contents

There are often some details in the specification of a programming language that were not well specified and later need to *be* gradually evolved and extended. It is difficult to get all the details of the specifications of a programming language right from the start. The language has to be tested on real examples. In order to do so, a light and flexible prototype implementation, able to evolve with the specifications, is required. This chapter shows the ability of CALI and its plugins to support such requirements with respect to AOPLs by presenting two extensions of AspectJ. Section 8.1 presents an extension of AspectJ, called *Dynamic AspectJ* that supports the dynamic scheduling of aspects. Section 8.2 presents an alternative semantics for AspectJ pointcuts (call and execution) and shows that with a few changes in the pointcut implementations we can implement the alternative semantics.

8.1 Dynamic Aspect Scheduling

This contribution was the subject of our paper [14], where it was used to provide a new version of AspectJ, called Dynamic AspectJ. It makes it possible to dynamically schedule aspect execution at a shared join point, including the possibility of canceling aspects. In this chapter, we reconsider the semantics of AspectJ and show how a minor modification of the semantics, giving access to the current aspect group, that is, the pending aspects at the current join point, makes it possible to introduce dynamic scheduling. Section 8.1.1 and Section 8.1.2 present two examples illustrating the benefits of dynamic scheduling. The first example, based on the decorator pattern, presents a case that calls for dynamic deployement and canceling of aspects, whereas the second one, a virus checker, asks for reordering aspects at runtime.

8.1.1 The Decorator Example

The *decorator* (or *wrapper*) pattern is described in [49] as a way to dynamically add (and remove) responsibilities to an object. Its use is illustrated in [48] with a coffee ordering system that takes various types of coffee into account and makes it possible to complement each cup of coffee with various condiments.

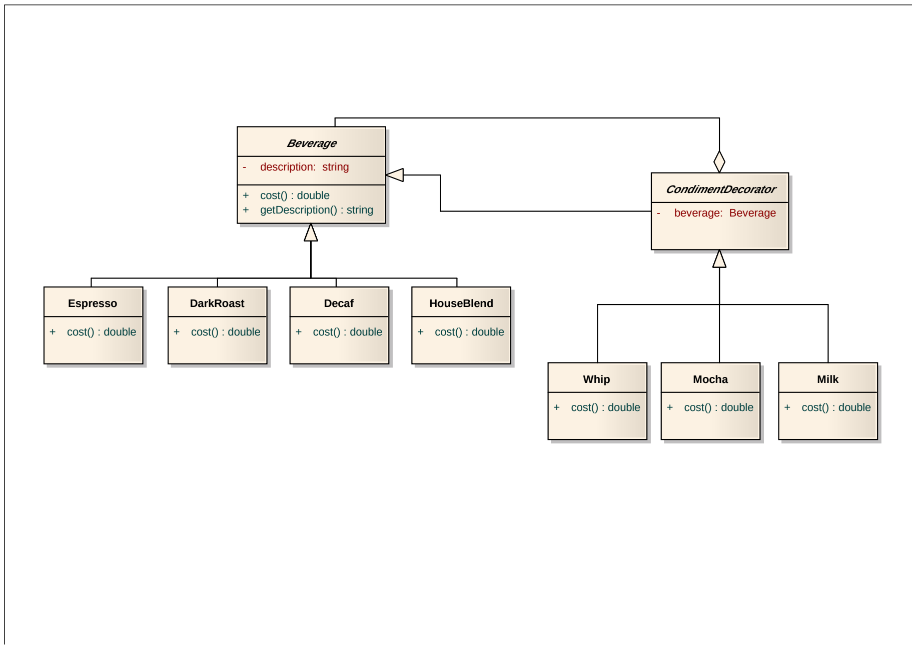

Figure 8.1: Class diagram of the coffee ordering system.

8.1.1.1 Implementation with the Decorator Pattern

A class diagram of the implementation described in [48] is shown in Figure 8.1. The participants of the decorator pattern are a *component* abstract class, `Beverage`, with its concrete subclasses, one per coffee type, and a *decorator* abstract class, `CondimentDecorator`, with its concrete subclasses, one per condiment type. As part of the decorator pattern, the decorator inherits from the component, which means that condiment decorators can be seen as beverages. Concrete decorators also include a reference to the component they decorate. It is then possible to wrap beverages in multiple layers, and build, for instance, an order for a double mocha dark roast coffee topped off with whipped milk:

```
Beverage beverage = new DarkRoast();
beverage = new Mocha(beverage);
beverage = new Mocha(beverage);
beverage = new Whip(beverage);
```

In general, decorators can add new state and new methods to the objects they decorate. Here, the decorators just add behavior in order to update the order price and description. For instance, the definition of the method `cost` of the class `Mocha` looks as follows[1]:

```
public double cost() {
  return .20 + beverage.cost();
}
```

Such decorators are very similar to proxys. As decorators are transparent to the components they decorate, it is very easy to update the individual prices and evolve the system by adding new beverages and condiments.

1. It could have made sense to add an instance variable `cost` in `CondimentDecorator` together with a generic implementation of the method `cost`.

8.1.1.2 Implementation with AspectJ

Adding new state and methods can be done through intertype declarations. Here, we just need to use the pointcut/advice model of AspectJ to add behavior. Instead of using an abstract class `CondimentDecorator`, we can now use an abstract aspect, which defines where in the base classes (the concrete component classes) the behavior has to be added:

```
public abstract aspect CondimentDecorator {
  pointcut cost():
    call (* Beverage.cost());
  pointcut getDescription():
    call (* Beverage.getDescription());
}
```

Concrete aspects can then be used to associate a specific piece of advice to each concrete decorator. For instance, an aspect `Mocha` can be implemented as follows:

```
public aspect Mocha extends CondimentDecorator {
  double around():
    if (Order.hasMocha) && cost() {
      return 0.20 + proceed();
    }
  String around():
    if (Order.hasMocha) && getDescription() {
      return proceed() + ", Mocha";
    }
}
```

Whereas constructors were previously used to build an order, we assume here that the details of the order are available through boolean variables like `Order.hasMocha`. These variables can then be dynamically used by the aspect pointcuts to test whether a condiment must be included or not. A simple (but not so elegant) extension consists of using integers rather than booleans to iterate condiment orders.

Let us now consider the possibility that condiments may run out. If a condiment is not available any longer, the corresponding aspect should not apply. It is of course possible to add to each pointcut of a condiment aspect a condition testing the stock, but this is reintroducing concern tangling at the level of the aspect pointcuts. Using aspect inheritance is not an option as concrete aspects cannot be extended and turning the condiment aspects into abstract aspects does not really help because neither pointcuts nor advices can be directly extended. Implementing advice bodies as method calls and extending these methods does not work either. The point is that it is necessary to decide on whether to proceed or not with the join point but using `proceed` is not allowed outside of advice bodies. A last possibility is to use an aspect of aspect. Here is a (naive) attempt at implementing this idea:

```
public aspect MochaCanceling {
  pointcut mochaCanceling():
    adviceexecution()
      && this(Mocha) && if (!Stock.hasMocha);

  Object around(): mochaCanceling() {
    return null;
  }
}
```

We want to execute the piece of advice when there is no mocha in stock (if we do not execute the advice, the aspect will anyway have no effect). But what shall the piece of advice do? It should not proceed, but if it does not then the advised advice in `Mocha` does not either. This means that, as we shall see in rest of this section, all the remaining pieces of advice, selected when calling the methods `cost` and `getDescription` but not yet executed, will be dropped. As soon as `Mocha` is not the last aspect, the order is incomplete.

8.1.1.3 Discussion

In [51], Hannemann and Kiczales mention that, with aspects, some patterns disappear because they are not needed any longer. It may seem that the decorator is such a pattern but this is not really the case. This has even actually led to fairly complex schemes, such as the one proposed by Monteiro and Fernandes [80], which relies on registering the decorated objects and making the advice check the target object. Additional code is needed for registering and unregistering decorators and a complication is that some pieces of code (`Decorator.aspectof()`) are not accessible outside of the aspect. Another way of looking at the problem is to rather consider that the issue is that it is not possible to dynamically deploy and possibly cancel the decorators that should not apply.

8.1.2 The Virus-Checker Example

Let us consider the scenario presented in [73]. A company has developed a client-server application for file hosting. Customers upload files via the method `send`. On the server side, a virus checker, called via the method `virusCheck`, checks that the received files are virus-free. The issue is then to upgrade the application in a modular way by adding two features, each being implemented as an aspect:

```
public aspect CompressUpload {
  before(File f):
    call(* Client.send(..)) && args(f) {
      zip(f);
    }
  before(File f):
    call(* Server.virusCheck(..)) && args(f) {
      unzip(f);
    }
}

public aspect SecureUpload {
  before(File f):
    call(* Client.send(..)) && args(f){
      encrypt(f);
    }
  before(File f):
    call(* Server.virusCheck(..)) && args(f) {
      decrypt(f);
    }
}
```

The first feature improves performance by compressing the file on the client side before sending it. On the server side, the virus checker has to decompress the file to analyze it. The second feature improves security by encrypting the file on the client side before sending it. On the server side, the file is decrypted before being analyzed by the virus checker.

The two join points corresponding to the calls to the methods `send` and `virusCheck` are shared between the two aspects. The composition order of the aspect has then a fundamental influence on the semantics of the application.

8.1.2.1 AspectJ Implementation

Let us see how to fulfill the requirements set by this scenario with AspectJ. A `declare precedence` statement has to be used in order to make aspect ordering explicit. Let us assume that `SecureUpload` is applied first:

```
declare precedence: SecureUpload, CompressUpload;
```

This declaration is statically applied at each shared join point shadow so that the pieces of advice of `SecureUpload` always precede those of `CompressUpload`. On the client side, files are encrypted, zipped, and finally sent.

– `declare precedence: CompressUpload, SecureUpload`
  zip → *encrypt* → *send.*
– `declare precedence: SecureUpload, CompressUpload`
  encrypt → *zip* → *send.*

The client can upload different type of files like images, text, etc. The goal of the class `CompressUpload` is to improve the performance of file transfer, as a result it makes sense to use different compression algorithms depending on the type of the file compression. For instance, an image compression algorithm is better than a regular compression algorithm to decrease the size of an image file. We suppose that the method `zip(File f)` applies the right algorithm for each file type. For a non-image type, compression works better if the file is encrypted first because encrypted files exhibit certain patterns that can be exploited by compression techniques. The desired behavior is to have a different order depending on the type of the file:

– For image files: *zip* → *encrypt* → *send.*
– For other formats: *encrypt* → *zip* → *send.*

8.1.2.2 Discussion

This example makes clear the interest of ordering the execution of aspects at runtime. An interesting alternative solution, not resorting to dynamic scheduling, has been suggested in [73, 74]. It consists of a domain-specific declarative aspect composition language for composing aspects but it deviates from the standard structure of AspectJ aspects.

8.1.3 Scheduling in AspectJ

In Section 6.2, we have discussed a modified version of the CASB to be used as the semantics of CALI. Now we are going to present an extension of this semantics in order to add dynamic scheduling to AspectJ. In AspectJ, the statically matching aspects are statically ordered. It is then only possible, either to skip a given aspect through dynamic matching, or to skip all the aspects and the join point by not executing `proceed` in an advice.

8.1.4 Scheduling in Dynamic AspectJ

As previously discussed, one possibility to give more control to the user would be to give her/him support for building her/his own α function. But, although we think that it is an interesting path to pursue, we consider here a more direct approach, suggested by the rules described in Section 6.2. In AspectJ, there is already an explicit *aspect-level* instruction, `proceed` that can be used within a piece of advice to manipulate, in a controlled way, the current aspect group. Why not add some new instructions making it possible to skip and order aspects as required by our motivating examples? The candidate instructions are `skipall`, `skip` n, `skipfirst` (a simple alias for `skip` 1) and `setfirst` n. The semantics of these instructions is described by these new rules:

$$\textsc{SkipAll}\ \frac{}{(\texttt{skipall} : C, \Sigma, (\Phi :: [i]) : P) \rightarrow (C, \Sigma, [i] : P)}$$

$$\textsc{Skip}\ \frac{\Phi' = \Phi - \{\phi_n\}}{(\texttt{skip}\ n : C, \Sigma, \Phi : P) \rightarrow (C, \Sigma, \Phi' : P)}$$

$$\textsc{SetFirst}\ \frac{\Phi' = \Phi - \{\phi_n\}}{(\texttt{setfirst}\ n : C, \Sigma, \Phi : P) \rightarrow (C, \Sigma, (\phi_n : \Phi') : P)}$$

The instruction `skipall` removes all the aspects from the current aspect group. As the current join point is kept in the aspect group, a following execution of the instruction `proceed` will skip these aspects but still execute the join point. This has to be compared with the execution of a piece of advice that does not proceed: all the aspects are skipped, together with the join point.

The instruction `skip` n removes the n^{th} aspect from the current group (with `skipfirst` removing the first aspect).

The instruction `setfirst` promotes the n^{th} aspect from the current aspect group as the first.

Note that our proposal is fairly conservative in that it does not make it possible to introduce new aspects. Our intuition is that, without this restriction, mastering the complexity of aspect programs may become very difficult.

8.1.5 Dealing with Aspect Groups

```
public class AspectGroup {
  List<Phi> phis;
  public void skipAll(){ ... }
  public boolean skip(String name){ ... }
  public void skipFirst(){ ... }
  public boolean setFirst(String name){ ... }
  public boolean member(String name){ ... }
}
```

Listing 8.1: The class `AspectGroup`

In Chapter 6, aspect groups could not be explicitly manipulated. In the following, we show how they can be reifed as instance of a class `AspectGroup` (see Listing 8.1) implementing as methods the instructions previously defined. On the first occurrence of a join point and before executing the first aspect, the remaining aspects, together with the join point, are now packaged into an instance of a class `AspectGroup` as a list of `Phi` instances.

Note that removing an aspect from an aspect group, for instance, with the method `skip` is different from undeploying this aspect. It will affect the behavior at the current join point but will not affect any other join point.

Within an advice, the current aspect group can be accessed through the instance variable `thisAspectGroup`. This is similar to the use of the instance variable `thisJoinPoint` to access the current join point. It is then possible to write an aspect such as:

```
public aspect SkipAllAtCalls {

  Object around(): call(* *.*){
    thisAspectGroup.skipAllAspects();
    System.out.println("all aspects skipped");
    proceed();
  }
}
```

As soon as the aspect is scheduled, all the remaining aspects at the call join point are skipped and the current join point is executed.

8.1.6 Revisiting the Motivating Examples

This section revisits our motivating examples and shows how they can be elegantly implemented using Dynamic AspectJ.

8.1.6.1 The Decorator Example

Thanks to a combination of dynamic deployment and dynamic scheduling, we can now implement each condiment straightforwardly:

```
public aspect Mocha extends CondimentDecorator{
  double around(): cost() {
    return 0.20 + proceed();
  }
  String around(): getDescription() {
    return proceed() + ", Mocha";
  }
}
```

Dealing with the stock concern can also be implemented in a modular way by scheduling the aspect `MochaAvailability` shown in plain AspectJ syntax in Listing 8.2.

```
public aspect MochaAvailability {
  Object around(): cost() {
  if (!Store.contains("Mocha"))
    thisAspectGroup.skipAspect("Mocha");
  else
    return proceed();
  }
}
```

Listing 8.2: The aspect `MochaAvailability`

8.1.6.2 The Virus-Checker Example

The composition of the two aspects `CompressUpload` and `SecureUpload` can be addressed by the aspect shown in Listing 8.3, again in plain AspectJ syntax.

The use of dynamic (re)scheduling allowed by aspect groups makes it possible to contextually schedule first either the aspect `CompressUpload` or `SecureUpload` depending on the file type.

```
public aspect SecureCompression {
  before(File f):
   call(* Client.send(..)) && args(f) {
    if(f.getType().isImage ())
      thisAspectGroup.setFirst("CompressUpload");
    else
      thisAspectGroup.setFirst("SecureUpload");
   }
  before(File f):
   call(* Server.virusCheck(..)) && args(f) {
    if(f.getType().isImage ())
      thisAspectGroup.setFirst("SecureUpload");
    else
      thisAspectGroup.setFirst("CompressUpload");
   }
}
```

Listing 8.3: The aspect `SecureCompression`

8.2 Alternative semantics for AspectJ pointcuts

In Chapter 7, we have described the semantics of the pointcuts `call` and `execution` in the current AspectJ version (1.6) in terms of relation between the *declaring type* of the pointcut and the *qualifying type* of the join point. In the following, we are going to discuss this semantics and to describe alternative semantics for `call` and `execution` and how this could be implemented using small changes in the implementations of these pointcuts described in the previous chapter. This contribution was the subject of our paper [13].

8.2.1 Discussion

8.2.1.1 Relating the semantics of the pointcuts `call` and `execution`

With a proper definition of the qualifying type of the method signatures of the join points, we get syntactically homogeneous definitions of the pointcuts `call` and `execution`. Still, the definition of the qualifying type of an execution join point, though in line with dynamic lookup, is tricky and leads to the source of surprise mentioned before, originally discussed by [18], in the context of an earlier semantics, and rediscovered by [16] while precisely defining the semantics of static pointcuts in AspectJ as datalog queries. Basically, two pointcuts `call` and `execution` using the same declaring type do not always capture both the call join point and its related execution join point.

Indeed, if we consider, in the context of Listing 7.8, the pointcut `call(public void Middle.*())`, a join point is captured for both `new Sub().m()` and `new Middle().m()`. As a result, in the presence of an aspect comprising both the pointcuts `call` and `execution`, a call join point followed by an execution join point is captured in the first case whereas, in the second case, only a call join point is captured.

Such a discrepancy does not occur in the alternative semantics proposed in [18]. These semantics were proposed as a reaction to the earlier semantics of AspectJ 1.1.1. The difference with the current semantics (AspectJ 1.6) was that the method of interest had to be (re)defined in the declaring type of both the pointcuts `call` and `execution`, and the qualifying type of an execution join point was simply the dynamic type of the current object (the receiver of the call).

Basically, the proposal of [18] consists of:

– always considering that the method of interest *exists* in the pointcut declaring type and

– defining the qualifying type of the call join points as either the static type of the receiver or its dynamic type, the qualifying type of the execution join points remaining the dynamic type of the receiver.

The first point fixes the surprise that the pointcut `call(void Middle.f())` did not capture the call `new Sub().f()` in Listing 7.5. The second point defines two semantics, a *static* semantics or a *dynamic* semantics depending on the definition of the qualifying type of the call join points.

The *static* semantics corresponds to the current semantics of AspectJ with respect to method calls, but method executions are still handled differently.

8.2.1.2 Semantics of the pointcut `call`

An argument against using the *dynamic* semantics, which has the advantage of being simpler to explain and reason about as only dynamic types have to be considered in the join points, is that it is less efficient and that using another primitive pointcut, `target`, makes it possible to get this semantics anyway. The efficiency issue comes from the fact that in the case of the static semantics, the matching condition $J <: P \wedge m$ exists in P can be fully evaluated statically as J is a static type. There is no need for a residue. As for using the pointcut `target`, the idea is that this pointcut makes it possible to select call join points based on the *dynamic* type of the callee. For instance, in our previous example (Listing 7.6), it is still possible to capture the call to the method `run` using `call(void Runnable.run()) && target(Service)`.

The authors of [18] note that the static semantics of `call(`$P.m$`()) && target(`P`)` is slightly different from the dynamic semantics of `call(`$P.m$`())`. Indeed, if D is the dynamic type of the join point and S its static type, we have, in the first case, the condition $S <: P \wedge D <: P$, where the first conjunct comes from the pointcut `call` and the second from the pointcut `target`. As by definition $D <: S$, this amounts to $S <: P$. This has to be compared with the condition required by the dynamic semantics: $D <: P$. Conversely, if the dynamic semantics were chosen, it could make sense to change the semantics of the pointcut `target` so that it deals with static types.

8.2.1.3 Semantics of the pointcut `execution`

We have seen that a pointcut `execution(`P`.`m`())` does not capture a method-execution join point `m()`, where the receiver is of dynamic type J, if m is not (re)defined in the hierarchy between P and J. This may look surprising but actually corresponds to the lookup semantics. In practice, if m is only defined in a superclass of P, there is no bytecode that can be instrumented in order to implement the pointcut and transfer control to the advice if necessary apart from the bytecode in the superclass. But this would then slow down the execution of the method with some additional tests on the type of `this` to distinguish the cases when the pointcut should apply and when it should not. An alternative would be to add bytecode for the missing methods at weave time. For instance, let us consider the example in Listing 7.8. If we redefine `m` in `Middle`: `protected void m() super.m()`, then `execution(public void Middle.*())` captures a join point when executing `new Middle().m()`. Such a redefinition could be done at weave time at the bytecode level. One would then get the semantics of [18].

8.2.1.4 Existence of the method in the declaring type

At first sight, the necessity of the condition m exists in P is also arguable. As we have previously seen in Chapter 7 when considering Listing 7.5, without this condition, the pointcut `call(void Middle.g())` would then select a call `new Sub().g()`. This has the drawback that the semantics of the pointcut `call` is not purely static any longer (which is of course not an issue in the context of the dynamic semantics of [18]). But the main point is that this semantics can anyway be already obtained by using, instead of a type P, the *subtype pattern* P`+`, which stands for P or any of its subtype.

```
public boolean staticTest(JoinPoint jp) {
  boolean basicMatch;
  if (exists) {
    ...
    return basicMatch;
  } else
    return false;
}
public boolean dynamicTest(JoinPoint jp) {
  try {
    jp.getTarget().getClass().asSubclass(pointcutClass);
    return true;
  } catch (Exception e) {
    return false;
  }
}
```

Listing 8.4: The *dynamic* `Call` selector

8.2.1.5 Summary

The semantics of the pointcuts `call` and `execution` is not easy to grasp. Matching uses the static type of the caller in the pointcuts `call` and the dynamic type of the callee in the pointcuts `execution`. Understanding the semantics of the pointcuts `execution` requires to reason about the behavior of dynamic lookup and is counter-intuitive when a pointcut `execution(`P`.`m`())` does not capture an execution join point whereas the corresponding call join point is captured by the pointcut `call(`P`.`m`())`.

Alternative semantics that seem easier to grasp have been proposed. In particular, the dynamic semantics of [18] looks attractive. Matching is always based on dynamic types and the simple condition that the method of interest should exist in the type of interest is used for both the pointcuts `call` and `execution`.

Switching to such a semantics may however require a lot of work in the current compiler-based implementations of AspectJ and result in some performance overhead, whereas the exact benefits have not yet been fully assessed. It looks interesting but does it really help in practice? Therefore, it may be worth experimenting with this approach in a more agile environment. In the following sections, we show how we can, by switching to an interpreter-based approach, easily implement the current semantics of the pointcuts `call` and `execution`, and then switch to the static and dynamic semantics of [18] with minimal changes.

8.2.2 Implementation of Alternative Semantics

In the following, we show the minor changes that are necessary to the previous implementation in order to get the dynamic semantics of Barzilay *et al.* [18].

8.2.2.1 Call with dynamic semantics

The conditions are the same as with the static semantics except that we need to replace the declaring type of the join point by its dynamic type, that is `jp.getSignature().getDeclaringType()` by `jp.getTarget().getClass()`. We use again the facilities provided by the AspectJ join points.

Also, as the dynamic type is, as its name indicates, not static, related computations should be moved from the method `staticTest` to the method `dynamicTest`.

Starting from Listing 7.7 and applying these changes, we get Listing 8.4.

8.2.2.2 Execution with dynamic semantics

The principles are the same as before. The differences are that, when computing `basicMatch` `"method-call"` should be replaced by `"method-execution"`, and that the dynamic type of the join point is accessed through `jp.getThis()` instead of `jp.getTarget()`.

8.3 Conclusion

In this chapter, we have discussed the ability of CALI to provide flexible AOPL prototypes. We have considered the AspectJ plugin as running example and we have shown how it is simple to provide two extensions, the first for scheduling aspects at runtime, the second presents an alternative semantics for AspectJ pointcuts (call and execution). Regarding dynamic scheduling, we have described the static scheduling semantics of AspectJ and extended it with the concept of aspect group, the aspects scheduled for the current join point, with the opportunity to access them from the advice body. We have then given all the elements needed to control this group during runtime. Regarding the alternative semantics, we have formally discussed a more dynamic semantics for call and execution designators and we have shown how this semantics can be easily implemented with few changes in the implementation of AspectJ plugin.

Chapter 9

EAOP and DSLs plugins

Contents

The goal of CALI is to facilitate prototyping and composing AOPLs. In Chapters 7 and 8, we have seen how to use CALI to prototype AspectJ and variants. However, we need to implement more AOPLs with CALI in order to, on the one hand, further validate our prototype with mechanisms different from the AspectJ ones and, on the other hand, have in hand several AOPL implementations when exploring the CALI composition feature in the next chapter. Section 9.1 presents the implementation of Event-based AOP (EAOP) for Java, where aspects relate sequences of events. Section 9.2 presents a Domain-Specific Aspect Language to implement the decorator pattern. Section 9.3 prototypes an experimental language to *memoize* method return values. Section 9.4 presents the implementation of COOL, a Domain-Specific Aspect Language for modularizing thread synchronization aspects.

9.1 EAOP

The implementation of EAOP on top of CALI consists of starting from the EAOP conceptual model described in Chapter 2 and deducing a concrete model that can be (easily) implemented by using the API provided by CALI and by reusing the implementation of AspectJ pointcuts described in Chapter 7.

9.1.1 EAOP model

Let us reconsider the EAOP aspect used in Chapter 2:

$$\mu a.(login \triangleright proceed; \mu b.(update \triangleright skip; b) \Box logout \triangleright proceed; a)$$

We have seen that this EAOP aspect can be directly represented using Extended FSP as a state machine where $C \triangleright I$ is attached to each transition. Figure 9.1a represents this aspect according to this conceptual model. This aspect starts in a state (`Server`) waiting for `login` join points. The detection of a login join point and the execution of the corresponding advice transfers the aspect to the second state (`Session`). The second state corresponds to a choice between two crosscuts.

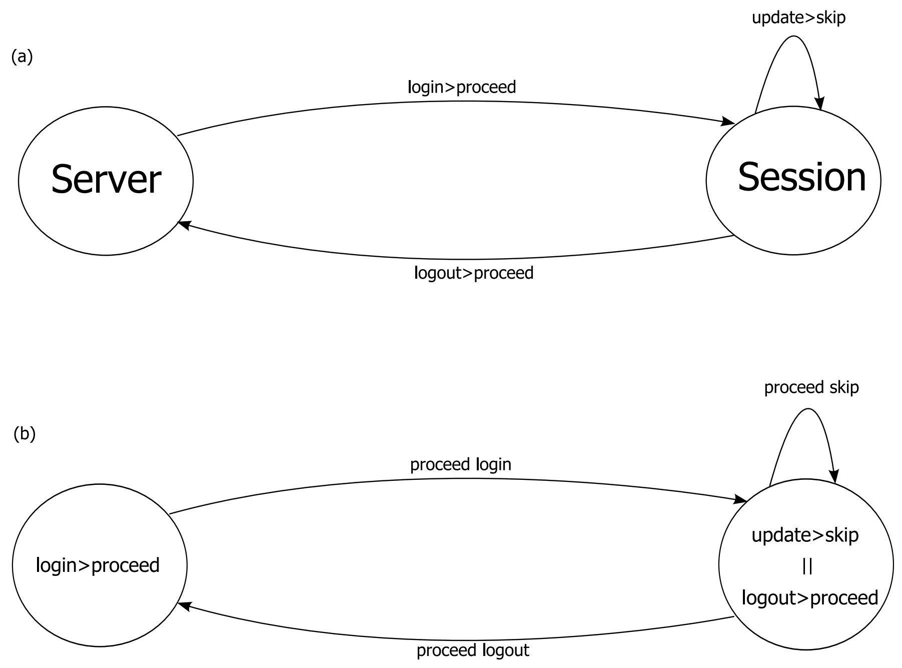

Figure 9.1: Comparison between the 2 models: a) Initial conceptual model, b) Modified model

The occurrence of a logout join point returns the aspect state to the first one while an update join point is canceled without changing the aspect state.

We propose an alternative model of the conceptual one but before starting to describe it, we should note that the state of the aspect evolves to the next state after the execution of the advice and not just after join point matching. In addition, the aspect should have access to the all crosscuts corresponding to each state in order to iterate the evaluation of the current join point over them. For this reason, in the modified model, we attach all the informations about the continuation of the aspect to the state. This means that the simple $C \triangleright I$ or the choice between the list of $C \triangleright I$ that have to intercept the join point will be attached to the state instead of attaching a $C \triangleright I$ to each transition. The transition between states is represented by the event corresponding to the end of advice execution. Figure 9.1b) shows the representation of the consistency aspect with the modified model.

9.1.2 Basis of EAOP implementation

The basic rule $C \triangleright I$ can be straightforwardly implemented by a selector/advice binding: a crosscut is implemented as a selector and an insert as an advice. A powerful benefits of our approach is that we can reuse the pointcuts of AspectJ implemented in Chapter 7. A key point in the implementation of EAOP is the representation of the aspect state. Each state should contain the list of selector/advice bindings corresponding to the list of crosscuts that are attached to the choice operator.

9.1.3 Implementation using Dynamic AspectJ

According to the alternative EAOP model, each state contains a list of selector/advice bindings and could be straightforwardly represented by an AspectJ aspect. The general translation scheme starts from the labelled transition system corresponding to the Extended FSP version of the EAOP aspect (edges are labelled with inserts). Each state of the system corresponds to an AspectJ-like "atomic" aspect singleton instance with as many pairs (selector, advice) as they are outgoing edges/inserts $C \triangleright I$. The selector implements the crosscut C and the piece of advice, the insert I. Once the insert has been executed, the piece of advice is also responsible for implementing, unless the state does not change, the transition from the source state to the destination state by undeploying the current aspect instance, corresponding to the source state of the edge, and deploying the aspect instance corresponding to its destination state. The execution is initialized by deploying the aspect instance associated to the start state of the system.

Listings 9.1 and 9.2 shows two aspects implemented using Dynamic AspectJ. `ConsistencyOutOfSession` represents the first state of the aspect `Consistency`. It contains one selector/advice binding. The selector matches any call of the method `login` while the corresponding advice proceeds, undeploys the aspect `ConsistencyOutOfSession` and deploys `ConsistencyInSession`, which corresponds to the second state. `ConsistencyInSession` contains two selector/advice bindings: the first one is responsible of canceling the call of the method `update`, the second one changes the aspect state by undeploying the aspect `ConsistencyInSession` and redeploying the aspect `ConsistencyOutOfSession`.

9.1.4 Dedicated EAOP implementation

The representation of the aspect states as Dynamic AspectJ aspects may cause problems. Let us consider the case of composing EAOP with AspectJ. We would have a general composition configuration, which consists of executing EAOP aspect before all AspectJ ones. The (state) aspects could interacts with other (normal) Dynamic AspectJ aspects existing in the environment. For example, the method `order` of the platform may schedule one of the state aspects at the end of the proceed stack. For this reason, we present here another EAOP implementation, where the aspect state is reified.

```
class ConsistencyOutOfSession extends AspectJ {
  static ConsistencyOutOfSession aspect = new
     ConsistencyOutOfSession();
  ConsistencyOutOfSession(){
    Class[] parameterTypes = {};
    JoinPointSelector login =
            new Call(Server.class,"login", void.class,
                parameterTypes);
    Advice<Object> loginAdvice =
            new Advice<Object>(){
              public Object adviceexecution(){
                System.out.println("=> out-of-session login
                     detected");
                Object result = proceed(); // dummy returned object
                aspect.undeploy();
                ConsistencyInSession.aspect.deploy();
                return result;
              }
            };
    addSelectorAdviceBinding(new SelectorAdviceBinding(login,
        loginAdvice));
  }
}
```

Listing 9.1: Implementing the state `Server` using Dynamic AspectJ

According to the API provided by CALI, an EAOP aspect must implement the interface `Aspect` and must define the method `staticTest`. The method `staticTest` in EAOP should return an instance of the class `Phi`. In fact, this instance is returned by one of the selector/advice bindings of the current state of aspect. The implementation of a state is described below.

Requirement 6 *An EAOP aspect evolves to the next state after the execution of the piece of advice. For this reason, we create the notion of* ***`advice-state`*** *which associates an advice with a state (the next state of the aspect). This feature is implemented using the class* ***`AdviceState`*** *(see Listing 9.3).*

We can see that at a specific moment, the state of the aspect can be associated to a list of crosscuts with a choice between them. This leads us to Proposition 7.

Requirement 7 *Each state is associated with a list of selector/advice bindings. When evaluating a state with a join point, the state iterates the evaluation of this join point over its selector/advice bindings. The advice of the first selector/advice binding matching the join point will be executed.*

Note that the matching of a join point will be done in two steps. When testing the static part of the join point, the state creates a list of selector/advice bindings that statically match the join point. The state is returned to the `Platform` encapsulated within an instance of `EAOPPhi` (see Listing 9.7). This instance is pushed onto the proceed stack then evaluated in the same way an `Phi` instances returned by an AspectJ aspect. The interface `State` (Listing 9.4) possesses two main methods: `staticTest` and `dynamicTest`. The method `staticTest` is called to determine the list of selector/advice bindings that statically match the join point by adding them to a list used during the dynamic test. The method `dynamicTest` is called when evaluating the instance of `EAOPPhi` in the `Platform`. The execution of the method `dynamicTest` evaluates the advice corresponding to the *first* selector/advice binding that dynamically matches the current join point. The class `StateImpl` 9.5 provides an implementation of the interface `State`.

```
class ConsistencyInSession extends AspectJ {
  static ConsistencyInSession aspect = new ConsistencyInSession();
  ConsistencyInSession(){
    Class[] parameterTypes = {};
    JoinPointSelector update =
            new Call(Server.class,"update", void.class,
               parameterTypes);
    JoinPointSelector logout =
            new Call(Server.class,"logout", void.class,
               parameterTypes);
    Advice<Object> updateAdvice =
            new Advice<Object>(){
              public Object adviceexecution(){
                System.out.println(" =>in-session update
                  detected");
                return null;
              }
            };
    Advice<Object> logoutAdvice =
            new Advice<Object>(){
              public Object adviceexecution(){
                System.out.println("=> in-session logout detected")
                    ;
                Object result = proceed(); // dummy returned object
                ConsistencyOutOfSession.aspect.deploy();
                aspect.undeploy();
                return result;
              }
            };

    addSelectorAdviceBinding(new SelectorAdviceBinding(update,
       updateAdvice));
    addSelectorAdviceBinding(new SelectorAdviceBinding(logout,
       logoutAdvice));
  }
}
```

Listing 9.2: Implementing the state `Session` using Dynamic AspectJ

```java
public class AdviceState extends Advice {
  private State nextState;
  private Advice advice;
  EAOPAspect aspect;
  public State getNextState() {
    return nextState;
  }
  public AdviceState(Advice advice, State nextState) {
    this.nextState = nextState;
    this.advice = advice;
  }
  public Object eval(JoinPoint jp) {
    Object result = advice.eval(jp);
    aspect.setCurrentState(nextState);
    return result;
  }
}
```

Listing 9.3: The class `AdviceState`

```java
public interface State {
  boolean staticTest(JoinPoint jp);
  AdviceState dynamicTest(JoinPoint jp);
  List<SelectorAdviceBinding> getBinds();
}
```

Listing 9.4: The interface `State`

Requirement 8 *The implementation of an EAOP aspect on top of CALI consists of defining a class that implements the interface `Aspect` and contains a current state. The state transition takes place after the execution of the piece of advice.*

According to Propositions 7,8, we implement the class `EAOPAspect` (Listing 9.6) where the basic constituents are the attribute `currentState` and the method `staticTest`.

When the platform calls this method, the method `staticTest` of the current state is called. As we said before, a list of selector/advice bindings that statically match the current join point is prepared in the state, to be used when the join point is dynamically tested.

At each state, the aspect should not intercept any join points until the end of the advice execution. This behavior is implemented using the variable `available` to block the matching of join point during aspect execution. This is done by setting as `false` the value of this variable when matching a join point and as `true` once the execution of the piece of advice is finished.

The method `staticTest` of `EAOPAspect` (Listing 9.7) returns an instance of `EAOPPhi` which inherits from `Phi` to represents the ϕ returned by an aspect. An `EAOPAspect` returns a list that contains one instance of `EAOPPhi`. The evaluation of `EAOPPhi` (dynamic test) consists of dynamically evaluating the current state associated with this instance of `EAOPPhi`. The dynamic evaluation consists of calling the method `dynamicTest` of `StateImpl`. The method `StateImpl.dynamicTest` tests if one of the `SelectorAdviceBindig` that statically match the join point, dynamically matches it. The result is an instance of `AdviceState` which being evaluated. When the advice terminates its execution, it must inform the aspect to change the state to the next state referenced by the `AdviceState` instance. It is done by calling the method `setCurrentState` of the class `EAOPAspect`.

```
public class StateImpl implements State {
  List<SelectorAdviceBinding> bindings;
  List<SelectorAdviceBinding> statictestlist;
  String name;

  public StateImpl(String name, List<SelectorAdviceBinding>
     bindings){
    this.name = name;
    this.bindings=bindings;
  }
  public void addBindings(List<SelectorAdviceBinding> bindings){
    this.bindings.addAll(bindings);
  }
  public boolean staticTest(JoinPoint jp){
    statictestlist = new ArrayList<SelectorAdviceBinding>();
    for(SelectorAdviceBinding o :bindings){
      if(o.staticTest(jp)) statictestlist.add(o);
    }
    return !statictestlist.isEmpty();
  }
  public AdviceState dynamicTest(JoinPoint jp){
    int i=0;
    AdviceState adv = null;
    do {
      adv=(AdviceState)statictestlist.get(i).dynamicTest(jp);
      i++;
    } while(i<statictestlist.size() && adv==null);
    return adv;
  }
  public String getName() {
    return name;
  }
  public void addBinding(SelectorAdviceBinding binding) {
    bindings.add(binding);
  }
  public List<SelectorAdviceBinding> getBindings() {
    return bindings;
  }
}
```

Listing 9.5: The class `StateImpl`

```java
public class EAOPAspect extends Aspect {
  public State currentState;
  public State nextState;
  boolean available = true;

  public void setCurrentState(State state) {
    currentState = state;
  }
  public State getCurrentState() {
    return currentState;
  }
  public List<Phi> staticTest(JoinPoint jp) {
    List<Phi> list = new ArrayList<Phi>();
    if (available && this.currentState.staticTest(jp)) {
      list.add(new EAOPPhi(this, currentState));
    }
    return list;
  }
}
```

Listing 9.6: The `EAOPAspect` implementation

```java
public class EAOPPhi extends Phi {
  State currentState;
  EAOPAspect aspect;
  public EAOPPhi(EAOPAspect aspect, State state) {
    this.aspect= aspect;
    this.currentState = state;
  }
  public Object eval(JoinPoint jp) {
    AdviceState obj = currentState.dynamicTest(jp);
    if (obj != null) {
      obj.aspect = aspect;
    }
    return obj.eval(jp);
  }
}
```

Listing 9.7: The `EAOPPhi` class

```
class Consistency extends EAOPAspect {
  Call login = new Call(Server.class, "login", void.class, new
     Class[0]);
  Call update = new Call(Server.class, "update", void.class, new
     Class[0]);
  Call logout = new Call(Server.class, "logout", void.class, new
     Class[0]);
  State Server = new StateImpl("Server");
  State Session = new StateImpl("Session");
  Consistency() {
    // Server
    {..}
    // Session
    {..}
      currentState = Server;
    }
}
```

Listing 9.8: The `Consistency` EAOP aspect

9.1.4.1 The example revisited

Listing 9.9 shows the translation of the consistency aspect into our implementation. It is represented by the class `Consistency` that inherits from `EAOPAspect`. It contains three `Call` selectors to implement the three pointcuts `login`, `update` and `logout`. Listing 9.9 shows the part of the constructor that defines the state `Session`.

9.2 Decorator

In this section, we present a domain specific aspect language optimized for enforcing the decorator pattern. This language is described in [53]. We first introduce a running example that we will use to demonstrate how this language is implemented using CALI.

9.2.1 A DSAL to enforce the decorator pattern

The application consists of a simple word processor application. This example is also described in [53]. Figure 9.2 shows the class diagram of this application where a class `Document` defines some methods to modify a document (`addLine()` and `setContent()`), a method to obtain the document content (`getContent()`), as well as a method that counts the current number of words in the document (`wordCount()`).

In order to add autosave behavior to the word processing application, the decorator pattern [49] can be used by defining a class `AutoSaveDocument`. According to the decorator pattern, this class must implement the same methods as the class `Document` (Figure 9.3) but adds the behavior to save any changes made to the document, before forwarding method calls to the original document object. Listing 9.10 shows how we could wrongly use this decorator class. The class `AutoSaveDocument` (decorator) is instantiated with an instance of the class `Document` (decarotee). The method `addLine` of `AutoSaveDocument` executes the behavior related to the autosaving after calling the method `addLine` on its `decoratee`.

Indeed, this code uncovers two issues:

1. The decorated object (`decoratee`) `doc` is still accessible and the behavior of the decorator could not be invoked.

```
    ..
    Class[] parameterTypes = {};
    JoinPointSelector update =
            new Call(Server.class,"update", void.class,
               parameterTypes);
    JoinPointSelector logout =
            new Call(Server.class,"logout", void.class,
               parameterTypes);
    Advice<Object> updateAdvice =
            new Advice<Object>(){
              public Object adviceexecution(){
                System.out.println(" =>in-session update
                  detected");
                return null;
              }
            };
    AdviceState updateTransition = new AdviceState (updateAdvice,
       Session);
    Advice<Object> logoutAdvice =
            new Advice<Object>(){
              public Object adviceexecution(){
                System.out.println("=> in-session logout detected")
                    ;
                Object result = proceed(); // dummy returned object
                ConsistencyOutOfSession.aspect.deploy();
                aspect.undeploy();
                return result;
              }
            };
    AdviceState logoutTransition = new AdviceState (logoutAdvice,
       Server);
    addSelectorAdviceBinding(new SelectorAdviceBinding(update,
       updateTransition));
    addSelectorAdviceBinding(new SelectorAdviceBinding(logout,
       logoutTransition));
```

Listing 9.9: The implementation of the state `Session` of EAOP aspect

```
public class WordProcessor {
  Document doc;
  AutoSaveDocument autoSaveDoc;
  public void testAutoSave() {
    doc = new Document();
    autoSaveDoc = new AutoSaveDocument(doc);
    autoSaveDoc.addLine("AutoSaved"); // autosaving takes place
    doc.addLine("Not AutoSaved"); // autosaving does not takes
       place
  }
}
```

Listing 9.10: The `WordProcessor` class

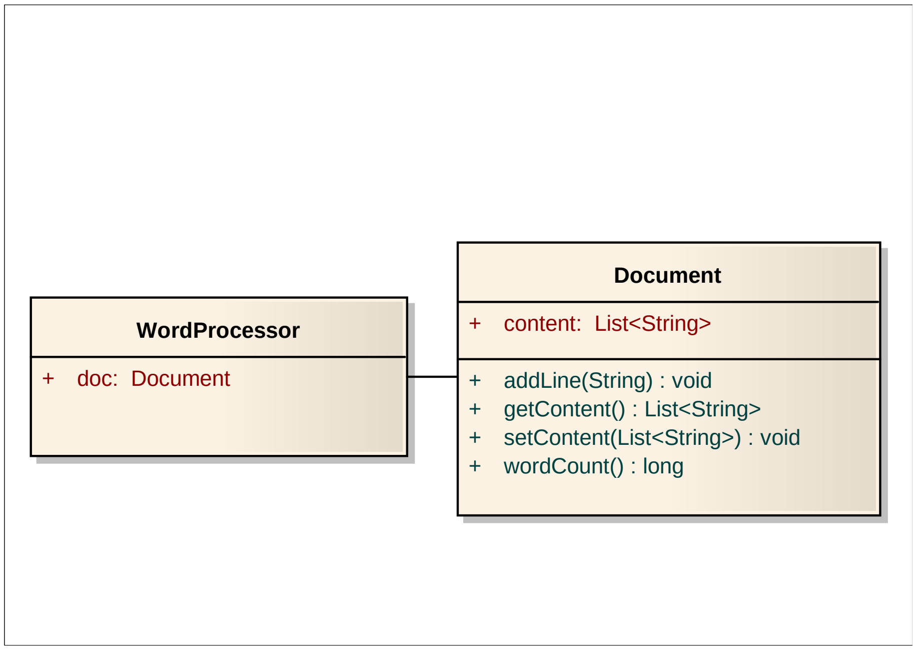

Figure 9.2: Class diagram of `WordProcessor`

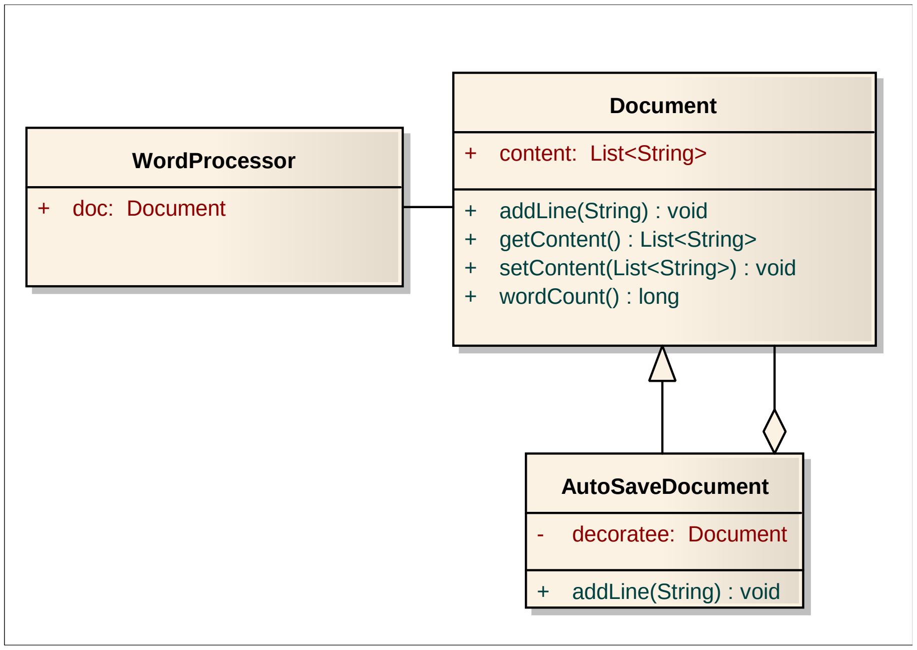

Figure 9.3: Decorator pattern example

```
decorate: Document -> AutoSaveDocument
```

Listing 9.11: Enforce the decorator pattern on the class `Document`

```
public class Decorator extends AspectJ {
  public Decorator(List<SelectorAdviceBinding> list) {
    list = new ArrayList<SelectorAdviceBinding>();
    this.list = list;
  }
}
```

Listing 9.12: The decorator aspect

2. Part of the code dealing with the decorator pattern is visible in the client (class `WordProcessor` in this example). It is not fully modularized because we had to add a new field (`autoSaveDoc` of type `AutoSaveDocument`).

To enforce the decorator pattern, once a decorator is associated with a decoratee, all subsequent calls should be made to the decorator. Havinga *et al.* [53] propose a new DSAL to enforce the decorator pattern and to fully separate the code dealing with the decorator pattern.

The grammar of the language is defined as follows:

Aspect ::= `decorate:` *ClassIdentifier* `->` *ClassIdentifier*

An aspect declaration starts by the keyword `decorate` followed by two identifiers *ClassIdentifier*. The first identifier represents the decoratee class and the second identifier represents the decorator class. Listing 9.11 shows an example of a program written in the DSAL that contains one declaration to enforce that `AutoSaveDocument` decorate `Document`.

9.2.2 Implementation of the DSAL on top of CALI

Listing 9.12 shows the implementation of the decorator aspect. The class `Decorator` must extend the class `AspectJ` because the implementation is very similar to the implementation of AspectJ. A decorator is translated using a translation layer (also using Stratego+SDF) to a class that inherits from the class `Decorator`. The constructor of the resulting class must implement the semantics of the decorator.

For example, Listing 9.13 shows the translation of the DSAL aspect of Listing 9.11. The list of bindings of the aspect contains two selector/advice bindings. The aspect has two additional references, one to a decoratee and another for decorator instances. When starting the example, we deploy an instance of the class `MyDecorator` by calling `new MyDecorator().deploy()`.

When capturing a join point, the `Platform` call the method `staticTest` of the instance of `MyDecorator`. If the join point is a call to a method of a `Document` instance, the aspect returns two instances of `Phi` wrapping two selector/advice bindings. If the `decoratee` field is not yet bound (`decoratee==null`), the method `dynamicTest` of the first selector/advice binding returns true and the corresponding advice will be executed. The execution of the advice will bind the `decoratee` field, create a decorator (`new AutoSaveDocument()`) and bind the corresponding `decoratee` to it (see Line 21). If the method `dynamicTest` of the first `Phi` returns false then the method `dynamicTest` of the second `Phi` returns true (the assignment was performed before) and the corresponding advice will be executed. The advice uses reflection to execute the called method (`thisJoinPoint.getSignature().getName()`) by getting a `Method` instance (Line 40) and invoking it with the decorator and the corresponding parameter (the same parameters as the join point).

```
public class MyDecorator extends Decorator {
  Document decoratee;
  AutoSaveDocument decorator;
  static ArrayList<Document> list = new ArrayList<Document>();
  public MyDecorator() {
    Class[] l = { String.class };
    Call call = new Call(Document.class, "*", void.class,
                                        new Class[] { Wildcards.
                                           class });
    ArrayList<SelectorAdviceBinding> choice = new ArrayList<
       SelectorAdviceBinding>();
     // if the decoratee has not been yet decorated
      If anIf = new If() {
        public boolean condition(JoinPoint jp) {
          return !list.contains(jp.getTarget());
        }
      };
      ComposedSelector event =
              new ComposedSelector(new And(), call, anIf);
      Advice advice = new Advice() {
        public Object adviceexecution() {
          decoratee = (Document) thisJoinPoint.getTarget();
          decorator = new AutoSaveDocument(decoratee);
          list.add(decoratee);
          new MyDecorator().deploy();
          return proceed();
        }
      };
      addSelectorAdviceBinding(event, advice);

     // if the decoratee has been decorated
      If anIf = new If() {
        public boolean condition(JoinPoint jp) {
          return jp.getTarget() == decoratee;
        }
      };
      ComposedSelector event =
              new ComposedSelector(new And(), login, anIf);
      Advice advice = new Advice() {
        public Object adviceexecution() {
          Method m = Document.class.getMethod(
                                              thisJoinPoint.
                                                 getSignature().
                                                 getName(),
                                              getClasses(
                                                 thisJoinPoint.
                                                 getArgs()));
          m.invoke(decorator, thisJoinPoint.getArgs());
        }
      };
      addSelectorAdviceBinding(event, advice);
  }
}
```

Listing 9.13: The class `MyDecorator`

```
1 cache Document object {
    memoize wordCount,
3   invalidated by assigning content
    or calling addLine(java.lang.String);
5 }
```

Listing 9.14: An example in Memoization DSAL.

9.3 Memoization DSAL

Memoization is a technique used to improve the speed of program execution. This mechanism consists of returning from a cache the values for previously-processed inputs that have already been calculated, rather than recomputing them each time.

An experimental language that introduces a modular way to specify caching of method return values has been described in [53]. Its syntax follows:

Aspect	::=	`cache` *ClassIdentifier* `object` { *Exp* }
Exp	::=	`memoize:` *MethodIdentifier* (`invalidated by` *InvalidExps*)?
InvalidExps	::=	*SimpleInvalidExp* | *ComplexInvalidExp*
ComplexInvalidExp	::=	*SimpleInvalidExp* `or` *ComplexInvalidExp*
SimpleInvalidExp	::=	`assigning` *FieldIdentifier* | `calling` *Signature*

The class of the method to memoize is specified after the keyword `cache` and its name after the keyword `memoize`. The aspect contains a declaration *Exp* which specifies the method to be cached after the keyword `memoize`. The declaration contains also invalidation expressions which invalidate the cached values and impose their recomputation. The invalidation can be done after assigning a field or calling a method.

9.3.1 Example

Let us reconsider the word processor example of Section 9.2. The method `wordCount` is a good candidate for memoization, as repeatedly calculating the number of words - even when the document has not changed - can become quite time consuming on large documents. The example of Listing 9.14 means the following: apply a caching aspect on each `Document` object (Line 1). This caching aspect will memoize the return value of method `wordCount` (Line 2). The cache will be invalidated when a new value is assigned to the instance variable `content` within the corresponding `Document` object (Line 3), or when the method `addline` is called on a `Document` object (Line 4).

9.3.2 Implementation of Memoization DSAL on Top of CALI

The implementation of Memoization DSAL on top of CALI consists of:

– Creating a `MemoizeAdvice` class.
– For each `memoize` declaration, we create a variable to cache the return value of the memoized method.
– For each memoized method, we create a binding using a `Call` selector and `MemoizeAdvice`. The `Call` selector intercepts calls to that method, and the associated advice returns the cached value (if one is stored) or caches the value returned by `proceed` (if there is no corresponding value in the cache).
– Finally, a pointcut is needed for each cache invalidation specification, coupled with an advice that invalidates the cache. For an `assigning` declaration, we create a `SelectorAdviceBinding` that binds a `Get` selector and a `MemoizeAdvice` instance and for a `calling` declaration, we create a `SelectorAdviceBinding` that binds a `Call` selector and a `MemoizeAdvice` instance.

```
1  public class CachingWordCount {
     Document doc;
3    Object[] args;
     static ArrayList<Document> list = new ArrayList<Document>();
5    static ArrayList<Object[]> listargs = new ArrayList<Object[]>();
     Call wordCount = new Call(Document.class, "wordCount",
7                long.class, new Class[0]);
     Get getContent = new Get(Document.class, "content");

9    Call addLine = new Call(Document.class, "addLine", void.class,
        new Class[]{String.class});
     long memoizedWordCount;
11
     CachingWordCount() {
13     ...
     }
15 }
```

Listing 9.15: A translated memoization aspect

9.3.3 The example revisited

In Listings 9.15, we present the translation of the memoization aspect presented before (Listing 9.14) into CALI (Listings 9.16, 9.17, 9.18 and 9.19 are parts of the construction of the class `CachingWordCount`):

- The variable `memoizedWordCount` (Listing 9.15, Line 10) is used to cache the return value of `wordCount`. As the returned value of a method depends on the target object and the inputs, an instance of the aspect must be created for a different target or different inputs. The variables `doc` and `args` are used to store the target and the input values.
- The call selector `wordCount` (Listing 9.15, Line 6) is used to select the call to the memoized method `wordCount`.
- The advice of Listing 9.17, Line 9 is associated to the call selector to store the return value if it was invalidated (Listing 9.17, Line 11) or to return the memoized value (Listing 9.17, Line 15).
- The selector (Listing 9.18, Line 8) selects the `get` join point of `Document.content` and the associated advice of (Listing 9.18, Line 8) invalidates the memoized value.
- The selector (Line 9) selects a call to the method `addLine` of `Document` and the associated advice of (Listing 9.19, Line 8) invalidates the memoized value.

9.3.4 Translation

The translation from the concrete syntax of the DSL to the CALI representation can be simply achieved with Stratego/SDF.

9.4 COOL

Consider the following piece of code:

```
Advice advice = new Advice() {
  public Object adviceexecution() {
    synchronized (this) {
      return proceed();
    }
```

```
If anIf = new If() {
  public boolean condition(JoinPoint jp) {
    return !list.contains(jp.getTarget())
           || !listargs.contains(jp.getArgs(),args);
  }
};
ComposedSelector event =
        new ComposedSelector(new And(), wordCount, anIf);
MemoizeAdvice advice = new MemoizeAdvice() {
  public Object adviceexecution() {
    doc = (Document) thisJoinPoint.getTarget();
    args = thisJoinPoint.getArgs();
    list.add(doc);
    new CachingWordCount().deploy();
    return proceed();
  }
};
binding.add(new SelectorAdviceBinding(event, advice));
```

Listing 9.16: Selector/advice binding to associate an aspect instance to a target document

```
If anIf = new If() {
  public boolean condition(JoinPoint jp) {
    return (Array.equals(jp.getArgs(),args))
           && (jp.getTarget()==doc);
   }
 };
 ComposedSelector event =
         new ComposedSelector(new And(), wordCount, anIf);
 MemoizeAdvice advice = new MemoizeAdvice() {
   public Object adviceexecution() {
     if(memoizedWordCount == 0l) {
       long l = proceed();
       memoizedWordCount = l;
     }
     return memoizedWordCount;
   }
 };
 binding.add(new SelectorAdviceBinding(event, advice));
```

Listing 9.17: Selector/advice binding to cache the value returned by `proceed`

```
If anIf = new If() {
  public boolean condition(JoinPoint jp) {
    return list.contains(jp.getTarget());
  }
};
ComposedSelector event =
        new ComposedSelector(new And(), getContent, anIf);
MemoizeAdvice advice = new MemoizeAdvice() {
  public Object adviceexecution() {
    memoizedWordCount == 0l;
  }
};
binding.add(new SelectorAdviceBinding(event, advice));
```

Listing 9.18: Selector/advice binding to invalidate cached value when assigning `content`

```
If anIf = new If() {
  public boolean condition(JoinPoint jp) {
    return list.contains(jp.getTarget());
  }
};
ComposedSelector event =
        new ComposedSelector(new And(), addLine, anIf);
MemoizeAdvice advice = new MemoizeAdvice() {
  public Object adviceexecution() {
    memoizedWordCount == null;
  }
};
binding.add(new SelectorAdviceBinding(event, advice));
```

Listing 9.19: Selector/advice binding to invalidate cached value when calling `addLine`

```
public class SelfexAdvice extends Advice<G> {
  public synchronized G adviceexecution() {
    return proceed();
  }
}
```

Listing 9.20: The implementation of `SelfexAdvice`

```
public class MutexAdvice extends Advice<G> {
  List<String> exclusionSet;
  private String name;
  public MutexAdvice(List<String> exclusionSet) {
    this.exclusionSet = exclusionSet;
  }
  public G adviceexecution() {
    if (!thisJoinPoint.getSignature().getName().equals(name)) {
      synchronized (this) {
        name = thisJoinPoint.getSignature().getName();
        return proceed();
      }
    } else
      return proceed();
  }
}
```

Listing 9.21: The implementation of `MutexAdvice`

```
  }
};
```

It represents an advice instance whose role is simply to *proceed* in a synchronized way (in the `adviceexecution()` method). Now let us associate this piece of advice with a selector/advice binding, for example a `Call` selector. The resulting selector/advice binding will prohibit the concurrent execution of the method given in the `Call` selector. This is the base idea of our implementation. Each `selfex` declaration is specified by a selector/advice binding. The selector is implemented as a `Call` selector. We add a special class for selfex advice, called `SelfexAdvice` (Listing 9.20). The `synchronized` declaration of the method `adviceexecution` guarantees that the piece of advice is executed by only one thread at a time. Similarly, each `mutex` is implemented as a selector/advice binding. The piece of advice, implemented as an instance of `MutexAdvice` (see Listing 9.21) manages the synchronization while the selector routes all the methods in the exclusion set to the piece of advice. The method `adviceexecution` of the `MutexAdvice` guarantees that one of the exclusion-set methods is executed at a time. The `mutex` declaration does not prohibit the concurrent execution of the same method by several threads. For this reason, we add a variable `name` to register the method name being executed and we evaluate the condition `thisJoinPoint.getSignature().getName().equals(name)`, before entering in the `synchronized` statement.

For this instance, we implement `mutex` and `selfex` declarations. For statements, expressions and condition, we add a class `CoordinatorAdvice` (see Listing 9.22). The methods `requires`, `on_entry` and `on_exit` are declared as abstract and will be implemented when the class is instantiated. Their definitions is directly extracted from the definitions of the corresponding statements. The method `adviceexecution` combines the three methods.

```
public abstract class CoordinatorAdvice extends Advice<G> {
  public G adviceexecution() {
    while(!requires()) {
      try {
        wait();
      }
      catch(Exception e) {}
    }
    notify();
    on_entry();
    G result = proceed();
    on_exit();
    return result;
  }
  public abstract boolean requires();
  public abstract void on_entry();
  public abstract void on_exit();
}
```

Listing 9.22: The implementation of `CoordinatorAdvice`

```
public class Coordinator implements Aspect {
  private ArrayList<SelectorAdviceBinding> binding;
  public List<Phi> staticTest(JoinPoint jp) {
    ArrayList<Phi> list = new ArrayList<Phi>();
    for(SelectorAdviceBinding b:binding){
      if(b.staticTest(jp)) {list.add(new Phi(b));};
    }
    return list;
  }
  public void addSelectorAdviceBinding(SelectorAdviceBinding sadb){
    binding.add(sadb);
  }
}
```

Listing 9.23: The implementation of `Coordinator` with CALI

```java
public class StackCoord extends Coordinator {
  Stack stack;
  boolean full = false;
  boolean empty = true;
  int len = 0;
  Class[] args = { Object.class };
  Call push = new Call(Stack.class, "push", void.class, args);
  Call pop = new Call(Stack.class, "pop", Object.class, new Class
     [0]);
  public StackCoord() {
     // selfex part
     // mutex part
     // statements
  }
}
```

Listing 9.24: The translated coordinator

```java
    {//  selfex{push}
      SelfexAdvice<Void> advice = new SelfexAdvice<Void>();
      addSelectorAdviceBinding(new SelectorAdviceBinding(push,
         advice));
    }
    {//  selfex(pop)
      SelfexAdvice<Object> advice = new SelfexAdvice<Object>();
      addSelectorAdviceBinding(new SelectorAdviceBinding(pop,
         advice));
    }
```

Listing 9.25: The selfex part of the translated coordinator

Listing 9.23 shows the implementation of the whole coordinator as a class `Coordinator` which contains a list of selector/advice bindings. The method `staticTest` returns a list of `Phi` corresponding to the bindings that match the current join point.

9.4.1 The example revisited

The class `StackCoord` in Listing 9.24 represents the translation of the coordinator of Listing 2.13.

Listing 9.25 shows the representation of the two `selfex` declarations. The implementation consists of a selector/advice binding per declaration. The two bindings are added to the `binding` list of the coordinator. Listing 9.26 shows the representation of `mutex{push,pop}` as a binding between an instance of `ComposedSelector` and an instance of `MutexAdvice`. The selector matches push or pop calls and the advice implements the synchronization policy as described before.

Listing 9.27 shows the implementation of the statement of the coordinator `Stack`. As we described before, the methods `requires`, `on_entry` and `on_exit` are implemented with the instantiation of `CoordinatorAdvice` directly using the definitions of the corresponding statements with the members of the coordinator target (here `buf`) properly prefixed.

```
    { //  mutex{push, pop}
      ComposedSelector event =
              new ComposedSelector(new Or(), pop, push);
      ArrayList<String> list = new ArrayList<String>();
      list.add("push");
      list.add("pop");
      MutexAdvice<Object> advice = new MutexAdvice<Object>(list);
      addSelectorAdviceBinding(new SelectorAdviceBinding(event,
         advice));
    }
```

Listing 9.26: The mutex part of the translated coordinator

```
    {
      CoordinatorAdvice<Object> advice = new CoordinatorAdvice<
         Object>() {
        public void on_exit() {
          empty = false;
          len++;
          if (len == stack.buf.length) {
            full = true;
          }
        }
        public void on_entry() {}
        public boolean requires() { return !full; } };
      addSelectorAdviceBinding(new SelectorAdviceBinding(push,
         advice));
    }
    {
      CoordinatorAdvice<Object> advice = new CoordinatorAdvice<
         Object>() {
        public void on_entry() {
          len--;
        }
        public void on_exit() {
          full = false;
          if (len == 0) {
            empty = true;
          }
        }
        public boolean requires() { return !empty; } };
        addSelectorAdviceBinding(new SelectorAdviceBinding(pop,
           advice));
    }
```

Listing 9.27: The statement part of the translated coordinator

9.5 Conclusion

In this chapter, we have presented the CALI implementation of aspect mechanisms that are different from the AspectJ one. This chapter is a proof of the methodology presented in Chapter 6 because it have been used for implementing heterogeneous aspect mechanisms. The homogeneity of the different implementation makes it simple to compose aspects mechanisms as we will see in the next chapter.

Chapter 10

Composition of AOP languages

Contents

Once the different concerns are expressed with DSALs, we must have the possibility of composing them in a single application. Implementing this composition is not straightforward. Even if this is feasible (for example by sequentially running the different DSAL weavers), running the combined application may lead to unexpected and/or undesired results as it was shown in the state of the art (see Chapter 4). In this chapter, we first describe the ability to compose AOPLs and to make it possible to deal with much larger problems and configure this composition in CALI. Section 10.1 describes technically how aspect composition is feasible. Section 10.2 shows the possible interactions that lead to undesired behavior when composing AOPLs and what CALI provides in order to configure the composition. Section 10.3 presents an example of composing the AspectJ and COOL prototypes in CALI. This example is typical of problems that occur when composing aspects written using different languages.

10.1 From composing multiple aspects to composing multiple AOPLs

In the previous chapters, we have seen the semantics of composing several aspects written in the same language (implemented with CALI). The platform communicates with the aspect instances through their interface `staticTest`. Each aspect instance returns a list of instances of the class `Phi` to the platform, which will complete the evaluation of this list with the current join point.

The key of composing AOPLs in CALI is that all the aspect implementations implements the interface `staticTest` that returns a list of `Phi`, which is uniformly treated by the platform. With this common interface, aspects from different AOPLs can co-exist in the same application and communicate in the same way with the platform.

10.2 Scaling composition

Composing aspects in CALI is natural and direct. The join points occurring within the base program are determined by the platform while each individual prototype decides the additional set of join points that have to be generated within the implementation. For example, advice execution of AspectJ was reified. Unlike transformation approaches, the approach of CALI allows full control about the generated join points and prevents the generation of implementation-level join points.

Now let us consider two prototypes of two AOPLs implemented with CALI. The first language has not reifed its pieces of advice (genereting join point during the execution of each piece of advice) while the second one has. Scaling the composition of these two AOPLs could be described as follows:

– An aspect from the second language must match join points occurring within advice of the first language.
– An aspect from the first language must not handle join points occurring within advice of the second language.

In the rest of this section, we present the different types of aspect interactions, their level of resolution as well as the CALI features allowing composition scaling.

10.2.1 Interaction

Let us recall some important properties of aspect interactions (see Chapter 4).

Categories of Interaction Kojarski and Lorenz [63, 65] characterize two chief categories of aspect interactions whose resolution is key to support multiple AOPLs composition:

Co-advising happens when several aspects from differents aspect extensions need to be superimposed on the same join point. This is usually an advice scheduling problem where the resolution consists of defining what is the order of execution of the various pieces of advice.

Foreign advising happens when an aspect of an aspect extension matches a join point within the execution flow of a piece of advice of another extension. Resolving this type of interaction consists of defining what is the part of a piece of advice that can be matched by foreign aspects.

Level of resolutions There are two levels [65] at which interactions are typically resolved:

Language level requires the *language designer* to specify the semantics for interactions between aspects in each Aspect-Oriented language and aspects in all the other languages.

Program level requires the *aspect programmer* to resolve how a concrete set of aspects interact.

10.2.2 Interactions resolutions in CALI

Co-advising CALI represents an approach for resolving *co-advising* at both the language level and the program level. Let us return to the `order` subprocess in the aspect `Platform`. This process is configurable to order aspects according to the aspect language (language level) or by reading a specification written by the programmer (program level). The syntax of the specification language (DSL), in which the specification of ordering aspects, is future.

Foreign advising In CALI, the platform is only responsible for reifying the join points of the base program. It is up to each extension to reify the join points that occur in its own aspects. In the case of AspectJ (Section 7.1), we have added the following pointcut

```
pointcut reifyAspectJ():
  (
    !call(AspectJ.staticTest) &&
    !execution(AspectJ.staticTest) &&
    !set(AspectJ.list) &&
    !get(AspectJ.list) &&
  ) &&
  within(AspectJ) &&
  cflow(execution(* AspectJAdvice.adviceexecution(..)));
```

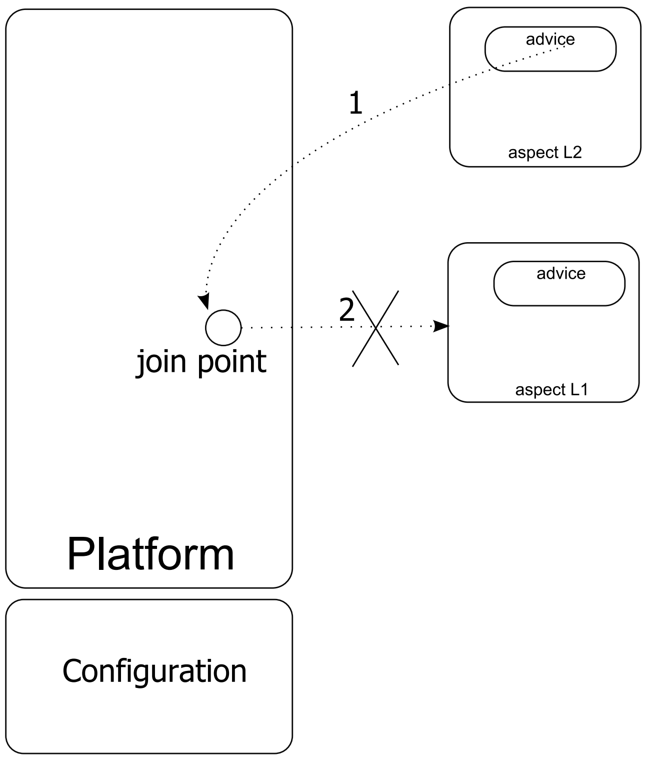

Figure 10.1: Composing two aspects from different languages

to the pointcut `reifyBase` of the platform in order to implement the semantics of AspectJ where it is possible to match join points within pieces of advice as well as the specific join point `adviceexecution`. CALI resolves *foreign advising* at the language level by controlling the scope of aspect of L_1 over join point generated from advice of L_2 and by controlling the generation of join points resulting from the execution of synthetic code of L_2. Let us consider the composition between two AOPLs, L_1 and L_2. We suppose that the *specification* of the composition consists of preventing L_1 aspects from matching join points generated within L_2 aspects. Figure 10.1 shows that the $aspect_1$ is prevented from matching a join point generated by the execution of $aspect_2$ piece of advice using an aspect `Configuration`. When composing AOPLs implemented with CALI, we need to define an aspect `Configuration` to specify if a foreign join point has to be matched by an aspect. An example of such a configuration is described in 10.3.

10.3 AspectJ and COOL

10.3.1 Problem

As we mentioned in Chapter 3, when composing COOL and AspectJ, resolving foreign advising consists of controlling the weaving of AspectJ advice into COOL coordinators, and the weaving of COOL advice into AspectJ aspects. According to what we have said in the head of Section 10.2, AspectJ aspects are reified by `reifyAspectJ` and COOL aspects are reified by `reifyCOOL`.

10.3.2 Specification of the composition

In the composition of AspectJ and COOL, an aspect is woven into classes and aspects (AspectJ aspects) according to the weaving semantics of AspectJ. Similarly, a coordinator is woven into classes according to the weaving semantics of COOL. The specification must include the behavior of aspects and coordinator when dealing with with foreign advising and co-advising. We rely on the specification defined in [64]:

Foreign advising The specification permits coordinators to advise methods that are declared within aspects in the same way as methods within classes. The specification restricts coordinators and aspects from advising any synthetic code introduced by the foreign aspects: coordinators do not advise advice methods in aspect classes, and aspects do not advise methods of the coordinator classes by the COOL implementation.

Co-advising The specification imposes that:

- The lock (unlock) advice of COOL is executed before (after) the before, around, and after advice of AspectJ.
- From the perspective of AspectJ aspects, COOL advice executes in the control flow of the method execution join point it advises.

The specification of [64] is defined in the context of static weaving in order to precise the behavior of AspectJ and COOL weavers.

The specification is restated as follows in CALI: Coordinators advise methods that are declared within aspects except the method `staticTest`. COOL aspects are applied before AspectJ ones.

10.3.3 Composition configuration

When implementing AspectJ on top of CALI, we have introduced two methods: `staticTest` and `addSelectorAdviceBinding` (see Listings 7.1). These methods are not reifed by the implementation of AspectJ to prevent the fault interception of the corresponding join points. This guarantees that coordinators will never advise these implementation-level methods while coordinators can match other methods. The same reasoning is valid for coordinators: by construction, there are no join points generated within coordinator in the prototype of COOL, and therefore, AspectJ aspects will not apply.

10.4 Conclusion

In this chapter, we have shown that our framework supports applications composed of aspects written in several AOPLs. In addition, we have discussed how CALI interactions that may occur when combining multiple AOPLs, and demonstrated that the aspect mechanisms implemented using CALI decrease synthetic code, and therefore aspect interactions.

Part III

Perspectives

Table of Contents

Chapter 11

Performance

Contents

In the previous chapters, we have demonstrates the flexibility and expressiveness of CALI as a framework for prototyping AOPLs. In order to evaluate the overhead of using CALI mechanisms, we use a profiling tool and compare the execution of a simple application with plain AspectJ and with the AspectJ prototype using CALI described in Chapter 7 to analyze application execution and identify performance problems, such as execution bottlenecks, object leaks, and system resource limitations. Section 11.1 introduces the profiling platform *TPTP Eclipse plugin*. Section 11.2 presents a running example consisting of a class `Fibonacci` that computes the Fibonacci series. This class is profiled after adding an AspectJ caching aspect. The aspect is implemented with AspectJ and CALI AspectJ prototype. The results of profiling are exposed in Section 11.3. They are discussed in Section 11.4.

11.1 TPTP Eclipse plugin

The Eclipse Test and Performance Tools Platform (TPTP) Project [4] offers a profiling tool for identifying and isolating performance problems such as performance bottlenecks, object leaks and system resource limits. The tool targets applications of all levels of complexity, from simple standalone Java applications to Eclipse plugins or complex enterprise applications running on multiple machines and on different platforms. The Profiling and Logging perspective provides resources for starting a profiling session as well as obtaining comprehensive information on the performance of the monitored application. This is done by collecting four types of information about the execution of each method of the application:

Base Time The amount of time of the method execution not including the execution time of any other methods called from this method.

Average base time The average base time of the method execution.

Cumulative base time Unlike base time, this time includes the execution time of any other methods called from this method.

Calls The number of times this method was invoked.

Regarding the whole application, TPTP calculates the sum of the above values for all the methods.

```java
public class Fibonacci {
  public long fibonacci(int n) {
    if (n == 0 || n == 1)
      return (long) n;
    else
      return fibonacci(n - 1) + fibonacci(n - 2);
  }
  public static void main(String[] args) {
    if (args.length == 0)
      System.out.println("Specify which fibonacci number should be
         calculated.");
    else {
      int n = Integer.parseInt(args[0]);
      long theFibonacciNumber = new Fibonacci().fibonacci(n);
    }
}
```

Listing 11.1: The `Fibonacci` class

11.2 Running application

The parameter of the method `fibonacci` of the class Fibonacci 11.1 specifies how many numbers in the series it should compute. The program is implemented using recursion. The execution time of the method is exponential. The machine used in the experiments of this chapter has one 1.86 GHz intel processor and 1 GB of RAM. The AspectJ version is 5 running with JRE 1.6.0_01.

In order to decrease the computation time, we add a memoization aspect, `FibonacciSaverAspect`, based on the concept of memoization discussed in the previous chapter to cache values that will be needed in the future. Its implementation in AspectJ is given in Listing 11.2 and in CALI in Listing 11.3. In our example, in order to compute `fibonacci(n)` for some `n`, we also need the values of `fibonacci(n-1)` and `fibonacci(n-2)`. And since we need to compute these values recursively, we end up having to use all values of `fibonacci(m)`, where $1 \leq m \leq n$. The regular solution to the problem (which is presented in the class `Fibonacci`) is to compute each of these values over and over again, resulting in exponential execution time. With memoization, we will only compute the value of `fibonacci(m)` for any `m` exactly once. As soon as the value is computed the first time, it is stored for future reference. From the next time on, the value is simply looked up from this table, and not computed. As a result, the execution time becomes linear. In order to do this, the `FibonacciSaverAspect` include a private data member called `fibValues`, an array of `long`, that stores the computed fibonacci values. The aspect also includes an around advice to intercept any call to `fibonacci(n)`. The advice checks the `fibValues` array to check if the requested fibonacci value has already been computed, and if so, the value is directly returned from the cache instead of recursively invoking `fibonacci`.

11.3 Results

In this section, we present the profiling results when writing the aspect using AspectJ and AspectJ-CALI. We proceed in two stages: the first without the aspect and the second with the aspect. The results are given in Table 11.1, lines"AspectJ without aspect" and "AspectJ with aspect", respectively.

When profiling the application with CALI, some performance overhead is expected due to the reifying mechanism even in the absence of aspects. For this reason, it is important to profile the application in the absence and the presence of the aspect. Listing 11.3 shows the implementation of the caching aspect using our AspectJ prototype. The results are also given in Table 11.1, lines

```
public aspect FibonacciSaverAspect {
  long[] fibValues = new long[100];
  pointcut p(int n):
          call(long Fibonacci.fibonacci(int)) && args(n);
  long around(int n):p(n){
    if (fibValues[n] == 0) {
      long l = proceed(n);
      fibValues[n] = l;
      return l;
    } else
      return fibValues[n];
  }
}
```

Listing 11.2: The `FibonacciSaverAspect` AspectJ aspect

framework	Base Time	Average Base Time	Cumulative Time	Calls
AspectJ without aspect	11.316547	0.000197	11.316547	57315
AspectJ with aspect	0.309567	0.001420	0.309567	218
CALI without aspect	86.466075	0.000216	86.466075	401227
CALI with aspect	0.948928	0.00487	0.898928	306

Table 11.1: Summary of profiling results

“CALI without aspect” and “CALI with aspect”, respectively.

11.4 Discussion

Table 11.1 presents a summary of all the results. In order to interpret these results and evaluate the overhead on performance, we must start by reconsidering the interception mechanism of CALI. The AspectJ weaver transforms all join point shadows defined by the pointcut `reifyBase` and inserts additional code that calls the advice of the aspect `Platform`. The pointcut `reifyBase` is based on `call`, `execution`, `set`, etc. The matching of all these designators are based on static information in the current version of AspectJ as we have shown in Chapter 7 and Chapter 8, and therefore there is no dynamic test needed for join point matching.

11.4.1 Without Aspects

In AspectJ, the weaver does not change the base program because there is no aspects and therefore, no overhead on performance related to the additional code. However, without memoization, all the calls to the method `fibonacci` are executed. In AspectJ-CALI, after intercepting a join point, the platform verifies that there is no aspects and executes the method `proceed`. This mechanism is repeated for all the join points defined by `reifyBase`, which adds a considerable overhead (8 times) in addition to the absence of memoization.

11.4.2 With Aspects

In AspectJ, the weaver inserts additional code to call the advice performing memoization but this code is a forward call only and does not cause overhead on performance. In addition, the memoization of the method `fibonacci` decreases the number of executions and therefore, the overall performance is improved. In AspectJ-CALI, the performance is better with aspect

```
public class FibonacciSaverAspect extends AspectJ {
  long[] fibValues = new long[100];
  public FibonacciSaverAspect() {
    Class[] l = { int.class };
    addSelectorAdviceBinding(new SelectorAdviceBinding(
      new Call( Fibonacci.class, "fibonacci", long.class, l),
      new Advice<Long>() {
        public Long adviceexecution() {
          int n = (Integer) thisJoinPoint.getArgs()[0];
            if (fibValues[n] == 0) {
              long l = proceed();
              fibValues[n] = l;
              return l;
            } else
              return fibValues[n];
         }}));
  }
}
```

Listing 11.3: The `FibonacciSaverAspect` aspect with CALI-AspectJ plugin

than without aspect. The number of execution of `fibonacci` is decreased but the overhead of interpretation still impacts the overall performance. This explains why the performance of AspectJ is better than the performance of AspectJ-CALI. Neither the execution of the advice nor the execution of the matched join point significantly impact performance but some major overhead is due, first, to the reification of join points which are not associated to any aspect and, second, to the evaluation of a join point by join point selectors. In fact, the piece of advice is not interpreted in CALI but it is executed as normal Java code because pieces of advice are defined as inner classes. The execution of the current join point is optimized using a closure as we have described in Chapter 6. An evolution of CALI would consist of statically generating the `reifyBase` according to the existing aspects to avoid the reification of unnecessary join points. This means that in our example, the pointcut `reifyBase` would match only the calls to the method `fibonacci`, not its executions.

11.4.3 Conclusion

The profiling results show that even though we have an overhead on performance with CALI mechanisms, this still acceptable and allows the use of CALI with large applications, where different concerns can be modularized as aspects and implemented using the different AOPL prototypes based on CALI.

Chapter 12

Related Work

Contents

12.1 JAMI

JAMI (Java Aspect MetaModel Interpreter) [53, 54, 52] is a framework to prototype and compose domain-specific aspect languages.

12.1.1 Features and Benefits

JAMI shares with CALI these points:

– It finds its origin in Metaspin. Both JAMI and CALI implement the basic concepts of join point, pointcut/join point selector, advice and binding.
– It relies on two-step weaving with AspectJ to generate join points and interpret the corresponding pieces of advice.
– The aspects are completely dynamically evaluated. This makes it easy to experiment with pointcuts that express complex selection criteria over the runtime state. In addition, their support for aspect state using variables that each may have different instantiation policies provides a flexible way to implement aspect language features, while requiring relatively little effort.

The implementation of memoization in JAMI and CALI is not essentially different: a set of selector/advice bindings is used to retrieve and cache values and another set is used to invalidate cached values.

The implemention of Decorator using CALI benefits from the implementation of AspectJ by reusing different elements while the implementation using JAMI needs the creation of new specific building blocks like `AssociateDecoratorAdvice`, `BindObjectToVariableAction`, `SkipOriginalCallAction`, etc.

The implementation of join-point selectors in JAMI consists of subclassing the class `JoinPointSelector` and implementing the abstract method `evaluate(JoinPoint jp, InterpreterContext context)`. This shows that there is no separation between static and dynamic information of join point matching while CALI clearly separates it between two methods `staticTest` and `dynamicTest` and makes it possible to partially evaluate selectors at compile time.

JAMI also provides a set of dedicated selectors like `SelectByAssociatedVariable` (to compare `target` or `this` objects of the join point with a variable of the aspect specified with its name), `SelectByFieldContainingType` (to verify the class of the field in a field-access join point), that can be composed into complex selectors useful for implementing DSALs. But CALI could use the generic selector `If` composed with other selectors to get the same effect.

12.1.2 Limitations

The lack of a formal semantics for JAMI makes it difficult to understand the interpretation flow and the supported mechanisms and notions. Also it relies on a simplified join-point model without around advice and `proceed`, and has not been designed with the objective of supporting a general-purpose language such as AspectJ.

12.2 AWESOME

AWESOME [60, 64] is a composition framework for the construction and the composition of aspect extensions *weavers* as plugins on top of the framework. The starting point is the conceptual model of a weaver that enables the construction of an aspect weaver on top of a basic *platform*. Once a weaver is built with AWESOME, it can be composed, as third-party composition, with other weavers that were also built on top of AWESOME.

12.2.1 Features and Benefits

The conceptual model used in CALI (the four sub-processes) is based on the model proposed by AWESOME. Also resolving aspect interactions in CALI is very close to the AWESOME's one but applied in a different context. The two frameworks use configuration aspects to resolve foreign-advising but in CALI, this is done at runtime rather than at weave time as in AWESOME. AWESOME has a component and aspect oriented architecture. The framework is based on three notions:

1. **Platform**: The class `Platform` implements the 4 basic subprocesses: **reify**, **match**, **order** and **mix**. The word *basic* means that they are independent from any aspect language. `Platform` has a list of *mechanisms*, each one corresponds to an aspect language. This feature allows AWESOME to compose different aspect extensions by having different mechanisms in this list.
2. **Mechanism**: It is an abstract notion to express what a new aspect extension built with AWESOME adds to each subprocesses implemented in the class `Platform`. When impementing a class `Mechanism`, a plugin designer must implement the corresponding **match** and **order** subprocesses, which will affect the global match and order subprocesses of the platform. This feature is implemented as an aspect that intercepts join points in the platform to add behavior to the default **match** and **order** subprocesses.
3. **Config**: The class `Platform` provides a default composition behavior but this can be customized using a configuration aspect `Config` which is implemented as an aspect that intercepts join points in the platform and each mechanism (see item 2) to control the interactions between languages (foreign and co-advising).

The AWESOME framework provides plugin composability. The plugins that were independently developed using AWESOME can be composed as third-party components by implementing a subclass of `Mechanism` and adding an instance of it to the list of mechanisms in the platform.

12.2.2 Limitations

– AWESOME is more dedicated to managing interaction of AOPLs composition than to easy prototyping. In fact, it does not propose features like our Abstract AOPL in order to facilitate the prototyping of AOPLs.

- AWESOME supports the configurability of resolving aspect interactions at the language level, but does not allow the programmer to resolve these interactions among a set of aspects (written in different languages). For example, the composition of AspectJ and COOL, called COOLAJ, imposes that the AspectJ aspects or the COOL aspects have to be executed first at a shared join point. There is no possibility for the programmer to define a scheduling of aspects execution by choosing which aspects have to be executed first (based on the aspect name and not the aspect language).
- AWESOME resolves feature interactions at compile time and this makes it unable to resolve the foreign advising interactions that depend on run-time information.

12.3 The Art of the Meta-Aspect Protocol

Dinkelaker *et al.* [35] propose an architecture for AOPL with an explicit interface to language semantics inspired by meta-object protocols. Using, the meta-aspect protocol, AOP developers can then tailor language semantics in an application-specific manner. Among the applications of the meta-aspect protocol, let us mention the resolving of the aspect interactions that depend on the dynamic program context and dynamic deployment.

The meta-aspect manager is responsible for loading aspects. Specializing the meta-aspect manager (especially overriding `interactionAtJoinPoint`) and replacing it at runtime allows dynamic advice ordering.

The architecture of the Meta-Aspect Protocol is very similar to our extension of MetaJ and provides the same abilities such as dynamic reordering and dynamic deployment. The dynamic access to the proceed stack in Dynamic AspectJ is similar to accessing the meta-aspect manager and replacing it at runtime.

12.4 Composing aspects with aspects

Marot *et al.* [75] share with us the idea of composing aspects using aspects. They propose an extension of AspectJ, called Oarta, to empower aspects to express aspect composition similarly to what is done in Dynamic AspectJ where an advice can access the list of aspects matching the same join point. Oarta extends existing join point models that focus on the relations of aspects with the base program in order to obtain a join-point model that focuses on the relations between aspects. This guarantees the separation of composition-specfic code from the composed aspects.

Chapter 13

Conclusion

Contents

Aspect-Oriented Programming has celebrated its first decade of research, development and industry adoption. There are many AOPL tools, languages, and frameworks.

To promote more mature use of AOP, new features are continually being proposed. These features should be an extension of existing features or new ones. The realization of these features imposes the extensibility of the existing implementations and the existing of tools providing utilities for facilitating these implementations.

Extensible compilers, like `abc`, allow only fine-grained extension to the already implemented AOPLs (for instance, AspectJ for `abc`). Moreover, some simple extensions, like adding a new type of pointcut or changing the semantics of existing pointcuts (alternative semantics for `call` and `execution`) do not require all the machinery of a full fledged compiler like `abc`. When initially experimenting with such extensions, a lightweight extensible implementation of AspectJ suffices, allowing rapid prototyping. At this stage, performance considerations are not necessarily an issue. When the extension matures and optimizations are considered, turning to a full compiler infrastructure makes sense.

Alternative approaches have been proposed to realize new features like AOP kernels (Reflex, XAspects) that introduce primitives for expressing various weaving operations. Their principle consists of translating AOPL programs to a shared intermediate representation. Unfortunately, this way of implementing AOP features still requires a big effort to bridge the gap between the semantics of the aspect language and the semantics of the intermediate representation. Moreover, it is often impossible to safely use a kernel-based implementation with another one because each one of these implementations introduce implementation-specific code in the translated representation which can be considered as normal code for the other implementations.

Providing support for prototyping and composing independent aspect languages is a worthwhile challenge, it gives the ability to accelerate the evolution of aspect mechanisms by bridging the gap between the design and the test of new aspect mechanisms because the easy experimentation of new features makes the validation or the correction of the novel approaches possible.

This dissertation contributes to both axes: the prototyping and the composition of AOP languages. It gives means to:

– Guide the design and the implementation of individual aspect languages as extensions of the abstract aspect language interpreter.
– Implement the composition of different aspect languages as a natural result of implementing these languages using the same framework.

– Support the resolution and the configuration of aspect interactions when composing aspect languages (at program level and language level).

13.1 Prototyping and Open implementations

We contribute to the prototyping axis with a practical framework, based on interpreters to easily prototype AOP languages. The approach consists of providing an interpreter that implements the common aspect semantics base of an aspect language and keeps the specific mechanisms abstract. The prototyping of a concrete aspect language consists of specifying the abstract notions in the interpreter.

Tho key points of our approach are:

1. An existing definition of the semantics of aspect languages, the Common Aspect Semantics Base (CASB).
2. A representation of the common notions in aspect languages, the metamodel of aspect languages, MetaSpin.

Chapter 5 has shown how to build an interpreter for the abstract aspect language, represented by the metamodel, based on the semantics of the CASB, by considering the evolution of an existing base interpreter for a subset of Java. The properties of the aspect interpreter are the following:

– It is cleanly separated from the base one.
– It directly implements the CASB semantics.
– Its architecture facilitates its extension to implement concrete aspect languages.

We have validated the above approach by extending the interpreter and implementing a light version of AspectJ. We have shown how the specification and the extension of the abstract parts in the interpreter is easy due to the architecture of the aspect interpreter.

To apply the same approach and the same architecture to the whole Java without changing its interpreter (JVM), we reuse AspectJ to generate join points and forward them to a thin interpretation layer. Chapter 6 has described the resulting framework which is called CALI.

Chapter 7 has validated the approach by describing an implementation of AspectJ while Chapter 8 has shown how this implementation is easily extensible with two variants of AspectJ, one [14] dealing with the dynamic scheduling of aspects like dynamic reordering and dynamic canceling and another [29] dealing with alternative semantics for `call` and `execution`. We have also validated our approach by describing prototypes for EAOP, COOL and a couple of other DSALs in Chapter 9. The prototype of EAOP was used, in the context of the European project AMPLE [1], to prototype ECaesarJ [83], an extension of CaesarJ [11].

From the point of view of open implementations, we can say the language implementation with CALI provides a high level of extensibility and reutilisability.

The AspectJ implementation is very flexible. The implementation of pointcut in an interpreted way, as a direct implementation of their semantics, gives us the possibility to directly reflect the changes to the semantics in the implementations. We have applied this to the `call` and `execution` pointcuts, and have shown that variant semantics could be easily implemented.

The version of dynamic reordering has also been implemented with few changes in the basic implementation by inserting methods in the advice implementation to give access to the proceed stack.

Finally, we have reused the pointcuts implementations of AspectJ in EAOP and other DSALs without reinventing the wheel every time.

13.2 Composition

The second axis of this dissertation is the ability to compose aspect languages. In fact, CALI supports, in its core, the composition of several aspects. Each implemented language extends the notion of *aspect* existing in the abstract aspect language. The interaction between the core and each aspect instance is the same for all the implemented languages. The result is that the common interpreter can compose the aspects of different languages as abstract aspects while they extend the same abstract notion (`Aspect`) of the abstract language and have the same interface with the core.

We can say that the implemented languages can naturally coexist but the *designer* must control the *conflicts* between the residents. The two expected conflicts are the *co-advising* and the *foreign advising* conflicts.

We give the possibility to resolve the *co-advising* conflicts at two levels:

Language level By configuring the `order` method in the `Platform` aspect to define the aspect execution order depending on their languages.

Program level CALI is the only framework that supports the resolution of *co-advising* at the program level by configuring the `order` method. This is done by reading a specification defining the aspect execution depending on the aspects and not their languages. An additional type of configuration is the dynamic scheduling of aspects, implemented not only in the advice of Dynamic AspectJ but as a feature in the advice notion of CALI.

We give the possibility to resolve the *foreign advising* conflicts at the *language level* by using the `Config` aspect. This aspect controls the scope of join points of the considered language to prevent the matching of these join points by an advice of another language (foreign advice).

Chapter 14

Future Work

Contents

Directions for future work include improving the existing prototypes and apply them to different domains like context-aware application and debugging aspect programs.

14.1 Dynamic AspectJ Plugin

An important enhancement would be the implementation of Dynamic AspectJ as an Eclipse plugin for Dynamic AspectJ and the improvement of the reifying mechanism to generate join points only where needed. This would increase the performance of Dynamic AspectJ while conserving all its features except the possibility of defining new aspects. This consists of parsing all the aspect declarations, capturing their pointcuts and generating a `Platform` aspect containing a `reify` pointcut that is the union of all the pointcuts.

14.2 Context-Aware Application

Context awareness originated as a term which sought to deal with linking changes in the environment with systems, which are otherwise static. The idea is that programs can change their behavior, and react based on their environment. Software systems must adapt to changing contexts over time, even while they are running. Unfortunately mainstream programming languages and development environments do not support this kind of dynamic change very well, leading developers to implement complex designs to anticipate various dimensions of variability.

The notion of Context-oriented Programming (COP) [31, 56] directly supports variability depending on a large range of dynamic attributes. In effect, it should be possible to dispatch runtime behavior on any properties of the execution context. According to [31], the main notions in COP are:

Behavioral variations Variations typically consist of new or modified behavior, but may also comprise removed behavior. They can be expressed as partial definitions of modules in the underlying programming model such as procedures or classes, with complete definitions representing just a special case.

Layers Layers group related context-dependent behavioral variations.

Activation Layers aggregating context-dependent behavioral variations can be activated and deactivated dynamically at runtime. Code can decide to enable or disable layers of aggregate behavioral variations based on the current context.

Context Any information which is computationally accessible may form part of the context.

Scoping The scope within which layers are activated or deactivated can be controlled explicitly. The same variations may be simultaneously active or not within different scopes of the same running application.

The link between AOP (specially Dynamic Aspect-Oriented Programming) and COP is deep as both paradigms help with separation of concerns and system adaptation.

Our proposition is to use our prototype of Dynamic AspectJ for Context-Oriented Programming. We can introduce the notion of *layer* to Dynamic AspectJ as a set of aspects that are related context-dependent behavioral variations. A layer can be activated by the activation of its aspects. An additional feature provided by Dynamic AspectJ is that the *order* of layer execution can be modified depending on runtime information (dynamic scheduling of aspects) whereas this feature does not exist in the languages supporting COP like ContextL [31].

Another perspective for future work is also the notion of *context-aware aspect* [91] which is an aspect whose behavior depends on the context. CALI could be used to define new types of pointcuts to control the scope of aspects depending on the context.

14.3 Debugging Aspect-Oriented Programs

Debugging means the ability to diagnose faults in a software system, and to improve comprehension of a system, by monitoring the execution of the system.

The ability to debug programs composed using AOP techniques is critical to the adoption of AOP. Nevertheless, many AOP systems lack adequate support for debugging, making it difficult to diagnose faults and understand the composition and control flow of the program. For example, when using the AspectJ compiler during AspectJ software evolution, when regression tests fail, it may be tedious for programmers to find out the failures.

Being an interpreted framework, CALI could provide support for *debugging*. Aspects in CALI are not translated and we have control to correctly debug aspect programs to resolve the interaction of aspects with the system. A new feature of CALI can serve when debugging aspect programs: it is the access to the proceed stack because some failures come not only from the interactions between aspects and base but also from the interactions between the aspects themselves. These feature are also applicable when debugging several aspect languages implemented with CALI because the different implementation source code abstractions are visible (able to represent the executing software system in terms of the pro- gramming language abstractions) and traceable (abile to trace a specific executing behavior back to its source code segment) [10].

Appendices

Résumé

Cette thèse introduit une approche basée sur l'usage des interpréteurs pour le prototypage rapide et la composition des langages d'aspects. Nous avons construit notre infrastructure en créant un cadre dédié à la définition de la sémantique opérationnelle concrète des langages d'aspects. L'interprétation joue un rôle intermédiaire dans la définition des langages d'aspects entre un niveau plus abstrait qui est la sémantique et un niveau plus bas qui est le tissage. Des extensions et des modifications sur la sémantique peuvent être facilement supportées dans un interpréteur ce qui permet d'explorer l'espace de conception des langages d'aspects et d'expérimenter avec diverses variantes de sémantiques des langages existants. Partant d'un prototype d'AspectJ implémenté selon notre approche, nous montrons comment des sémantiques alternatives d'AspectJ concernant les coupes et la planification dynamique des aspects peuvent être facilement implémentées avec peu de changements dans le prototype. La thèse montre aussi comment un langage d'aspects dédié peut être facilement implémenté.

1 Introduction

Les langages de programmation sont au cœur du génie logiciel. L'histoire de la recherche dans ce domaine peut être considérée comme une quête perpétuelle pour la modularisation idéale des préoccupations dans un système. Les paradigmes de programmation existants comme la programmation par objets et la programmation par composants sont utiles pour modulariser les *préoccupations de base* du système (calculs et données); par contre ces paradigmes ne fournissent pas un support pour la modularisation d'un type de préoccupations, appelé *préoccupations transversales*, représentant des aspects du système qui affectent (coupent transversalement) d'autres préoccupations comme le logging, la synchronisation et la sécurité.
Souvent, ces préoccupations ne peuvent pas être proprement découplées à partir du reste du système, et le code de chaque préoccupation est dispersé à travers le programme et emmêlé avec d'autres préoccupations. Certaines techniques de programmation comme les protocoles à méta-objets [58] et la réflexion [30] ont tenté de résoudre ce problème. Ces techniques sont complexes et ne fournissent pas de support linguistique pour la modularisation de préoccupations transversales.

La programmation par aspects (PPA) [47] a été proposée pour répondre aux problèmes des préoccupations transversales en fournissant un support linguistique pour les modulariser. Ceci est réalisé en laissant l'utilisateur définir le comportement d'une préoccupation transversale, puis décrire d'une manière déclarative où cette préoccupation doit recouper les autres modules. Il est de la responsabilité de l'infrastructure du langage d'élaborer le programme final. La prochaine section résume les principes de la PPA et motive l'utilisation de plusieurs langages d'aspects pour exprimer différents types de préoccupations transversales qui peuvent affecter simultanément le système.

1.1 Programmation par Aspects

La PPA [47] tente d'aider les programmeurs dans la séparation des préoccupations transversales en proposant des extensions des langages existants; ces préoccupations transversales sont elles même modularisées comme des *aspects*.

La séparation des préoccupations transversales avec la PPA améliore la qualité des modules en diminuant leur couplage avec d'autres modules et donc en améliorant leur maintenabilité et leur réutilisabilité. Actuellement, la PPA est principalement réalisée en tant qu'extension de la programmation par objets (en particulier, plusieurs extensions de Java ont été proposées dont le langage standard de PPA AspectJ [57]), et implémentée à l'aide de technologies de *transformation* de programme afin de composer statiquement (*weaving*) des aspects.

En PPA, le point où deux préoccupations se coupent est appelé *point de jonction* (*join point*) [57]. Pour déterminer l'ensemble des points de jonction intervenant dans une composition, la majorité des langages fournissent un sous-langage, appelé le langage de *coupe* (*pointcut*). Les expressions du langage de coupe déterminent l'ensemble des points de jonction qui doit être capturé par l'aspect. Une coupe est liée à une déclaration qui sera exécutée lors de l'occurrence d'un point de jonction. Cette liaison est appelée *greffon* (*advice*). Ce type de langage d'aspects est dit conforme au modèle *coupe-action* qui sera décrit dans la suite.

Un langage d'aspect peut être générique ou dédié à un domaine. L'utilisation d'un langage d'aspects dédié à un domaine présente de nombreux avantages : représentation déclarative, analyse et raisonnement simplifié, vérification des erreurs au niveau du domaine et optimisation [32]. Cette approche suit celle de la programmation orientée langage [102]. C'est un style de programmation passant par la méta-programmation qui, plutôt de résoudre un problème dans des langages de programmation génériques, consiste à créer un ou plusieurs langages dédiés au domaine du problème, et de le résoudre en utilisant ces langages. Plusieurs langages d'aspects dédiés ont été en effet proposés au début de l'apparition de la PPA (COOL et RIDL à Xerox Parc [68]), puis après une période de concentration sur les langages d'aspects génériques (AspectJ à Xerox Parc aussi), l'intérêt des langages dédiés a été ravivé [28, 100, 92]. Un langage d'aspects dédié fournit un moyen de simplifier le développement d'une préoccupation, comme la concurrence, la distribution, la sérialisation, etc. Le développement des grandes applications distribuées implique l'intégration de préoccupations multiples, avec de multiples parties prenantes manipulant le système par le biais de points de vues différents. Quand plusieurs aspects sont traités dans le même module d'un logiciel, il est intéressant d'être en mesure de combiner plusieurs approches de la PPA, plusieurs langages d'aspects dédiés [86, 92] ou un langage générique avec plusieurs langages d'aspects dédiés comme le cas d'AspectJ avec COOL.

1.2 Problématique

Dans le domaine du développement des logiciels par aspects, on trouve peu d'espaces de conception des langages d'aspect, ce qui pourrait améliorer leur qualité en termes d'intelligibilité, de réutilisation et de maintenabilité. En outre, la complexité des techniques utilisées pour implémenter les langages d'aspects rend difficile la définition et le test de nouvelles fonctionnalités et de sémantiques alternatives tandis que la diversité de ces techniques rend difficile la composition de ces langages. Le problème peut être décomposé en trois parties :

1. Le développement d'un langage d'aspects dédié peut être délicat. La première étape du processus de conception de ce type de langage est de considérer les schémas (de conception et de programmation) communs au domaine de la préoccupation traité par le langage. Cette étape risque de prendre beaucoup de temps. Compte tenu de ce fait, la mise à disposition d'un prototype de ce langage est très utile pour tester les fonctionnalités proposées. Le prototype doit être évolutif et maintenable dans le but d'ajouter de nouvelles fonctionnalités au langage capturé.

2. L'expérimentation avec diverses variantes de sémantiques et l'exploration de l'espace de conception des langages d'aspects sont communes dans le domaine de recherche des langages d'aspects. Ainsi, la sémantique des coupes en AspectJ a subi des évolutions depuis la première version (par exemple la coupe `execution` vis-à-vis de l'héritage [18]). Malheureusement, l'exploration de ces variantes n'est pas facile car la plupart des compilateurs existants (par exemple, le compilateur d'AspectJ, *ajc* [2]) n'ont pas été conçus avec l'extensibilité comme objectif. Même si c'est le cas de *abc*, *AspectBench Compiler for AspectJ* [15]. Travailler avec

cet outil nécessite la maitrise de tous les composants de *abc* comme Polyglot, Soot, Jimple, etc. En plus, il est plus facile de proposer des variantes de la sémantique des coupes que de modifier le mécanisme de l'*ordonnancement* des aspects.

3. Lors du travail avec des grosses applications, il faut considérer des préoccupations différentes et spécialement des préoccupations transversales telles que la synchronisation, la concurrence, etc. L'utilisation de plusieurs langages dédiés (un langage par domaine) nous amène à composer les différents langages utilisés ensemble. La diversité des techniques d'implémentation de ces langages rend presque impossible leur composition. Pour clarifier le problème, considérons les deux langages d'aspects, AspectJ et COOL. Pour chacun de ces deux langages, il y a un compilateur qui tisse les aspects correspondants. Le compilateur prend une application de base et génère une représentation spécifique de celle-ci puis cherche les lieux où les préoccupations transversales doivent être ajoutées. Lors de la construction de la représentation spécifique, chaque compilateur ajoute, dans le programme transformé, certaines instructions spécifiques à l'implémentation, qui peuvent être considérées comme des instructions de base pour le second tisseur (puisque le tissage est séquentiel). Cela conduit à un comportement inattendu comme décrit dans [70]. Une infrastructure de composition doit prendre en charge à la fois la gestion des interactions entre les aspects et le paramétrage de la composition des différents langages.

1.3 Thèse

L'objectif général de cette thèse est de proposer une infrastructure pour faciliter le prototypage et la composition des langages d'aspects.

Propositions L'approche générale consiste à recourir à des interpréteurs au lieu de compilateurs (tisseurs) parce que :

1. Un interpréteur peut nous aider à mettre directement en œuvre la sémantique du langage d'aspects lors de son prototypage et à gérer facilement les interactions entre les langages lors de leur composition.
2. Un interpréteur est plus ouvert et plus extensible qu'un compilateur.
3. Il est plus facile de composer des interpréteurs que des compilateurs (tisseurs) pour composer les langages.

Notre approche consiste à définir un interpréteur commun pour les langages à aspects, interpréteur qui peut être étendu pour construire un interpréteur concret pour un langage spécifique, tandis que les extensions peuvent être assemblées afin de composer différents langages.

Nous partons de deux points existants:

1. La CASB [36], un cadre pour la définition de la sémantique des langages d'aspects comme base pour exprimer la sémantique de notre interpréteur.
2. Un métamodèle des langages d'aspects [26] qui représente les notions communes des langages d'aspects.

L'interpréteur commun met en œuvre la sémantique commune aux langages d'aspects basée sur la CASB. Comme exemple de sémantique commune, citons l'interaction entre un interpréteur de base et un interpréteur d'aspects, la sélection des aspects, l'ordonnancement des aspects, l'exécution des actions, la possibilité, au sein d'une action de redonner le contrôle au programme de base (instruction *proceed*). L'interpréteur fait abstraction de la sémantique spécifique comme la sémantique des *coupe* qui doit être spécifiée lors de l'implémentation d'un langage concret en étendant l'interpréteur. L'interpréteur commun est conçu pour être ouvert et flexible de sorte que même la sémantique commune peut être configurée.

À titre d'exemple, nous citerons la notion de l'ordonnancement des aspects, pour laquelle il est important d'avoir différents types de stratégies d'ordonnancement, dynamiques ou statiques. Le prototypage des langages d'aspects concrets est réduit à la spécification des traits abstraits dans l'interpréteur. En ce qui concerne la composition des langages d'aspects, les langages développés indépendamment par extension de l'interpréteur commun peuvent être facilement assemblés. La composition des langages d'aspects est réduite à la composition des aspects, car les aspects de tous les langages héritent de la notion d'aspect abstrait et interagissent de la même façon avec l'interpréteur. Le cadre offre une configuration par défaut de la composition des aspects et un support pour la configuration de cette composition.

Prototypes La sémantique de la CASB suppose qu'un interpréteur de base met en œuvre la sémantique du langage de base et la partie commune de l'interpréteur doit gérer les interactions entre les deux interpréteurs (interpréteur de base et interpréteur des aspects). Selon l'interpréteur de base que nous considérons et en suivant nos propositions, nous avons construit deux prototypes :

1. MetaJ [40] étendu. Utiliser un interpréteur de base comme la JVM est très complexe. Notre point de départ pour construire l'interpréteur est de prendre MetaJ, un interpréteur écrit en Java pour un sous-ensemble de Java, et de l'étendre avec un interpréteur d'aspects.
2. CALI. Pour appliquer la même architecture à l'ensemble de Java, sans faire de changements dans la JVM, nous utilisons AspectJ pour mettre en œuvre les points de jonction et les transmettre à une mince couche d'interprétation responsable de la gestion spécifique des aspects. Cela peut être considéré comme un exemple intéressant d'intégration des interpréteurs et des compilateurs, des mondes dynamique et statique. Le cadre qui en résulte est appelée CALI, pour *Common Aspect Language Interpreter*.

1.4 Validation

Pour valider notre travail, nous présentons un prototype d'un sous-ensemble significatif d'AspectJ implémenté en utilisant CALI. L'extensibilité de cette implémentation d'AspectJ est validée par deux variantes de ce langage, l'une traite l'ordonnancement dynamique des aspects [14] et l'autre une *sémantique dynamique* des coupes prédéfinies sélectionnant les appels et les exécutions de méthodes [13]. CALI est également utilisé pour prototyper des langages d'aspects dédiés très différents comme EAOP et COOL. CALI a été utilisée au sein du projet AMPLE [1] pour prototyper ECaesarJ [11], que l'on peut voir comme une intégration du modèle EAOP et de CaesarJ.

Les implémentations reposant sur CALI peuvent être facilement composées parce qu'elles sont toutes basées sur le mêmes langage abstrait. En outre, CALI prend en charge la configuration de la composition et la résolution des interactions entre aspects à différents niveaux (conception et programmation).

1.5 Structure de la thèse

La section 2 décrit des techniques d'implémentation des langages de programmation en général et spécifiquement langages d'aspects ainsi que les approches existantes pour prototyper et composer des langages d'aspects. La section 4 décrit l'interpréteur commun pour les langages d'aspects, appelé CALI. La section 5 montre comment une grande partie d'AspectJ est implémentée avec CALI. Dans la section 6 nous montrons comment des petites modifications dans le prototype d'AspectJ nous permet d'implémenter facilement deux variantes de ce langage. La section 7 montre l'implémentation des langages autres qu'AspectJ avec CALI comme EAOP, COOL. La section 9 traite des travaux connexes et des perspectives de nos travaux puis conclut.

2 Etat de l'art

Cette section décrit ce qu'est un langage d'aspects, comment il est défini et quelles sont les principales techniques utilisées pour sa mise en œuvre.

2.1 Langages d'aspects

2.1.1 Types de PPA

La PPA consiste à modulariser le code dispersé et mélangé dans le code du logiciel puis laisser l'infrastracture du langage s'occuper de *tisser* les aspects. Selon la nature du code dispersé, on peut distinguer deux types de PPA:

Structurelle signifie le tissage des modifications dans la *structure statique* (classes, interfaces, etc.) du programme. En AspectJ, ce type est appelé *Inter-type declarations* [94]. Les aspects d'AspectJ peuvent déclarer des membres (des champs, des méthodes et des constructeurs) pour qu'ils soient détenus par d'autres types. Ils peuvent également déclarer que d'autres types implémentent de nouvelles interfaces ou étendent une nouvelle classe.

Considérons un example qui consiste à ajouter une fonctionnalité partagée par certaines classes existantes qui font déjà partie d'une hiérarchie de classes, c'est à dire qu'elles ont déjà hérité d'une classe. Dans Java, on crée une interface qui prend compte de cette nouvelle capacité, puis ajoute pour chaque classe concernée une méthode qui implémente cette interface. AspectJ peut exprimer la préoccupation en un seul endroit, en utilisant des déclarations inter-type. L'aspect déclare les méthodes et les champs qui sont nécessaires pour mettre en œuvre la nouvelle fonctionnalité, et associe les méthodes et les champs aux classes existantes.

Comportementale signifie le tissage de nouveaux comportements dans l'exécution d'un programme. Il augmente ou même remplace le flux de l'exécution du programme d'une manière qui franchit les limites des modules, ce qui modifie le comportement du système. AspectJ fournit la notion de `pointcut` pour définir les points concernés dans le flux d'exécution du programme.

2.1.2 Technique d'implémentation

Un langage d'aspects est normalement une extension d'un langage de programmation existant, appelé langage de base, avec lequel sont exprimées les préoccupations de base du programme. L'extension du langage existant est à la fois:

Syntaxique Pour ajouter les expressions nécessaires à la déclaration des coupes et des greffons. Cependant chaque langage définit sa propre syntaxe. Dans AspectJ, une coupe est définie avec un langage de patterns qui permet d'indiquer où l'aspect doit être intégré dans l'application en utilisant des quantificateurs (`call`, `execution`, etc.), des opérateurs booléens et des caractères joker ou *wildcards* (comme le caractère *). Les langages d'aspects dédiés ajoutent des instructions plus spécifiques au domaine auquel ils s'intéressent. Par exemple, COOL [68] utilise les mots clés: `selfex` et `mutex` pour la coordination des méthodes.

Sémantique Pour exprimer comment le programme de base se réagit en présence des aspects ainsi que d'autres spécifications comme l'interaction entre les aspects et l'ordonnancement de l'exécution des aspects.

Du point de vue utilisateur, différents modules de préoccupations indépendantes sont écrits. Du point de vue du concepteur du langage, certains de ces modules recoupent d'autres modules et doivent être tissés avec eux aux endroits appropriés. Il existe deux façons différentes pour implémenter les langages d'aspects:

1. Un programme combiné écrit avec le langage de base est produit à partir du programme de base et les aspects. Le code aspect est tissé dans le code base et les aspects n'existent plus à l'exécution et ne sont pas considérés comme des entités de première classe.
2. L'interpréteur et l'environnement du langage de base sont mis à jour pour supporter et implémenter des fonctionnalités de la PPA. Dans cette approche, les aspects sont des entités de première classe.

2.2 Prototypage et composition

Comme déjà mentionné, il peut être avantageux d'utiliser plusieurs langages d'aspects, surtout dédiés au sein de la même application. Il y a deux approches de la composition des langages d'aspects: *transformation par étapes* et la *transformation directe.*

Transformation par étapes Cette approche consiste à tisser les aspects d'un langage avec le tisseur correspondant puis à faire circuler le programme résultant vers le tisseur d'un autre langage et ainsi de suite.

Transformation directe Cette approche consiste à traduire les programmes d'aspects dans les différents langages (T_1 et T_2 vers un langage commun T_3). Dans ce cas, la composition de deux aspects, respectivement de T_1 et T_2, se reduit à la composition de deux aspects dans T_3, où la sémantique de la composition de deux aspects devraient être définie.

Un framework de prototypage des langages d'aspects doit fournir une abstraction commode pour implémenter un nouveau langage d'aspects. Aujourd'hui, il n'existe que quelques frameworks pour le prototypage et la composition des langages d'aspects. La plupart d'entre eux utilisent un mécanisme de transformation générale. Le prototypage d'un nouveau langage consiste à traduire les aspects en terme de transformation générale prévue par le framework. Par exemple, implémenter un nouveau langage avec Reflex [92] consiste à transformer les programmes en des *classes de configuration* de Reflex. XAspects consiste à transformer les aspects décrits avec des langages dédiés en AspectJ. En général, un framework pour le prototypage des langages d'aspects fournit un support pour composer des langages d'aspects différents implémentés avec ce framework en composant différents modules traduits dans la même représentation intermédiaire.

2.2.1 Classification des approches existantes

ajc Le compilateur AspectJ *ajc* a un front-end et back-end. Le front-end traduit les aspects décrits en AspectJ `.java` et `.aj` en classes annotées (bytecode). Un aspect est traduit en une classe de même nom et une action en une méthode de cette classe. Cette méthode compilée est également annotée avec des attributs stockant les données spécifiques à l'aspect (par exemple, les déclarations des coupes). Les annotations distinguent les classes ≪ aspects ≫des classes Java et servent au back-end pour tisser le code des actions dans le code des classes Java.

Le back-end implémente la sémantique de l'extension aspect. Il est basé sur une approche de transformation. Il prend les classes générées par le front-end, et instrumente le bytecode en injectant au niveau des points de programmes correspondants à des points de jonctions possibles, calculés par le front end, des appels aux actions (des méthodes des classes générées à partir des aspects).

abc Le compilateur *abc* est une autre implémentation complète d'AspectJ. Il a été conçu pour faciliter l'extension et l'optimisation du langage AspectJ.

La grammaire AspectJ développée pour *abc* est spécifiée comme une extension de la grammaire de Java et les grammaires des extensions sont conçues comme des modifications de la grammaire d'AspectJ.

Le front-end génère un arbre de syntaxe abstraite et une structure appelée *AspectInfo* contenant les informations liées aux aspects à partir des fichiers source.

Il y deux raisons pour dire que *abc* est plus extensible que *ajc* :

1. *ajc* effectue sa transformation au niveau du bytecode et génère du bytecode optimisé pour être exécuté sur la JVM alors que *abc* effectue sa transformation au niveau du code Jimple qui est une représentation intermédiaire de plus haut niveau que le pseudo-codedonnant encore plus d'abstraction à la transformation ce qu'il le rend plus extensible que *ajc*.
2. L'outil Soot utilisé dans *abc* est utile pour l'implémentation des extensions telles que les coupes décrivant des points précis dans le graphe de flot de contrôle.

On peut dire que *abc* se caractérise par une transformation décrite à un assez bon niveau d'abstraction qui peut être modifiée à l'aide d'outils génériques en vue d'étendre AspectJ. Toutefois, l'utilisation de *abc* pour étendre AspectJ nécessite la connaissance et la maîtrise de tous les outils utilisés dans *abc* comme Polyglot, Soot, Jimple, etc. alors qu'un lun framework extensible léger permettant le prototypage rapide suffit pour prototyper AspectJ. En outre, *abc* n'a pas été conçu pour permettre la composition de AspectJ avec autres langages d'aspects.

Reflex et XAspects Comme *abc*, Reflex et XAspects utilisent une approche de transformation pour implémenter leurs mécanismes aspect. Au lieu d'avoir une transformation générique, Reflex et XAspects définissent une transformation intermédiaire. Chaque fois qu'un nouveau langage doit être défini en utilisant Reflex, une transformation de la sémantique du langage aux classes de configuration et aux classes de metaobjects, qui sont les paramètres du transformation effectuée par Reflex sur le bytecode en utilisant l'outil Javassist [8]. XAspects traduit les aspects des différents langages aux aspects AspectJ où la transformation peut se faire en utilisant n'importe quel tisseur AspectJ.

Pluggable AOP À la différence d'autres propositions examinées, n'utilise pas une démarche de transformation. Pluggable AOP propose un cadre pour définir les langages d'aspects sous la forme d'interpréteurs composables. Un langage unique est constitué d'un interpréteur du langage de base et d'un interpréteur des aspects. La composition de plusieurs langages se résume à la composition des interpréteurs des aspects qui sont composés en couche. Le problème de cette composition est qu'elle impose un ordre unique d'évaluation qui n'est pas souhaitable en général.

3 Évaluation et Contributions

Le but principal d'un framework de prototypage des langages d'aspects est de permettre aux concepteurs de tester de nouvelles fonctionnalités d'une façon simple. L'origine de cette simplicité doit être la généralité du mécanisme d'aspects utilisé dans ce framework afin d'autoriser la projection des langages d'aspect vers ce mécanisme. Par exemple, le mécanisme d'aspects de Reflex repose sur la réflexion partielle du comportement alors que le mécanisme utilisé dans XAspects aspect est celui d'AspectJ. Les approches de transformation souffrent de deux problèmes:

1. Le saut en abstraction entre la sémantique du langage et les mécanismes d'implémentation des frameworks de prototypage complique la construction et l'extension du prototype.
2. En se basant sur le mécanisme d'aspects supporté par le framework, il est difficile de mettre en œuvre plusieurs alternatives de sémantique du langage. Par exemple, le plugin AspectJ de Reflex représente un point de jonction de type `call` comme une instance de la classe `MsgSend` qui, dans Reflex, a une sémantique bien définie ce qui rend difficile la proposition d'alternatives pour la coupe `call` similaires à celles de [18].

Ces problèmes dûs à l'approche de transformation de code nous conduisent à concevoir un mécanisme d'aspect extensible et de construire des langages d'aspects en étendant ce mécanisme au lieu de traduire différents langages vers une même représentation. L'étude des fondements des langages aspect peut se faire en considérant les changements à apporter aux interpréteurs classiques des langages (interpréteurs de base) pour introduire les aspects [46]. En général, l'écriture d'un interpréteur nécessite beaucoup moins de travail que l'écriture d'un back-end. En outre, il permet de prototyper, tester et étendre rapidement le langage.

Notre approche consiste à hiérarchiser les niveaux d'abstraction des mécanismes d'aspects en décrivant les relations entre ces niveaux : des modèles conceptuel et sémantique jusqu'au niveau de l'implémentation sous la forme d'un interpréteur. Ces relations nous fournissent une traçabilité qui permet de faciliter la compréhension de la sémantique des langages d'aspects, leur prototypage et leur composition.

Nous considérons le modèle défini dans [63] comme notre modèle conceptuel. Les quatre sous-processus décrits dans ce modèle sont au sommet de la hiérarchie. Le deuxième niveau est le modèle sémantique définie par la CASB (la base de sémantique des langages d'aspects) [36]. Pour chaque sous-processus, nous décrivons sa sémantique selon la CASB.

Nous réutilisons le métamodèle des langages d'aspects défini dans [26] sous forme d'une grammaire abstraite produisant des arbres de syntaxe abstraite tandis que la sémantique est implémentée en fonction du modèle conception et du modèle sémantique. La grammaire est abstraite parce qu'elle parle de syntaxe abstraite mais aussi puisqu'elle est incomplète et elle doit être complétée lors de l'implémentation d'un langage d'aspects concret. Les concepts sont modélisés dans le métamodèle qui représente les caractéristiques essentielles des langages d'aspects et leurs relations. Nous obtenons un langage d'aspects abstrait qui peut être étendu et spécifié, afin de modéliser un langage d'aspect concret. Chaque langage d'aspect doit être défini par une projection de ses fonctionnalités vers les concepts du métamodèle.

Nous appliquons notre approche à l'interpréteur de Java MetaJ [40]. MetaJ joue le rôle de l'interpréteur de base et interagit avec l'interpréteur d'aspects implémentant la sémantique du langage d'aspects. Nous obtenons une séparation claire entre la base et l'interpréteur aspect en définissant et en clarifiant les bornes et les interactions entre les deux interpréteurs. L'interpréteur d'aspects montre explicitement la séparation entre les quatre sous-processus. La séparation entre l'interpréteur de base et l'interpréteur d'aspects ainsi que notre mise en œuvre de chaque sous-processus facilitent la compréhension et l'extension de l'interpréteur d'aspects et forment aussi un pas vers une composition modulaire des langages d'aspects. Ceci signifie que des langages d'aspects différents, qui ont été mis en œuvre indépendamment, peuvent être assemblés sans modifier chacune des implémentations (*third-party composition*).

Les fonctionnalités spécifiques des langage d'aspects concrets peuvent être partiellement décrites comme des spécialisations des concepts décrits dans le métamodèle commun. L'approche du framework garantit également que tous les langages d'aspects seront issus du même métamodèle. En guise de validation, nous avons réalisé un prototype d'une version allégée d'AspectJ par la mise en œuvre de plusieurs coupes étendant une partie essentielle du métamodèle, les *sélecteurs de points de jonction.*

Après la mise en œuvre du framework pour une partie de Java, nous avons appliqué les mêmes principes à de Java en réutilisant un mécanisme appelé *tissage en deux étapes*, qui simplifie la construction de notre framework sur l'ensemble de Java sans avoir besoin de traiter des problèmes d'efficacité et de performance, mais en se concentrant sur la conception et l'extensibilité de l'interpréteur. Nous avons nommé ce framework CALI pour *Common Aspect Language Interpreter.*

4 Un interpréteur extensible pour les langages d'aspects

Une bonne façon de comprendre les principes des langages d'aspects basés sur un langage de base est de prendre un interpréteur de ce langage puis étudier quels sont les modifications à y apporter pour ajouter/introduire la notion des aspects. Nous considérons MetaJ, un interpréteur de Java écrit en Java [39, 40], comme notre interpréteur de base. Nous montrons les modifications (instrumentation et extension) pour obtenir un interpréteur pour les langages d'aspects. Nous gardons ces changements modulaire autant que possible en séparant entre l'interpréteur de base et l'interpréteur d'aspects. L'interpréteur d'aspects sera à son tour divisé en deux parties : l'une appelée *Plateforme* pour les traits communs aux langages d'aspects et l'autre pour les traits spécifiques à chaque langage. La partie spécifique doit fournir une *interface* pour pouvoir être appelée à partir de la partie commune. La séparation entre la base et l'interpréteur d'aspects améliore la compréhension et l'extensibilité de chaque interpréteur. L'architecture générale montre (Figure 5.3) chaque interpréteur en tant qu'un composant indépendant et affiche les interactions entre les deux interpréteurs. Les interactions entre les deux éléments se composent de deux types de message :

- Un point de jonction émis par l'interpréteur de base à chaque étape de l'évaluation des expressions.
- Les données résultant de l'évaluation du point de jonction dans l'interpréteur d'aspects par les aspects existants.

Les actions du langage d'aspects peuvent être écrites ou non en partie dans le langage de base. Le langage d'aspects peut être une extension du langage de base ou non. Comme architecture, il y a trois possibilités :

(a) L'interpréteur d'aspects est une extension complète de l'interpréteur de base et les actions sont interprétées dans l'interpréteur d'aspects. Les deux interpréteurs partagent l'objet *heap*.

(b) L'interpréteur d'aspects délègue l'évaluation des expressions de base (des actions) à l'interpréteur de base. Ce cas n'est qu'une optimisation du premier.

(c) L'interpréteur d'aspects est indépendant de l'interpréteur de base. C'est le cas où le langage d'aspects est indépendant de celui de base (l'action du greffon ne contient aucune expression du langage de base).

En utilisant cet approche, nous avons réalisé une version légère d'AspectJ par l'extension de l'interpréteur d'aspect. L'implémentation des coupes consiste à évaluer le point de jonction généré par l'interpréteur de base.

4.1 CALI

Pour appliquer la même architecture sur l'ensemble de Java, CALI réutilise AspectJ pour effectuer une première étape de tissage statique, que nous complétons par une seconde étape de tissage dynamique, implémenté à travers d'une couche d'interprétation. Ceci peut être considéré comme un exemple intéressant de concilier les interpréteurs et les compilateurs. Le modèle conceptuel de CALI est toujours celui de 4 sous-processus alors que le modèle sémantique est basé sur une version modifiée de la CASB.

4.1.1 Sémantique

Nous nous sommes basés sur la CASB comme framework de définition de la sémantique pour définir les mécanismes d'interprétation des points de jonction dans CALI. Le core de la CASB était étendu pour introduire la notion de groupe d'aspects qui est utilisé afin de modifier le mécanisme standard de planification des aspects dans AspectJ.

$$\textsc{Around}\ \frac{\psi(i) = \Phi \qquad \alpha(\Phi, \Sigma) = (\phi, \Phi')}{(i : C, \Sigma, P) \rightarrow (\phi : \texttt{pop} : C, \Sigma, (\Phi' :: [\bar{i}]) : P)}$$

Nous avons introduit la fonction α, non présente dans les règles de la CASB, pour montrer qu'il y aurait la possibilité d'effectuer un `ordonnancement dynamique` qui pourra être nécessaire dans certains langages d'aspects implémenté au dessus de CALI. En effet, dans nos règles étendues, la fonction α, choisit éventuellement, en se basant sur Σ, la première instruction ϕ ainsi que les instructions Φ' afin d'examiner si l'aspect procède.

4.1.2 Architecture

CALI représente un bon exemple qui mélange la compilation et l'interprétation. Le programme de base fonctionne comme un programme Java normal sur une JVM standard tandis que les aspects sont partiellement interprété. Tous les aspects (de n'importe quel langage implémenté avec CALI) doivent être traduits, en utilisant un parseur, vers une structure spécifique (en Java) définie par l'interpréteur, et seront représentés à l'exécution par des objets Java conformes à la présente structure spécifique. Les deux étapes de tissage sont les suivantes :

1. La première étape a lieu dans AspectJ lors de la compilation, à travers d'un aspect appelé `Platform`, et dont le résultat est l'instrumentation par AspectJ de tous les points de jonction possibles dans le programme de base;
2. La deuxième étape a lieu à l'exécution, lorsque le greffon de l'aspect `Platform`, qui se comporte comme une *couche d'interprétation*, évalue le point de jonction courant (accessible par `thisJoinPoint`) en cherchant des aspects qui séléctionne le point de jonction courant puis les exécute. Le code des greffons écrits en Java sont exécuté comme du code Java normal.

L'aspect `Platform` implémente les 4 processus de tissage. Le processus `reify` est implémenté comme une coupe qui génère tous les points de jonction possible. Les processus `match`, `order` et `mix` sont implémentés comme des méthodes et sont appelées dans le greffon de `Platform`.

4.1.3 Langage d'aspects abstrait

CALI fournit une API qui doit être utilisée par les différents langages d'aspects utilisant CALI. L'API est un langage d'aspects abstrait basé sur le métamodède des langages d'aspects `MetaSpin` [26]. Les principaux éléments (classes et interfaces) sont : `Aspect`, `JoinPointSelector`, `Advice` et `SelectorAdvicebinding`.

Aspect Une interface qui doit être implémenté par les classes qui représente un aspect dans les différents langages. Elle permet à `Platform` de communiquer les points de jonction aux aspects.

JoinPointSelector Une classe qui représente un évaluateur des points de jonction pour décider la sélection ou non du point de jonction courant.

Advice Une classe contenant la méthode `adviceexecution` qui sera appelée lors de l'évaluation du test dynamique.

SelectorAdviceBinding Une classe permettant de relier le sélecteur (instance de la classe `JoinPointSelector`) à l'action correspondant (instance de la classe `Advice`).

4.2 Conclusion

Dans cette section, nous décrivons comment le prototypage des langages d'aspects se fait en utilisant CALI. Le concepteur de langage doit fournir :

– L'implémentation d'un aspect comme une classe implémentant l'interface `Aspect`, en particulier la méthode `staticTest (Joinpoint)`.
– La mise en œuvre des sélecteurs qui héritent de la classe `JoinPointSelector`. Chaque sélecteur doit implémenter les deux méthodes `boolean staticTest (Joinpoint)` et `boolean dynamicTest (Joinpoint)`. La première vérifie si le sélecteur identifie la partie statique du point de jonction. Cela correspond aux pièces qui peuvent être déterminées au moment de la compilation, même si ce rapprochement des points de jonction se fait par interprétation lors de l'exécution, car l'instance de `Joinpoint` contient plusieurs éléments d'information statique (accessible par `getStaticPart()`). Le deuxième définit si le sélecteur identifie la partie dynamique en accédant à des informations d'exécution en utilisant des méthodes comme `getThis()`, `getTarget()`, etc. fournies par l'interface `Joinpoint`.
– La mise en œuvre des actions. Il y a deux cas : dans le premier, le langage des actions est aussi Java, le concepteur peut utiliser l'implémentation des actions prédéfinis dans CALI alors que dans le deuxième, le langage des actions n'est plus Java, le concepteur implémente la méthode `adviceexecution` à titre d'évaluateur des expressions du langage des actions du nouveau langage.

L'implémentation des sélecteurs comme des entités indépendantes et modulaire améliore leur réutilisation. Ceci apparaît bien avec nos versions de EAOP (pour Java) et AspectJ où nous avons réussi à réutiliser les mêmes sélecteurs et greffons, mais avec une implémentation diffèrente de la méthode `staticTest (Joinpoint)` des aspects. Pour AspectJ, cette méthode retourne une liste des `SelectorAdviceBinding` qui, statiquement sélectionne le point de jonction. Pour EAOP, elle retourne un seul `SelectorAdviceBinding` en fonction de l'état d'aspect. Pourtant, les deux langages partagent les sélecteurs `Call`, `Execution`, `This`, `Target`, etc. Une fois que AspectJ est implémenté, la mise en œuvre de EAOP ne nécessite que la gestion de l'état de l'aspect.

5 Implémentation d'AspectJ

Après avoir défini la manière de conception des langages d'aspects sur CALI dans la section précédente, celle-ci présente une implémentation d'AspectJ à l'aide CALI. L'implémentation comprends la plupart des caractéristiques d'AspectJ . Le couple (*shadow, residue*) est le principe de sélectionner un point de jonction dans AspectJ. Cela se traduit par une sélection avec les deux méthodes `staticTest` et `dynamicTest`, où un sélecteur de point jonction traite la sélection statique (telle que pratiquée dans la sémantique par la fonction Ψ) ainsi que dynamique correspondant (telle que pratiquée par les instructions Φ). Cette sélection se fait en additionnant les résultats issus de ses composantes qui sont des sélecteurs primitives, instances de la classe `JoinPointSelector`. Chaque coupe est implémentée comme une sous-classe de `JoinPointSelector` avec une définition appropriée des méthodes `staticTest` et `dynamicTest`, conformément à la spécification de la sémantique d'AspectJ [12, 16, 18].

La méthode `staticTest` retourne `true` si l'information statique (nom de la méthode, type statique, le nom du champs, etc.) contenue dans le point de jonction, est identifiée par la coupe alors que la méthode `dynamicTest` traite les information de l'exécution contenues dans le point de jonction comme : type dynamique (type de l'objet *this*), type de récepteur (*receiver*), etc.

Pour mettre en œuvre les sélecteurs, nous nous sommes basés sur la variable de référence spécifique, `thisJoinPoint` fournit par AspectJ et accessible à partir du corps du greffon. Elle contient des informations réflectives sur le point de jonction courant. Le greffon de l'aspect `Platform` passe cette information réifiée à tous les sélecteurs d'aspects, qui peuvent alors accéder à toutes les informations nécessaires pour la sélection des points de jonction.

5.1 Implémentation des coupes

5.1.1 Call

Toutes les informations nécessaires à un sélecteur `Call`, afin de déterminer si un point de jonction doit être sélectionné ou non, sont statiques (peuvent être obtenues à partir du *shadow* de point de jonction). En effet, il y a deux conditions pour qu'un sélecteur `call(`P`.`m`())` sélectionne un point de jonction $e.m()$ où J est le type de qualification du point de jonction (type statique de e) :

- $J <: P$ (J est un sous classe de P);
- m existe dans P.

En conséquence, les conditions de sélection sont implémentées dans la méthode `staticTest`, alors que la méthode `dynamicTest` toujours retourne `true` :

```
public boolean staticTest(JoinPoint jp) {
  boolean basicMatch;
  if (exists) {
    boolean basicMatch =
      jp.getKind().equals("method-call")
           && jp.getSignature().getName().equals(methodName)
           && Arrays.equals(((MethodSignature)jp.getSignature())
           .getParameterTypes(), parameterTypes);
    if (basicMatch) {
      try {
        jp.getSignature().getDeclaringType().asSubclass(pointcutClass);
        return true;
      } catch (Exception e) {
        return false;
      }
    } else
      return false
    } else
    return false;
  }
```

En utilisant l'API de réflexion de Java, la méthode `staticTest` implémente la sémantique de `Call`. La variable booléenne `exist` a la valeur true si la méthode existe dans la classe définie dans la coupe. La variable booléenne `basicMatch` est vrai lorsque le point de jonction est en effet un point de jonction de type appel de la méthode décrite dans la coupe. Cela nécessite de comparer le nom et les types de paramètres spécifiés dans la coupe et celles obtenues à partir du point de jonction.

Si les deux variables `exist` et `basicMatch` sont à `true`, la dernière étape consiste à vérifier, en utilisant la méthode `asSubClass` de l'API de réflexion de Java, que le type de déclaration du point de jonction `jp.getSignature().getDeclaringType()` est une sous-classe de la classe `pointcutClass` déclarée dans la coupe. Cela correspond directement à la condition $J \subseteq P$ dans la sémantique.

5.1.2 Execution

En analysant la sémantique de `execution`, on obtient les mêmes conditions que la coupe `call`, mais avec une définition spécifique de type de qualification. L'implémentation du sélecteur `Execution` diffère de celui par le fait de tester si le type du point de jonction est "method-execution".

5.1.3 This

La coupe `this`(*Type*) sélectionne les points de jonction où l'expression `this instanceof` *Type* retourne `true`. La sélection des points de jonction est donc basée sur des informations dynamique dans le contexte de l'exécution. Ceci veut dire que la méthode `staticTest` toujours retourne `true` alors que la méthode `dynamicTest` est implémentée de la façon suivante :

```
public boolean dynamicTest(JoinPoint jp) {
  return pointcutClass.isAssignableFrom(jp.getThis().getClass());
}
```

5.1.4 Target

L'implémentation de cette coupe est similaire à celui de `this` avec la différence que c'est la méthode `getTarget()` qui est appelée sur le point de jonction au lieu de `getThis()` :

```
public boolean dynamicTest(JoinPoint jp) {
  return pointcutClass.isAssignableFrom(jp.geTarget().getClass());
}
```

5.2 Résumé

L'implémentation d'AspectJ avec CALI possède les propriétés suivantes :

Extension directe de CALI elle est conforme à la méthodologie discutée dans la section précédente.

Extensibilité de la sémantique elle permet facilement de tester des sémantiques alternative d'AspectJ surtout la sémantique des coupes et des planification des aspects.

6 Extensions d'AspectJ

6.1 Planification dynamique des aspects

Cette contribution a fait l'objet de notre papier [14]. Il s'agit d'une nouvelle version d'AspectJ, appelé Dynamic AspectJ [14]. Elle permet de planifier l'exécution dynamique des aspects à un point de jonction partagé, y compris la possibilité d'annuler des aspects.

6.1.1 Exemples

Nous présentons deux exemples illustrants les avantages de la planification dynamique des aspects. Le premier exemple présente un cas qui necessite un déploiement dynamique et l'annulation des aspects, tandis que le deuxième demande une réorganisation des aspects à l'exécution.

Annulation des aspects Dans [51], Hannemann et Kiczales mentionnent qu'avec des aspects, certaines patrons de conception [49] disparaissent parce qu'ils ne sont pas plus nécessaires. Il peut sembler que c'est le cas du patron décorateur, mais ce n'est pas vraiment le cas. Cela a même conduit à des régimes assez complexes, comme ceux proposés dans [80], qui reposent sur l'enregistrement des objets décorés et de laisser les greffons vérifier l'objet cible. Du code additionnel est nécessaire pour l'enregistrement et l'annulation de l'inscription décorateurs et la complication ici est que certains morceaux de code comme `Decorator.aspectof())` ne sont pas accessibles en dehors de l'aspect. Une autre façon de regarder le problème est plutôt à considérer que la question est qu'il n'est pas possible de déployer dynamiquement et éventuellement d'annuler la décorateurs qui ne devrait pas s'appliquer.

Réorganisation des aspects Le deuxième exemple consiste à une application client-serveur développée par une entreprise pour l'hébergement de fichiers. Les clients téléchargent des fichiers via la méthode `Send`. Du côté serveur, un anti-virus, appelé par l'intermédiaire du méthode `virusCheck`, vérifie que les fichiers reçus ne contient pas de virus. La question est alors de mettre à jour l'application d'une manière modulaire en ajoutant deux préoccupations, chacun étant implémenté comme un aspect :

```
public aspect CompressUpload {
  before(File f):
    call(* Client.send(..)) && args(f) {
      zip(f);
    }
  before(File f):
    call(* Server.virusCheck(..)) && args(f) {
      unzip(f);
    }
}

public aspect SecureUpload {
  before(File f):
    call(* Client.send(..)) && args(f){
      encrypt(f);
    }
  before(File f):
    call(* Server.virusCheck(..)) && args(f) {
      decrypt(f);
    }
}
```

La première fonctionnalité améliore la performance en compressant le fichier chez le client avant de l'envoyer. Du côté serveur, l'anti-virus doit décompresser le fichier pour l'analyser. La deuxième caractéristique améliore la sécurité grâce au chiffrement de fichiers côté client avant de l'envoyer. Du côté serveur, le fichier est décrypté avant d'être analysé par le virus-checker.

Le client peut télécharger différents types de fichiers comme des images, texte, etc. Le but de la compression est d'améliorer les performances de transfert de fichiers, par conséquent, il est préférable d'utiliser des algorithmes de compression différents selon le type de fichier. Par exemple, un algorithme de compression d'images est meilleure qu'un algorithme de compression régulière pour diminuer la taille d'un fichier image. Nous supposons que la méthode `zip(file f)` applique l'algorithme correspondant à chaque type de fichier :

– Pour les fichiers images : *zip* $\rightarrow$ *encrypt* $\rightarrow$ *send*.
– Pour les autres formats : *encrypt* $\rightarrow$ *zip* $\rightarrow$ *send*.

6.2 Sémantique alternative des coupes d'AspectJ

Des sémantiques alternatives pour les coupes `call` et `execution`, semblant plus faciles à saisir ont été proposées. En particulier, la sémantique dynamique de *Barzilay et al.* apparait attractive. La correspondance est toujours fondée sur le type dynamique et la simple condition que la méthode doit exister dans le type déclaré est utilisé pour les coupes `call` et `execution`.

6.2.1 Sémantique dynamique pour Call

Les conditions sont identiques à ceux de la sémantique statique, sauf que nous avons besoin de remplacer le type de déclaration du coupe par son dynamique type, c-à-d `jp.getSignature().getDeclaringType()` par `jp.getTarget().getClass()`. En outre, on doit mettre cette partie de l'évaluation dans la méthode `dynamicTest` puisque elle prend en considération des informations qui dépendent du contexte de l'exécution.

6.2.2 Sémantique dynamique pour Execution

Les principes sont les mêmes que précédemment en remplaçant le type "method-call" par "method-execution".

7 EAOP et langages dédiés avec CALI

7.1 EAOP

A première vue, le modèle conceptuel de EAOP consiste à modéliser un aspect EAOP comme une machine à états où une déclaration $C \triangleright I$ est associé à chaque transition. Figure 9.1 a) représente un modèle conceptuel. Cet aspect commence dans un état d'attente pour les points de jonction. La détection d'un point de jonction et l'exécution des actions correspondant de l'aspect mènent l'aspect à l'autre État. Cet état plus tard, conduit à un choix entre deux $C \triangleright I$. L'état de l'aspect évolue en fonction du point de jonction apparaissant en premier, `logout` ou `query`.

Un point clé dans la mise en œuvre de EAOP est la représentation de l'aspect État. Il convient de noter que l'état de l'aspect évolue à l'étape suivante après l'exécution de l'avis, et pas seulement après la jointure correspondant à point. En outre, l'aspect doit avoir accès à $C \triangleright I$ correspondant à chaque état. Pour cette raison, nous décidons de joindre toutes les informations sur la poursuite de l'aspect de l'état. Cela signifie que le choix entre la liste des $C \triangleright I$ qui ont pour but d'intercepter le point de jonction sero jointe à l'état. Au lieu de fixer un $C \triangleright I$ à chaque transition, la transition entre les états est représentée par l'évènement correspondant à la fin de l'exécution des greffons. Figure 9.1 b) montre la représentation de l'aspect exploitation forestière avec le modèle modifié.

La règle de base $C \triangleright I$ peut être aisément mise en œuvre par un selecteur/greffon : une coupe transversale est implémenté comme un sélecteur et une action en tant que greffon. Avec notre approche, il est possible de réutiliser les coupes de AspectJ implémenté avec CALI. Chaque état devrait contenir les liste des sélecteur/greffon correspondant à la liste des $C \triangleright I$s qui s'attache à l'opérateur de choix.

7.2 Decorator

Havinga *et al.* proposent un langage d'aspects dédié qui permet d'appliquer le modèle décorateur tout en séparant le code portant sur le pattern Décorateur. L'implémenation de ce langage avec CALI consisté à réutiliser les sélecteurs définis dans l'implemenatation d'AspectJ et de détecter chaque association d'un decorator à un *decoratee* pour rediriger tous les appels vers le décorateur correspondant.

7.3 Memoization

Memoization est une technique utilisée pour améliorer la vitesse d'exécution du programme. Ce mécanisme consiste à retourner d'un cache la valeur précédemment traitée et déjà calculée, plutôt que de la recalculer à chaque fois.

La déclartion de l'aspect de memoization commence par mettre le nom de classe de la méthode qui doit être mise en cache après le mot-clé *cache*. L'aspect contient une déclaration *Exp* qui spécifie la méthode après la mot-clé *memoize*. La déclaration contient également des expressions d'*invalidation* qui invalident les valeurs mises en cache et imposent qu'elles seront recalculées. L'invalidation peut être effectuée après l'attribution d'un champ ou d'appeler une méthode.

L'implémentation de ce langage avec CALI consiste à créer :

- Une classe `MemoizeAdvice`.
- Une variable pour chaque déclaration `memoize` pour mettre en cache la valeur de retour de la méthode concernée.
- Un greffon pour chaque méthode concernée, en utilisant un sélecteur de type `Call` et un `MemoizeAdvice`. Le sélecteur intercepte les appels de cette méthode, et les greffons associés renvoie la la valeur mise en cache (si une est déjà stockée) ou des caches de la valeur retournée par `proceed` (s'il n'ya pas de valeur correspondante dans le cache).
- Un greffon pour chaque spécification d'invalidation qui contient une action qui invalide le cache.

7.4 COOL

COOL est un langage d'aspects dédié défini dans le cadre du framework D [68]. COOL fournit des moyens pour traiter l'exclusion mutuelle des threads, l'état de synchronisation et pour surveiller la suspension et la notification. Un aspect écrit en COOL est appelé `Coordinator`.

Chaque déclaration `selfex` est implémenté par un greffon qui lie un sélecteur de type `Call` avec une action spéciale pour les greffons selfex, appelé `SelfexAdvice` (Exemple 9.20). La déclaration `synchronized` de la méthode `adviceexecution` garantie que ce type d'actions est exécuté par un seul thread à la fois.

De même, chaque `mutex` est implémenté comme un greffon, où l'action est implémentée comme une instance de `MutexAdvice` qui gère la synchronisation tandis que le sélecteur achemine toutes les méthodes dans l'exclusion prévue au greffon.

8 Composition des langages d'aspects

Le clé de la composition des langages d'aspects construits avec CALI est l'interface commune `staticTest` qui retourne une liste de `Phi`, qui est uniformément traitée par la plate-forme. Grâce à cette interface commune, des aspects de langages différents peuvent co-exister dans la même application et communiquer de la même manière avec la plate-forme.

Nous pouvons décrire la spécification de la composition comme suit :

- Un aspect du premier langage ne doit pas avoir l'accès aux points de jonction se produisant au sein des actions du seconde langage.
- Un aspect du seconde langage seconde peut s'interesser à des points de jonction se produisant au sein des actions du premier langage.

Dans CALI, nous choisissons de ne pas réifier le point de jonction de l'aspect dans la plateforme mais nous laisser cette tâche à chaque définition de langage.

Co-advising CALI représente une approche pour résoudre ce problème à la fois au niveau du langage et au niveau du programme. Le processus `order` du plateforme est configurable pour les aspects selon le type d'aspect (niveau langage) ou par la lecture d'une spécification écrite par le programmeur (niveau programme). La syntaxe du langage de spécification, dans lequel le spécification de l'ordre des aspects, sera l'un de nos perspectives.

Foreign advising CALI résout ce problème au niveau langage en contrôlant l'application de l'aspect de L_1 sur les points de jonction générés à partir des actions de L_2. Lors de la composition des langages d'aspects implémentés avec CALI, il faut définir un fichier de configuration pour spécifier si un point de jonction d'un autre langage doit être sélectionné par un aspect.

9 Perspectives

9.1 Travaux connexes

9.1.1 JAMI

JAMI (Java Aspect MetaModel Interpreter) [53, 54, 52] et CALI ont quelques propriétés communes comme leur origine commun qui est MetaSpin, le mécanisme de tissage en deux pas et l'évaluation dynamique des aspects.

L'absence d'une sémantique formelle pour JAMI rend difficile de comprendre les flux et les mécanismes de l'interprétation utilisée. En outre, il s'appuie sur un modèle simplifié de point de jonction sans `proceed`, et n'a pas été conçu avec l'objectif de soutenir un langage générique comme AspectJ.

9.1.2 AWESOME

AWESOME [60, 64] est un framework pour la construction et la composition des extensions aspect sous forme de plugins par dessus du framework. Le point de départ réside dans son architecture qui permet la construction des tisseurs d'aspect au sommet d'une *Plateforme* de base. Une fois un tisseur est construit avec AWESOME, il peut être composé, comme la composition de tiers, avec d'autres tisseurs qui ont été construits aussi au-dessus de AWESOME.

AWESOME soutient la configurabilité de résoudre les interactions des aspects au niveau du langage, mais ne permet pas au programmeur de résoudre ces interactions entre un ensemble d'aspects. Par exemple, la composition de AspectJ et COOL, appelée COOLAJ, impose que les aspects AspectJ ou les aspects COOL doivent être exécutés en premier. Il n'y a pas de possibilité de définir (par le programmeur), une planification d'exécution des aspects en choisissant quels aspects doivent être exécutés en premier (basé sur le nom aspect par exemple et non pas sur le langage de l'aspect).

Dans AWESOME, les interactions sont gérées au moment de la compilation, ce qui lui rend incapable de résoudre les interactions dépendant des informations de l'exécution.

9.1.3 The Art of the Meta-Aspect Protocol

Dinkelaker *et al.* [35] proposent une architecture pour les langages d'aspects avec une interface explicite de la sémantique inspirée des protocoles à métaobjets afin de permettre l'adapter la sémantique du langage dynamiquement dans l'applications. Parmi les applications du protocole de métaaspect, nous pouvons citer la résolution des interactions entre les aspects qui dépendent du contexte du programme dynamique ainsi le déploiement dynamique.

L'architecture du protocole à métaaspects est très similaire à notre extension de MetaJ et fournit les mêmes capacités telles que la réorganisation dynamique et le déploiement dynamique. L'accès dynamique au pile des aspects dans AspectJ est similaire à l'accès à `MetaAspectManager`.

9.2 Perspectives

Une amélioration importante consiste à implémentater un plugin Eclipse pour Dynamic AspectJ et à améliorer le mécanisme de la réification en limitant le nombre de points de jonction générés. Cela permettrait d'améliorer la performance, tout en conservant toutes les fonctionnalités (sauf la possibilité de définir de nouveaux aspects). Ceci consiste à analyser toutes les déclarations d'aspects et de générer un aspect `Platforme` contenant une coupe `reify` qui est l'union de tous les coupes des aspects.

Nous proposons aussi d'utiliser Dynamic AspectJ pour la programmation contextuelle. Nous pouvions introduire la notion de couche (*layer*) dans Dynamic AspectJ comme un ensemble d'aspects qui sont liés en fonction du contexte. Une couche peut être activée par l'activation de ses aspects. Une caractéristique supplémentaire fournie par Dynamic AspectJ est que cette couche peut être modifié en fonction des informations du runtime (planification dynamique des aspects), alors que cette fonction n'existe pas dans les langages qui supportent ce type de programmation comme ContextL [31].

Une autre perspective pour les travaux futurs est également la notion de *context-aware aspect* [91] qui est un aspect dont le comportement dépend du contexte. CALI pourrait être utilisé pour définir un nouveau type de coupe pour contrôler la portée des aspects selon le contexte d'exécution.

CALI peut aussi aider à mettre en place un plugin pour le débogage. L'utilisation de PPA pour le débogage est bien connue. Le code pour le débogage comprend souvent des préoccupations transversales, tels que la production de messages de trace.

9.3 Conclusion

La PPA célèbre sa première décennie de recherche, de développement et d'adoption par l'industrie. Il existe de nombreux outils, des langages, et des frameworks pour la PPA.

Pour promouvoir une utilisation plus mature de PPA, de nouvelles fonctionnalités sont continuellement proposées. Ces fonctionnalités devraient être une extension des fonctionnalités existantes ou totalement nouvelles. Le réalisation de ces caractéristiques impose l'extensibilité des implémentations existantes et la fourniture d'outils pour faciliter leur implémentation.

Les compilateurs extensibles (comme `abc`) ne permettent que l'extension à grain fin à leur langage (AspectJ pour `abc`). En outre, certaines extensions simples, comme l'ajout un nouveau type de coupe ou de changer la sémantique des points de coupure existante (sémantique alternative pour `call` et `execution` ne nécessitent pas toutes les machines d'un compilateur plein Couverture comme `abc`. Lors de la première expérimentation de ces extensions, une version léger d'AspectJ a été implémenté. A ce stade, les facteurs de performance ne sont pas nécessairement un problème.

Des approches alternatives ont été proposées pour réaliser de nouvelles fonctionnalités comme des noyaux de la PPA (Reflex, XAspects) qui introduisent des primitives pour exprimer les diverses opérations de tissage. Leur principe consiste à traduire les programmes à un représentation intermédiaire commune. Malheureusement, cette façon d'implémenter les langages d'aspects nécessite encore un gros effort pour combler le fossé entre la sémantique du langage d'aspect et de la sémantique de la représentation intermédiaire. En outre, il est souvent impossible d'utiliser en toute sécurité une implémentation basée sur le noyau avec une autre. Chacune de ces implémentations introduit un code spécifique à l'implémentation et peut être considéré comme un code normal pour la d'autres implémentations.

Fournir un support pour le prototypage et la composition des langages d'aspects indépendant est un défi de taille, il donne la capacité d'accélérer l'évolution de l'aspect mécanismes par la réduction de la conception et l'essai des mécanismes nouvel aspect, car l'expérimentation de nouvelles fonctionnalités faciles fait de la validation ou l' correction du roman approches possibles.

La thèse contribue à deux axes : le prototypage et la composition des langages d'aspects. Elle consiste à :

– Guider la conception et la mise en œuvre des langages comme des extensions de l'interpréteur de langage d'aspects abstrait.
– Implémenter la composition des différents langages comme un résultat naturel de leur implémentation en utilisant le même framework.
– Supporter la résolution et la configuration des interactions entre les langages d'aspects au niveau du programme et au niveau du langage.

9.4 Prototypage

Nous contribuons à l'axe de prototypage d'un cadre pratique, fondé sur les interpréteurs. La démarche consiste à fournir un interpréteur qui implémente la sémantique d'un langage d'aspects abstrait. Le prototypage d'un langage d'aspects concret consiste à spécifier les notions abstraites de l'interpréteur.

Pour concevoir ce cadre, nous nous sommes basés sur :

1. Un framework pour la définition de la sémantique des langages d'aspects, Common Aspect Semantics Base (CASB).
2. Une représentation des notions courantes dans les langages d'aspects, le métamodèle des langages d'aspects, MetaSpin.

Chapitre 5 a montré comment construire un interpréteur pour le résumé langage d'aspect, représentée par le métamodèle, basée sur la sémantique de la CASB, en tenant compte de l'évolution d'un interpréteur de base actuel pour un sous-ensemble de Java. Les propriétés de l'interpréteur d'aspects sont les suivantes :

– Il est proprement séparé de l'interpréteur de base.
– Il implémente directement la sémantique du CASB.
– Son architecture facilite son extension pour implementer des langages d'aspects concrète.

Nous avons validé cette approche par l'extension de l'interpréteur et mettre en œuvre d'une version allégée de AspectJ.

Pour appliquer la même approche et la même architecture à l'ensemble de Java sans changer son interpréteur (JVM), nous réutilisons AspectJ pour générer les points de jonction et les transmettre à un couche d'interprétation. Chapitre 6 a décrit le framework résultant qui est appelé CALI.

Chapitre 7 a validé la démarche en décrivant une mise en œuvre d'AspectJ tandis que le Chapitre 8 a montré comment cette implémentation est facilement extensible avec deux variantes, une traite la planification dynamique des aspects [14] comme la réorganisation et l'annulation dynamique des aspects et l'autre [29] traite des sémantiques alternatives pour `call` et `execution`. Nous avons également validé notre approche en décrivant des prototypes pour EAOP, COOL et quelques langages dédiés dans le Chapitre 9. Le prototype de EAOP a été utilisé, dans le cadre du projet européen AMPLE [1], dans le prototype de ECaesarJ [83], une extension de CaesarJ [11].

9.5 Implémentation ouverte

Du point de vue de l'implémentation ouverte, on peut dire que les langages implémentés avec CALI offrent un haut niveau d'extensibilité et de reutilisabilité.

L'implémentation d'AspectJ avec CALI est très flexible. La mise en œuvre de coupe comme une implémentation directe de leur sémantique, nous donne la possibilité de répercuter directement sur les modifications apportées à la sémantique dans les implémentations. Nous avons appliqué ceci aux coupes `call` et `execution`, et on a montré que la sémantique variante pourrait être facilement implémentée.

La version de la planification dynamique a été également implémentée avec peu de changements dans l'implémentation pour donner accès au pile des aspects.

Enfin, nous avons réutilisé les implémentations des coupes d'AspectJ dans EAOP et les langages dédiés sans avoir à réinventer la roue à chaque fois.

9.6 Composition

Le deuxième axe de cette thèse est la possibilité de composer les langages d'aspects. En effet, CALI soutient, dans son noyau, la composition de plusieurs aspects. Chaque langage étend la notion d'aspect existante dans le langage d'aspect abstrait. L'interaction entre le noyau et chaque instance d'aspect est la même pour tous les langages appliqués. Le résultat est que l'interpréteur commune peut composer les aspects de différents langages.

Nous pouvons considérer que les langages implémentés ne peuvent naturellement coexister, mais le *designer* doit contrôler les *conflits*. Les deux types de conflits attendus sont *co-advising* et *foreign-advising*.

Nous donnons la possibilité de résoudre les conflits *co-advising* à deux niveaux :

Niveau langage En configurant la méthode `order` dans l'aspect `Platform` pour définir l'ordre d'exécution en fonction de leur aspect langages.

Niveau programme CALI est le seul cadre qui prend en charge la résolution des *co-advising* au niveau du programme par la configuration de la méthode `order`. Cela se fait par la lecture d'un cahier des charges définissant l'exécution aspect en fonction des aspects et non pas de leur langage. Un autre type de configuration est la planification dynamique des aspects en mettant en œuvre cette fonctionnalité, non seulement dans Dynamic AspectJ, mais comme une caractéristique de la notion de greffon de CALI.

Nous donnons la possibilité de résoudre les problèmes de *foreign-advising* au niveau langage en utilisant des aspects de configuration. Chaque aspect de ce type contrôle le champ d'application de points de jonction du langage considéré pour prévenir la sélection de ces points de jonction par un greffon d'un autre langage.

Bibliographie

[1] AMPLE project. `http://ample-project.net/`.

[2] AspectJ compiler. `http://www.eclipse.org/aspectj/`.

[3] Demeter in AspectJ . `http://daj.sourceforge.net/`.

[4] Eclipse Test & Performance Tools Platform Project.

[5] Event-based AOP (EAOP). `http://www.emn.fr/z-info/eaop/`.

[6] Java Compiler Compiler. `https://javacc.dev.java.net/`.

[7] Java Logging. `http://download.oracle.com/javase/1.4.2/docs/guide/util/logging/overview.html`.

[8] Javassist: Java programming assistant. `http://www.jboss.org/javassist`.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Wiley, 2006.

[10] Yoav Apter, David H. Lorenz, and Oren Mishali. Toward debugging programs written in multiple domain specific aspect languages. In *Proceedings of the sixth annual workshop on Domain-specific aspect languages*, DSAL '11, pages 5–8, New York, NY, USA, 2011. ACM.

[11] Ivica Aracic, Vaidas Gasiūnas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. In Awais Rashid and Mehmet Aksit, editors, *T. Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, 2006.

[12] The AspectJ website. `http://www.eclipse.org/aspectj`.

[13] Ali Assaf and Jacques Noyé. Flexible pointcut implementation: An interpreted approach. In Carré [29], pages 45–60.

[14] Ali Assaf and Jacques Noyé. Dynamic AspectJ. In Johan Brichau, editor, *DLS '08: Proceedings of the 2008 symposium on Dynamic languages*, pages 1–12, New York, NY, USA, 2008. ACM.

[15] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In Mira Mezini and Peri L. Tarr, editors, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005*, pages 87–98. ACM, March 2005.

[16] Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of Static Pointcuts in AspectJ. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007*, pages 11–23. ACM, January 2007.

[17] Henri E. Bal and Dick Grune. *Programming Language Essentials*. Addison-Wesley, 1994.

[18] Ohad Barzilay, Yishai A. Feldman, Shmuel Tyszberowicz, and Amiram Yehudai. Call and execution semantics in AspectJ. In Gary T. Leavens, Curtis Clifton, and Ralf Lämmel, editors, *FOAL 2004 Proceedings - Foundations of Aspect-Oriented Languages - Workshop at AOSD 2004*. Department of Computer Science, Iowa State University, March 2004.

[19] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual machine support for dynamic join points. In Karl Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development ,AOSD 2004*, pages 83–92, Lancaster, UK, March 2004. ACM.

[20] Noury Bouraqadi and Thomas Ledoux. Supporting AOP using reflection. In Filman et al. [47], pages 261–282.

[21] Gilad Bracha and William Cook. Mixin-based inheritance. *SIGPLAN Not.*, 25:303–311, September 1990.

[22] Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for AspectJ. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 209–228, New York, NY, USA, October 2006. ACM.

[23] martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, pages 365–383, New York, NY, USA, October 2004. ACM.

[24] Johan Brichau. Metaspin homepage. `http://www.squeaksource.com/Metaspin`.

[25] Johan Brichau, Michael Haupt, Nicholas Leidenfrost, Awais Rashid, Lodewijk Bergmans, Tom Staijen, Istvan Anis Charfi, Christoph Bockisch, Ivica Aracic, Vaidas Gasiunas, Klaus Ostermann, Lionel Seinturier, Renaud Pawlak, Mario Südholt, Jacques Noyé, Davy Suvee, Maja D'Hondt, Peter Ebraert, Wim Vanderperren, Monica Pinto, Shiu Lun Tsang, Lidia Fuentes, Eddy Truyen, Adriaan Moors, Maarten Bynens, Wouter Joosen, Shmuel Katz, Adrian Coyler, Helen Hawkins, Andy Clement, and Olaf Spinczyk. Survey of aspect-oriented languages and execution models. Deliverable D12, AOSD-Europe, May 2005.

[26] Johan Brichau, Mira Mezini, Jacques Noyé, Wilke Havinga, Lodewijk Bergmans, Vaidas Gasiunas, Christoph Bockisch, Theo D'Hondt, and Johan Fabry. An initial metamodel for aspect-oriented programming languages. Deliverable D39, AOSD-Europe, February 2006.

[27] Richard Cardone and Calvin Lin. Using mixin technology to improve modularity. In Filman et al. [47], pages 219–241.

[28] Denis Caromel, Luis Mateu, and Éric Tanter. Sequential object monitors. In Martin Odersky, editor, *Proceedings of the 18th European Conference in Object-Oriented Programming, ECOOP 2004*, volume 3086 of *Lecture Notes in Computer Science*, pages 316–340. Springer-Verlag, June 2004.

[29] Bernard Carré, editor. *Actes des journées Langages et Modèles à Objets*, Nancy, France, March 2009. Cépaduès-Editions.

[30] Pierre Cointe, editor. *Proceedings of 2nd International Conference on the Meta-Level Architectures and Reflection, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*. Springer, July 1999.

[31] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In Roel Wuyts, editor, *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM.

[32] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[33] Pierre-Charles David, Thomas Ledoux, and Noury M. Bouraqadi-Saâdani. Two-step weaving with reflection using aspectj. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[34] E. W. Dijkstra. On the role of scientific thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1974.

[35] Tom Dinkelader, Mira Mezini, and Christoph Bockisch. The art of the meta-aspect protocol. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development, AOSD 2009*, pages 51–62, Charlottesville, VA, USA, March 2009. ACM.

[36] Simplice Djoko Djoko, Rémi Douence, Pascal Fradet, and Didier Le Botlan. CASB: Common Aspect Semantics Base. Deliverable D54, AOSD-Europe, August 2006.

[37] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don S. Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the Generative Programming and Component Engineering, ACM SIGPLAN/-SIGSOFT Conference, GPCE 2002*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188. Springer, October 2002.

[38] Rémi Douence, Olivier Motelet, and Mario Südholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, REFLECTION 01*, pages 170–186, London, UK, 2001. Springer-Verlag.

[39] Rémi Douence and Mario Südholt. Une technique générique de réification pour les langages à objets. In *6th Maghrebian Conference on Information Technologies, MCSEAI 00*, Fes ,Maroc, November 2000.

[40] Rémi Douence and Mario Südholt. A generic reification technique for object-oriented reflective languages. *Higher Order Symbol. Comput.*, 14(1):7–34, 2001.

[41] Rémi Douence and Mario Südholt. A model and a tool for Event-based Aspect-Oriented Programming (EAOP). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[42] Éric Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, Université de Nantes and Universidad de Chile, 2004.

[43] Johan Fabry, Éric Tanter, and Theo D'Hondt. ReLAx: Implementing KALA over the Reflex AOP kernel. In *Proceedings of the 2nd Workshop on Domain-Specific Aspect Languages (DSAL 2007)*, 2007.

[44] Johan Fabry, Éric Tanter, and Theo D'Hondt. KALA: Kernel aspect language for advanced transactions. *Science of Computer Programming*, 71(3):165–180, May 2008.

[45] Daniel P. Filman, Robert E. Friedman. Aspect-oriented programming is quantification and obliviousness. In Peri Tarr, Lodewijk Bergmans, Martin Griss, and Harold Ossher, editors, *Workshop on Advanced Separation of Concerns at OOPSLA'00*. Department of Computer Science, University of Twente, The Netherlands, October 2000.

[46] Robert E. Filman. Understanding AOP through the study of interpreters, March 2003.

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

[48] Eric Freeman, Elisabeth Freeman, Kathy Sierra, and Bert Bates. *Head First Design Patterns*. O'Reilly, October 2004.

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[50] James Gosling, Bill Joy, Guy Lewis Steele Jr., and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[51] Jan Hannemann and Gregor Kiczales. Design Pattern implementation in Java and AspectJ. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, pages 161–173. ACM, November 2002.

[52] Wilke Havinga. JAMI homepage. `http://jami.sourceforge.net/`.

[53] Wilke Havinga, Lodewijk Bergmans, and Mehmet Aksit. Prototyping and Composing Aspect Languages. In Jan Vitek, editor, *Proceedings of the 22nd European Conference in Object-Oriented Programming, ECOOP 2008*, volume 5142 of *Lecture Notes in Computer Science*, pages 180–206. Springer, 2008.

[54] Wilke Havinga, Lodewijk Bergmans, and Mehmet Aksit. Prototyping and composing aspect languages using an aspect interpreter framework. In *Proceedings of the 3rd Domain-Specific Aspect Languages Workshop (DSAL), AOSD*, April 2008.

[55] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In Karl Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35, Lancaster, UK, March 2004. ACM.

[56] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with contexts. In Ralf Lämmel, Joost Visser, and João Saraiva, editors, *GTTSE*, volume 5235 of *Lecture Notes in Computer Science*, pages 396–407. Springer, 2007.

[57] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proceedings of the 15th European Conference in Object-Oriented Programming, ECOOP 2001*, number 2072 in Lecture Notes in Computer Science, pages 327–353. Springer-Verlag, June 2001.

[58] Gregor Kiczales, Jim d. Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[59] Ivan Kisely. *Aspect-Oriented Programming with AspectJ.* Sams, 2002.

[60] Sergei Kojarski. *Third-Party Composition of AOP Mechanisms.* PhD thesis, Northeastern University, 2008.

[61] Sergei Kojarski, Karl Lieberherr, David H. Lorenz, and Robert Hirschfeld. Aspectual reflection. In *Workshop on Software-engineering Properties of Languages for Aspect Technologies, AOSD 2003*, Boston, Massachusetts, mar 2003.

[62] Sergei Kojarski and David H. Lorenz. Pluggable AOP: designing aspect mechanisms for third-party composition. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA*, pages 247–263. ACM, 2005.

[63] Sergei Kojarski and David H. Lorenz. Modeling aspect mechanisms: a top-down approach. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *Proceeddings of the 28th International Conference on Software Engineering ,ICSE 2006*, pages 212–221. ACM, May 2006.

[64] Sergei Kojarski and David H. Lorenz. AWESOME: an aspect co-weaving system for composing multiple aspect-oriented extensions. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy Lewis Steele, Jr, editors, *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, pages 515–534. ACM, October 2007.

[65] Sergei Kojarski and David H. Lorenz. Identifying feature interactions in multi-language aspect-oriented frameworks. In *Proceedings of the 29th International Conference on Software Engineering ,ICSE 2007*, pages 147–157. IEEE Computer Society, May 2007.

[66] Ramnivas Laddad. *AspectJ IN ACTION.* Manning, 2003.

[67] Ralf Lämmel. A semantical approach to method-call interception. In Gregor Kiczales, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development, AOSD 2002*, pages 41–55. ACM, April 2002.

[68] Cristina Videira Lopes. *D: A Language Framework For Distributed Programming.* PhD thesis, College of Computer Science of Northeastern University, 1997.

[69] David H. Lorenz and Sergei Kojarski. Feature interaction in aspectj 5. In *Workshop on Software-engineering Properties of Languages for Aspect Technologies, AOSD 2006*, Bonn, Germany, March 2006.

[70] David H. Lorenz and Sergei Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *Proceedings of ADI 2007 - Workshop on Aspects, Dependencies, and Interactions at ECOOP 2007*, 2007.

[71] Welf Löwe and Mario Südholt, editors. *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria, March 2006. Springer-Verlag.

[72] Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs.* Wiley, 1999.

[73] Antoine Marot and Roel Wuyts. Composability of aspects. In *Proceedings of the 6th Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT 2008)*, March 2008.

[74] Antoine Marot and Roel Wuyts. A DSL to declare aspect execution order. In *Proceedings of the 3rd Domain-Specific Aspect Languages Workshop (DSAL08)*, 2008.

[75] Antoine Marot and Roel Wuyts. Composing aspects with aspects. In Jean-Marc Jézéquel and Mario Südholt, editors, *AOSD*, pages 157–168. ACM, 2010.

[76] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A fine-grained join point model for more reusable aspects. In Naoki Kobayashi, editor, *Proceedings of 4th Asian Symposium in Programming Languages and Systems, APLAS 2006*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147. Springer, nov 2006.

[77] Hidehiko Masuhara and Gregor Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In Luca Cardelli, editor, *Proceedings of the 17th European Conference in Object-Oriented Programming, ECOOP 2003*, volume 2743 of *Lecture Notes in Computer Science*, pages 2–28. Springer-Verlag, July 2003.

[78] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, *FOAL 2002 Proceedings - Foundations of Aspect-Oriented Languages - Workshop at AOSD 2002*, volume TR#02-06, pages 17–26. Department of Computer Science, Iowa State University, April 2002.

[79] Hidehiko Masuhara, Gregor Kiczales, and Christopher Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. In Görel Hedin, editor, *Proceedings of the 12th International Conference in Compiler Construction, CC 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2003.

[80] Miguel Pessoa Monteiro and João Miguel Fernandes. Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns. In *Proceedings of the Desarrollo de Software Orientado a Aspectos (DSOA'2004) workshop*, November 2004.

[81] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, a Formal Introduction.* Wiley, 1999.

[82] Angel Núñez and Jacques Noyé. A Domain-Specific Language for Coordinating Concurrent Aspects in Java. In *Troisième journée Francophone sur le Développement de Logiciels par Aspects (JFDLPA 2007)*, Toulouse, France, mar 2007.

[83] Angel Núñez, Jacques Noyé, and Vaidas Gasiūnas. Declarative definition of contexts with polymorphic events. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM.

[84] Leonardo Rodríguez, Éric Tanter, and Jacques Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, pages 48–58, Arica, Chile, November 2004. IEEE Computer Society Press.

[85] Ankit Shah and Macneil Shonle. *The XAspects Guide.* Northeastern University's College. `http://www.ccs.neu.edu/research/demeter/XAspects/`.

[86] Macneil Shonle, Karl J. Lieberherr, and Ankit Shah. Xaspects: an extensible system for domain-specific aspect languages. In Ron Crocker and Guy L. Steele Jr., editors, *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003*, pages 28–37. ACM, October 2003.

[87] Éric Tanter. Aspects of composition in the Reflex AOP kernel. In Löwe and Südholt [71], pages 98–113.

[88] Éric Tanter. An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.

[89] Éric Tanter. On dynamically-scoped crosscutting mechanisms. *SIGPLAN Not.*, 42(2):27–33, 2007.

[90] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In Theo D'Hondt, editor, *Proceedings of the 7th International Conference on Aspect-Oriented Software Development, AOSD 2008*, pages 168–179. ACM Press, April 2008.

[91] Éric Tanter, Kris Gybels, Marcus Denker, and Alexandre Bergel. Context-aware aspects. In Löwe and Südholt [71], pages 227–242.

[92] Éric Tanter and Jacques Noyé. A Versatile Kernel for Multi-language AOP. In Robert Glück and Michael R. Lowry, editors, *Proceedings of the 4th International Conference in Generative Programming and Component Engineering, GPCE 2005*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer-Verlag, oct 2005.

[93] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele Jr., editors, *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003*, pages 27–46. ACM, October 2003.

[94] The AspectJ Team. *The AspectJ Programming Guide.* Xerox Corporation and Palo Alto Research Center. `http://www.eclipse.org/aspectj/doc/released/progguide`.

[95] Rodolfo Toledo and Éric Tanter. A lightweight and extensible AspectJ implementation. *Journal of Universal Computer Science*, 14(21):3517–3533, 2008.

[96] Rodolfo Toledo, Éric Tanter, José Piquer, Denis Caromel, and Mario Leyton. Using ReflexD for a Grid solution to the n-queens problem. In *Proceedings of the CoreGRID Integration Workshop*, pages 37–48, Cracow, Poland, October 2006.

[97] Naoyasu Ubayashi, Genki Moriyama, Hidehiko Masuhara, and Tetsuo Tamai. A parameterized interpreter for modeling different AOP mechanisms. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering ,ASE 2005*, pages 194–203. ACM, November 2005.

[98] Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997.

[99] Eelco Visser. Program transformation with stratego/xt: Rules, strategies, tools, and systems in stratego/xt 0.9. In Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, March 2003.

[100] Mitchell Wand. Understanding aspects: extended abstract. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003*, pages 299–300. ACM, August 2003.

[101] Mitchell Wand, Gregor Kiczales, and Chris Dutchyn. A semantics for advice and dynamic join points in aspectoriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.

[102] Martin P. Ward. Language-oriented programming. *Software - Concepts and Tools*, 15(4):147–161, 1994.

A Common Aspect Languages Interpreter

The value of using different (possibly domain-specific) aspect languages to deal with a variety of crosscutting concerns in the development of complex software systems is well recognized. One should be able to use several of these languages together in a single program. However, on the one hand, developing a new Domain-Specific Aspect Language (DSAL) in order to capture all common programming patterns of the domain takes a lot of time, and on the other hand, the designer of a new language should manage the interactions with the other languages when they are used together.

In this thesis, we introduce support for rapid *prototyping* and *composing* aspect languages based on interpreters. We start from a *base* interpreter of a subset of Java and we analyze and present a solution for its modular extension to support AOP based on a common semantics aspect base defined once and for all. The extension, called the *aspect* interpreter, implements a common aspect mechanism and leaves holes to be defined when developing concrete languages. The power of this approach is that the aspect languages are directly implemented from their operational semantics. This is illustrated by implementing a lightweight version of AspectJ. To apply the same approach and the same architecture to full Java without changing its interpreter (JVM), we reuse AspectJ to perform a first step of static weaving, which we complement by a second step of dynamic weaving, implemented through a thin interpretation layer. This can be seen as an interesting example of reconciling interpreters and compilers. We validate our approach by describing prototypes for AspectJ, EAOP, COOL and a couple of other DSALs and demonstrating the openness of our AspectJ implementation with two extensions, one dealing with dynamic scheduling of aspects and another with alternative pointcut semantics. Different aspect languages implemented with our framework can be easily composed. Moreover, we provide support for customizing this composition.
Keywords Aspect-Oriented Programming (AOP), interpreter, semantics, prototyping, composition, Domain-Specific Aspect Language (DSAL)

Un interpréteur extensible pour le prototypage des langages d'aspects

L'intérêt de l'utilisation de différents langages d'aspects pour faire face à une variété de préoccupations transverses dans le développement de systèmes logiciels complexes est reconnu. Il faudrait être capable d'utiliser plusieurs de ces langages dans un seul logiciel donné. Cependant, d'une part la phase de développement d'un nouveau langage dédié capturant tous les patrons de programmation du domaine prend beaucoup de temps et, d'autre part, le concepteur doit gérer les interactions avec les autres langages quand ils sont utilisés simultanément.

Dans cette thèse, nous introduisons un support pour le prototypage rapide et la composition des langages d'aspects, basé sur des interpréteurs. Nous partons d'un interpréteur d'un sous-ensemble de Java en étudiant et en définissant son extension modulaire afin de supporter la programmation par aspects en se basant sur une sémantique d'aspects partagée. Dans l'interpréteur d'aspects, nous avons implémenté des mécanismes communs aux langages d'aspects en laissant des trous à définir pour implémenter des langages d'aspects concrets. La puissance de cette approche est de permettre d'implémenter directement les langages à partir de leur sémantique. L'approche est validée par l'implémentation d'une version légère d'AspectJ.

Pour appliquer la même approche et la même architecture à Java sans modifier son interpréteur (JVM), nous réutilisons AspectJ pour effectuer une première étape de tissage statique, qui est complétée par une deuxième étape de tissage dynamique, implémentée par une mince couche d'interprétation. C'est un exemple montrant l'intérêt qu'il peut y avoir à concilier interprétation et compilation. Des prototypes pour AspectJ, EAOP, COOL et des langages dédiés simples, valident notre approche. Nous montrons le caractère ouvert de notre implémentation d'AspectJ en décrivant deux extensions: la première permet l'ordonnancement dynamique des aspects, la deuxième propose des sémantiques alternatives pour les points de coupe. Les langages d'aspects implémentés avec notre approche peuvent être facilement composés. En outre, cette composition peut être personnalisée.
Mots-clés Programmation Par Aspects (PPA), interpréteur, sémantique, prototypage, composition, langage d'aspects dédié