Vrije Universiteit Brussel – Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes – France
and Universidad de Chile – Chile
2004 - 2005

Universidad de Chile

Aspect Language Modularization, Abstraction and Composition in the Reflex AOP Kernel

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

By: François Nollen

Promoter: Prof. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Dr. José Piquer (Universidad de Chile)

Abstract

Aspect-Oriented Programming (AOP) has emerged as a programming paradigm for better modularization of software. Thanks to AOP, features that crosscut a modular software can be cleanly modularized in aspects. In this line, many languages supporting AOP have been proposed, some of them being general purpose, while others are domain-specific. Versatile kernels based on partial reflection have recently been proposed as a means to foster interoperability between aspects, allowing aspects written in different languages to be composed appropriately. This thesis is concerned with an open architecture for a modular definition of aspect languages, as well as a generic model for aspect composition.

How to read this dissertation

The first part of the thesis presents the necessary background concepts, and introduces existing works in the area of separation of concerns (first chapter), aspect-oriented programming (chapter 2), computational reflection (3), and the prototype this work is based on: Reflex (4). From the issues raised in the introduction, chapter 5 states the motivations for the thesis.

The core of the dissertation is divided in two parts. Chapters 6 and 7 relate to abstraction and modularization issues, whereas 8, 9 and 10 present composition solutions.

Chapters 11 and 12 are dedicated to examples: they illustrate language modularization and aspect composition respectively.

Finally, three chapters compose the conclusion. Chapter 13 is a discussion about related works. Chapter 14 reviews the progression of the dissertation and the results. Lastly, chapter 15 presents the main perspectives for this work.

Contents

Part I

Preliminaries

Chapter 1

Separation of concerns

1.1 The SOC principle

The *separation of concerns* (SOC) is a key principle in engineering. It states that different kinds of concerns in a given problem should be identified and separated, in order to cope with complexity and achieve required engineering quality factors (the *-ities*: understandability, verifiability, maintainability, evolvability or adaptability, reusability and robustness). Said differently, a given problem must be decomposed appropriately in order to be solved.

The SOC principle can be applied in various ways and in different domains: for instance the static, dynamic and systemic approaches to a mechanical problem may be seen as concerns. The decomposition of problems has various names depending on the field of application: AI specialists use to talk about *problem reduction*, whereas the literature refers more generally to the *divide and conquer* strategy from a translation of Julius Caesar's words. In this dissertation we rather use the term *separation of concerns*, which is common for software engineering.

1.2 Traditional modularization

Modularization is the possibility to cleanly encapsulate concerns in separate modules. According to SOC, each concern of a given software design problem should preferably be mapped to exactly one module in the system.

The notion of module depending on the programming paradigm, this thesis focuses on *object-oriented* modularization. Indeed, *object-oriented programming* (OOP) was a major breakthrough in the history of modularization, by promoting *objects* as the encapsulation of both state and behaviour in a modular unit. The organization of objects in this paradigm, typically via inheritance mechanisms, reflects a modularization of concepts and concerns.

1.3 Crosscutting concerns

Despite SOC and object-oriented modularization, some concerns cannot easily be separated: when a concern must be mapped over many modules, this is called *crosscutting*. Conversely, *tangling* refers to multiple concerns applying in a single module. Generally, crosscutting causes tangling.

Typical examples of *crosscutting concerns* are monitoring, logging, synchronization, persistence management, load balancing, etc. They are a serious problem for software designers: on the first hand, crosscutting concerns have their code spread over many places; on the second hand, tangled concerns are bound to pieces of code from other (crosscutting) concerns. As a result, none of these concerns can achieve the -ities, i.e. they are hard to understand, reuse, extend, adapt or maintain.

1.4 Dealing with crosscutting

It may be surprising, but crosscutting concerns do not result from a bad design. Said differently, no refactoring will succeed in removing them. Indeed, they are inherent to the modularization paradigm, rather than the modularization itself.

Crosscutting may be considered as a limitation of OOP. Indeed, object-oriented languages and artefacts are designed from the perspective of a hierarchical decomposition which does not align well with crosscutting concerns. Highlighting some limits of the traditional modularization paradigm, these concerns lead software designers to propose and explore alternative design and programming techniques.

The following of this introduction presents two major approaches for adapting software in order to deal with crosscutting, namely *aspect orientation* and *computational reflection*. Finally, Reflex is presented in chapter 4 as a candidate for aspect-oriented programming using reflection.

Chapter 2

Aspect orientation

2.1 Encapsulating crosscutting

Where conventional mechanisms fail to appropriately cope with the crosscutting problem, *aspect-oriented software development* [AOSD.net] provides explicit abstractions for representing crosscutting concerns, namely *aspects*. In particular, *aspect-oriented programming* (AOP) enables the separation of traditional software components from crosscutting concerns at the programming level, achieving clean modularization.

2.2 AOP concepts and terminology

Several aspect-specific terms will be used in the dissertation. Therefore, this section defines the main concepts of AOP, despite there is still no consensus terminology[1].

Joinpoints. The points of the execution of a program where concerns may crosscut modules. Consequently, crosscutting reflects the presence of joinpoints in various modules for a single concern, whereas tangling can be defined as joinpoints from different concerns in a single module.

Cuts. *Cuts*, *pointcuts* and *pointcut specifications* are three terms referring to predicates over joinpoints. A cut specifies *where* crosscutting happens in a program.

Actions. Also called *advices*, they are semantic extensions applied over an existing program. To make it clear, one can consider them as instructions to be executed at certain points of the program.

Bindings. The *bindings* associate cuts and actions. In practice, they reflect crosscutting concerns.

[1] Most terms however were introduced by Kiczales in 1997.

Aspects. In the context of an *aspect-oriented language* (AOL), an *aspect* is a module encapsulating a crosscutting concern. By extension, aspects are generally considered as the abstraction for crosscutting concerns in AOSD (particularly at the design level as *early aspects*). There is a slight difference however: aspects are not bindings, they rather encapsulate them. Indeed, aspect modules may contain more than one pair cut-action (i.e. binding).

Actually, cuts and actions can be either *behavioural* or *structural*. Behavioural AOP applies to the behaviour or semantics of a base program (typically, executing an instruction when a given method is called). Structural aspects modify the structure of the program (for instance, adding a member to a set of classes). In this work we are concerned with behavioural AOP, as explained later in 4.2.

2.3 AOP technologies

There is a variety of models and proposals around AOP, which differ with respect to their level of support for cuts, actions and bindings, as well as various other considerations: behavioural and structural capabilities, *symmetry* of the paradigm [Harrison *et al.*, 2002], achievement of *obliviousness* [Filman *et al.*, 2000], technological approach (Composition Filters [Bergmans *et al.*, 2001], Aspect Attachment [Kiczales *et al.*, 1997a], Hyperspaces [Tarr *et al.*, 1999]), etc. In particular, some approaches try to achieve genericity, whereas the others are voluntary domain-specific.

Historically, first attempts to separate crosscutting concerns were *domain-specific aspect languages* (DSALs such as [Kiczales *et al.*, 1997b, Irwin *et al.*, 1997, Kiczales *et al.*, 1997c]). These solutions are limited, both in their context of application and their capacity to traduce complex behaviors. Nevertheless, DSALs are close to a particular problem area, which obviously presents benefits: declarative representation, simpler analysis and reasoning about aspects, domain-level error checking and optimizations.

However, development and deployment costs progressively replaced DSALs by generic AOLs. AspectJ [Kiczales *et al.*, 2001] in 1997 was a breakthrough as the first generally applicable solution in AOP. Aspect-oriented logic meta programming (AOLMP [De Volder, 1999]) is another attempt to genericity with the model and languages written in a common ProLog-like language. On the other hand, the Hyperspace proposal [Tarr *et al.*, 1999] stands for one of the most powerful AOP models, but its actual implementation is not satisfying and limits its acceptance. Conversely, Composition Filters [Bergmans *et al.*, 2001] have a very limited model but fairly simple to use and implement, and extensions were proposed still recently [Salinas, 2001].

The various approaches and technologies for AOP lead to tradeoffs: one would choose either a generic or a domain-specific solution depending on the application purpose. In [Tanter & Noyé, 2004], the authors argue that "the most adequate conceptual model and level of genericity for a given application domain actually depends on the situation: *there is no definitive, omnipotent approach that best suits all needs*".

2.4 Combining approaches

Recently, software and language designers started to refocus again on domain-specific solutions, assuming that each aspect is best specified in its own vocabulary. Some propose to explore an alternative approach consisting of combining different solutions, possibly domain-specific. [Rashid, 2001] reports a positive feedback on such a hybrid approach. Considering the advantages of DSALs as well as the need for versatility, several researchers like Wand argue that AOP should concentrate on building tools and environments supporting domain-specific solutions [Wand, 2003]. The combination approach seems promising for AOP, since it should free designers from traditional AOP tradeoffs between genericity and DSAL benefits.

Unfortunately, most aspect-oriented solutions are bound to their implementation, therefore hard to combine. Despite the similarity of their transformations on programs, AOP tools are generally not compatible with each other, or correctness is jeopardized when different technologies interact.

There is still no consensus solution, nevertheless several proposals aim to address the issue of combining AOP approaches. The Concern Manipulation Environment [CME] is a proposal for an ambitious AOSD kernel, dealing with any software representation, including source code as well as UML diagrams. AOLMP [De Volder, 1999] is a solution to building composable aspect-specific languages with logic metaprogramming. XAspects [Shonle *et al.*, 2003] is a plugin mechanism for allowing DSALs to cooperate. Josh [Chiba & Nakagawa, 2004] is an open AspectJ-like language aimed at new aspect language experiments. Finally, Reflex is an open reflective extension of Java [Tanter *et al.*, 2001] which has evolved to implement a model for partial reflection [Tanter *et al.*, 2003] and now turns into a versatile kernel for Java AOP [Tanter & Noyé, 2004].

This work is based on Reflex as a reflective AOP kernel prototype; consequently, this introduction continues with a presentation of the paradigm of reflection, then with Reflex itself. By the way, the last part of the thesis (chapter 13) compares Reflex with the other combination technologies.

Chapter 3

Computational reflection

3.1 Self meta computation

The reflective paradigm results from Brian Smith' studies around the foundations of consciousness and self-reference, as well as his work on the application of these concepts to computer science.

Computational Reflection breaks the traditional distinction between programs and data, by conceiving programs as data for other programs. Programs working on other programs are called *metaprograms*, working at a so-called *metalevel*. The metalevel is opposed to the *base level*, both being *causally connected*. By extension, a whole terminology is built with the 'meta' prefix: *metadata*, *metaprogramming*, *metaobject*, *metaclass*, *metamodel*, etc. The support for *metacomputation* is one of the main properties of reflection. Actually, reflection adds that programs can manipulate not only other programs but themselves; said differently, computational reflection is self metacomputation.

Both experimental and theoretical approaches to computer science met to consider reflection as a promising idea for software engineering. In particular, reflection seems a very appropriate technology for software adaptation. Therefore, it is not surprising that it has been progressively applied to SOC.

3.2 Reflective mechanisms

Computational reflection is self metacomputation, or the ability of a program to both view and modify its own state and behavior. The first mechanism is *introspection*, whereas modifications are called *intercession*. Like for AOP, the support for these mechanisms can be *structural* or *behavioural*, but we focus here on behavioural reflection.

To allow programs to access successively the meta and base levels, reflective systems implement two complementary mechanisms: *reification* shifts programs to data, whereas *absorption*[1] shifts data to programs. These two mechanisms make it possible to stop the execution of a (base level) program, give control to metaobjects accessing the reified program, then go back to the base level and continue the program execution.

3.3 Limits

Despite various approaches, models and implementations, the acceptance of reflection has been limited until now. This is mainly due to the following issues:

- First, reflective computation is expensive, in particular the reification mechanism which consists of shifting programs to data and delegating their interpretation to metaobjects. Full reification impacts seriously on the performance of reflective systems. As a consequence, programmers have to explore alternative solutions, such as *partial reflection* (we will see that Reflex implements such principles).
- Secondly, the adaptability of reflective solutions is often limited by hardwired *metaobject protocols* (MOPs). A MOP specifies the way metaobjects communicate. As a matter of fact, most reflective systems provide a particular infrastructure, designed for a specific purpose; said differently, the designer of the reflective extension commits to particular tradeoffs. As a consequence, the infrastructure cannot be adapted to the needs of a specific reflective application.
- Finally, reflection is generic and powerful, but consequently complex. Working at the metalevel with full visibility and few restrictions on possible modifications requires a good understanding of both reflective mechanisms and the application structure. In practice, computational reflection is reserved to metaprogramming specialists.

In the next chapter, one will be introduced to Reflex and its model for partial reflection, in particular how they provide a solution to these traditional issues of reflection.

[1] Also called *deification* or *reflection*.

Chapter 4

Reflex

This chapter closes the introduction by presenting Reflex, a reflective prototype evolving to an AOP kernel. First is highlighted the strong link between AOP and reflection. Secondly the Reflex model is described through its reflective primitives, their mapping to aspect-oriented concepts and the benefits. Thirdly, an example illustrates the implementation of a basic AspectJ aspect with Reflex. Finally is introduced the concept of a versatile kernel for AOP, as well as the relating requirements, addressed by this thesis.

4.1 AOP and Reflection

AOP as well as reflection are metaprogramming techniques, since they define semantic extensions over a base program. Consequently, it is not surprising to see both solutions being applied to the same issue, namely SOC. In this section, we want to highlight the fact that they also have a strong historical link.

AOP is all the more rooted in reflection that Kiczales, who introduced aspects and AspectJ in 1997 [Kiczales *et al.*, 1997a], strongly contributed in the domain of reflection before promoting aspects. Kiczales qualified AOP of a "principled subset of reflection". Indeed, most aspect techniques have limitations, such as domain specificity, which do not exist in generic reflective systems. By the way, it is probably what makes AOP a better candidate than reflection for wide acceptance, particularly in industrial contexts.

On the other hand, with the technology combination issue, reflection is promoted as an appropriate framework for AOP. First, [Kiczales *et al.*, 1997a] classifies computational reflection with aspectual decomposition paradigms. In [Malenfant & Cointe, 1996], it is argued that reflection is a promising approach to general and extensible aspect weavers. Finally, [Sullivan, 2001] shows how theorical AOP can be implemented with reflection.

4.2 Reflex model and mapping to AOP

Reflex as presented in [Tanter *et al.*, 2003] is the open implementation of an original model for partial reflection. This model is made of three layers, namely from bottom to top level: *hooksets*, *metaobjects* and *links*. These concepts are very similar to cuts, actions and bindings in the AOP terminology:

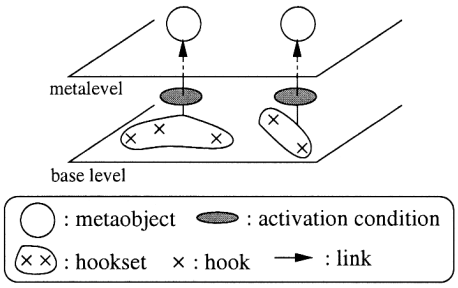

Figure 4.1: Reflex artefacts from [Tanter *et al.*, 2003].

Hooksets. A *hook* is a base-level piece of code responsible for a shift to the metalevel. To express specifications on hooks, Reflex introduces the model of *hooksets*. A hookset has no limitation with respect to the hooks it contains: in particular, it is not necessary to gather hooks on a per-class or per-object basis, contrary to other reflective systems. Secondly, hooksets can be composed at will via standard operators (union, intersection, difference). As a result, they are good abstractions for representing the cut of a crosscutting concern.

Metaobjects. In a reflective model, they act like an interpreter of the base level, possibly changing the way computation is performed. In practice, metaobject bodies implement actions over the base-level program.

Links. Also called *metalinks*, they are the association of a metaobject with a hookset, i.e. a binding in the AOP terminology. Contrary to most reflective systems, Reflex reifies links as first-class entities. They own various explicit and configurable properties, which enable a fine-grained taylorization of the system. The main link-specific properties are the metaobject, the MOP, the scope (object, class or hookset), the *activation condition* and the *control*.
A *control* determines whether the metaobject is given control *before*, *after*, *before and after* an operation occurrence, or if it is allowed to *replace* it. It is equivalent to the semantics of `before`, `after` and `before-after` advices in AspectJ, the *replace* control corresponding to an AspectJ `before-after` without any call to `proceed()`.

Lastly, *activation conditions* are dynamically-evaluated conditions controlling the actual application of a link. Comparing with AspectJ, there may be various ways to implement this kind of conditions (booleans, cflow conditions, etc.), but no dedicated primitive.

Hooksets as well as open-MOP links enable specifications over *what* will be reified, that is why Reflex is said to implement *spatial selection of reification*. Similarly, activation conditions limit *when* shifts to metaobjects occur, which reflects *temporal selection*. As a result, Reflex users can choose an appropriate level of power (or performance) for a specific application. Thanks to its model for partial reflection, Reflex addresses the traditional reflection issues seen in 3.3.

Finally, Reflex cuts, actions, and links can be either behavioural or structural. In order not to modify the standard Java execution environment, Reflex behavioural links (`BLink`) are *set up* at load time, i.e. hooks and necessary infrastructure elements are installed in base code according to the hookset, whereas structural links (`SLink`) are fully *applied* at load time. Reflex structural mechanisms being simpler, this thesis focuses on behavioural applications, and presents a model for behavioural link composition. Structural composition will be implemented as a future work, as a simplification of behavioural mechanisms.

A direct mapping is possible between Reflex primitives and AOP concepts. The following table shows the bridges between the concepts seen previously in 2.2, the AspectJ and Reflex terminologies:

Concept	AspectJ term	Reflex term
joinpoint	joinpoint	hook
cut	pointcut	hookset
action	advice body	metaobject body
binding	aspect	link

Figure 4.2: AOP concepts mapped to AspectJ and Reflex.

In [Tanter & Noyé, 2004], Reflex authors review the main features of AOP and explain how "an appropriate reflective model [...] is generic enough to handle a wide range of [...] aspect-oriented approaches". After presenting the back-end reflective model, they show how Reflex is able to apply aspects from various AOLs. Aspects are translated in terms of reflective primitives, in particular Reflex links.

4.3 An AspectJ-Reflex example

Since Reflex and AOP concepts align well with each other, it is not surprising to imagine an AspectJ aspect being written similarly with Reflex artefacts. For an illustration, we consider an aspect in charge of logging the argument of a given method each time it is called. To make it clear, let `Test.foo(String)` be the method and `System.out.println("foo executes with argument "+arg);` the behaviour to implement whenever the method is called.

Typically in AspectJ, one defines a pointcut to match the call to `foo`. The argument is passed to, for instance a `before` action, which implements the logging process.

```
public aspect LogAspect {

    pointcut callToFoo(String arg):
        call(Test.foo(String);
        && args(arg);

    before(String arg): callToFoo(arg) {
        System.out.println("foo executes with argument "+arg);
    }
}
```

In Reflex, a hookset is associated to the reception of the `foo` message. A metaobject implements the action in a standard method body. Thirdly, a link is created to bind the hookset and the metaobject. The link control is set to `BEFORE`, while the activation condition is left to (always) `TRUE`. Finally, the MOP is configured so that only the required argument is passed to the metaobject, since other pieces of information are useless for this specific application.

```
public class LogConfig {

    public static void initReflex() {

         PrimitiveHookset theHS = new PrimitiveHookset(
             MsgReceive.class,
             new NameClassSelector("Test"),
             new NameOperationSelector("foo"));

         MODefinition theMO = new MODefinition.MOClass(
             LogMetaobject.class.getName());

         BLink theLink = API.links().addBLink(theHS, theMO);
         theLink.setControl(Control.BEFORE);
         theLink.setActivation(Activation.TRUE);
```

```
            theLink.setMOCall(new CallDescriptor(
                LogMetaobject.class.getName(),
                "performLogging",
                Parameter.FIRST_ARGUMENT));
        }
    }

    public class LogMetaobject implements BMetaobject {

         public void performLogging(String arg) {
             System.out.println("foo executes with argument "+arg);}
    }
```

In the Reflex piece of code, the selector classes `NameClassSelector` and `NameOperationSelector`, the control `BEFORE`, the condition `TRUE` as well as the parameter `FIRST_ARGUMENT` are predefined. Nevertheless it is possible to replace them or implement them from scratch for a specific application.

Not all code details are explained here, but the example highlights most of the benefits of Reflex over AspectJ. Even if it takes more lines of code to specify the aspect with Reflex, at least the concepts and code fragments composing the metalevel application are well separated, and each concept has its own class (for standard reusable objects). Conversely, the AspectJ file gathers the whole metalevel specification, and individual elements are difficult to reuse since they are not objects. In a nutshell, Reflex shows benefits in terms of expressiveness (thanks to explicit artefacts such as the activation condition), reusability (promoting types and objects rather than syntax) and dynamicity (artefacts and properties being accessible at runtime[1]).

4.4 To a versatile kernel for AOP

Since most AOP approaches rely upon common implementation mechanisms, Reflex authors proposed to make the reflective tool evolve to a *versatile AOP kernel*: thanks to the reflective primitives and their mapping to AOP, it would support core semantics, allowing to use and compose aspects from different aspect technologies. By the way, such a kernel should provide expressive means for applying and composing aspects.

In [Tanter & Noyé, 2004], Reflex authors promote a language layer over the reflective kernel and abstractions (hooksets, links, etc.). Indeed, Reflex uses static configuration classes[2] to create and set up reflective primitives, but

[1] Keeping in mind that the current Reflex implementation cannot modify hooks in a class once this class was loaded.

[2] LogConfig seen in 4.3 is an example of a static configuration class.

no mechanism automates the translation of aspects into Reflex artefacts: users are responsible for translating aspect semantics. As a consequence, the syntatic support intrinsic to aspect-specific languages is lost. A language layer composed of AOL-specific translators would be a guidance for programmers, who would not have to know the details of the reflective model to make the kernel apply their aspects.

Secondly, the resolution of aspect interactions cannot be decided automatically (without taking into account the application semantics), according to [Tanter & Noyé, 2004]. Nonetheless, their detection can be automatic: there is an interaction whenever "several inserts are executed at the same point"; said differently, there is no conflict if "at most one aspect crosscuts at each join point" [Douence *et al.*]. Since aspects apply through links, Reflex detects aspect interactions as two or more links applying at the same joinpoint (i.e. hook). On top of detection mechanisms, composition features could be implemented in order to let users specify the resolution of interactions. Composition should be supported in a flexible manner, so that complex interactions (possibly overpassing the AOL composition capabilities) are handled by the AOP kernel (Figure 4.3). By the way, the composition models and implementations presented in this thesis will allow runtime specification, even if for the moment Reflex cannot apply new composition specifications after classes are loaded (i.e. links are set up as seen in 4.2).

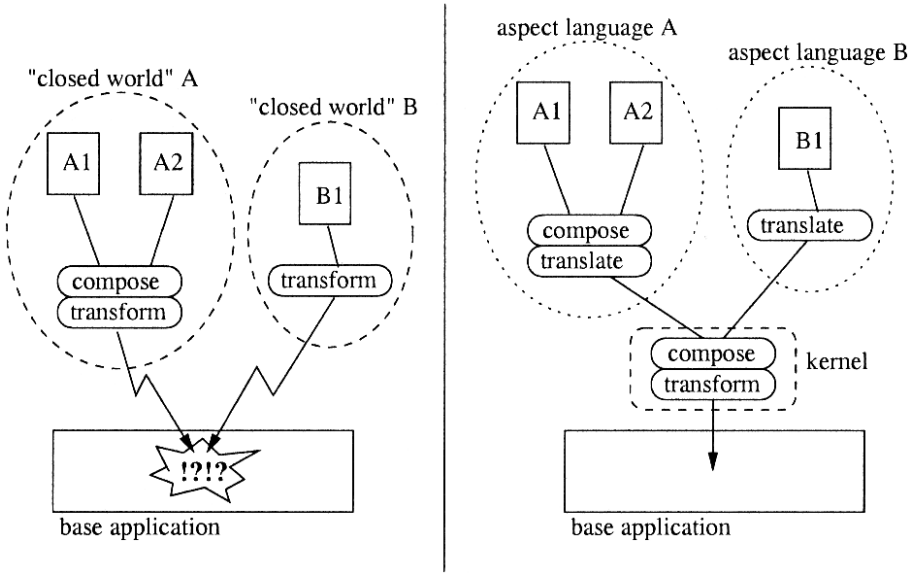

Figure 4.3: Aspects in different languages cooperate, being both composed and applied by the AOP kernel. (illustration from [Tanter & Noyé, 2004])

Chapter 5

Motivation

The context of this thesis is the transition of Reflex from a partial reflection tool to a versatile kernel for AOP.

The first motivation lies in a higher user level of abstraction. According to [Tanter & Noyé, 2004], Reflex can apply aspects from different languages; however users are responsible for translating them into reflective primitives. First, we want kernel users to manipulate aspects directly. Since Reflex links are bindings (seen in 4.2) and bindings are not exactly aspects (2.2), the thesis starts with the introduction of an appropriate abstraction for representing and reasoning about aspects (chapter 6). Secondly, a good AOP kernel should provide an open-language layer over low-level primitives (4.4), since the family of AOLs remains open-ended; besides we think a clean modularization of language translators is also a requirement for a good kernel. As a result, chapter 7 presents an open-language plugin architecture, implementing the language layer introduced in 4.4.

On the other hand, [Tanter & Noyé, 2004] states an AOP kernel should support aspect composition, without being limited to the semantics from a given aspect language. Consequently, the second part of the thesis is concerned with the definition of a generic composition model, an appropriate framework and its implementation in the Reflex AOP kernel. Chapter 8 presents a first solution based on link composition. Chapter 9 shows the limits of this approach through an example. At last, chapter 10 refines the model and presents new composition mechanisms.

Chapters 11 and 12 are dedicated respectively to modularization and composition examples. Chapters 13 to 15 make the conclusion, comparing the solution built from 6 to 10 with existing works, reviewing its incremental design and presenting perspectives.

Part II

Abstraction and Modularization

Chapter 6

An abstraction for aspects

In this chapter, a new abstraction is introduced to represent aspects. An appropriate artefact allows users to reify and compose aspects explicitly, instead of low-level kernel primitives.

6.1 Introduction

As seen in part 4.2, Reflex links reify the concept of bindings. However bindings and aspects are different notions: a binding is the association of a cut and an action, whereas an aspect is a programming artefact encapsulating one or more bindings. Therefore, it cannot be expected that aspects are always reified by only one link. As an illustration, an AspectJ cflow-based pointcut results in two Reflex links: one for the cflow itself, another one for the pointcut using the cflow. Actually, aspects correspond to a higher level of abstraction than links: aspects contain links.

Another layer above links is the language layer introduced in 4.4, which will be implemented by the plugin architecture in chapter 7: language translators form another level of abstraction. However, it is obvious that there is not a one-one relation between aspects and translators, therefore plugins no more than links are appropriate to reason about aspects.

One will conclude there is a lack for an appropriate abstraction to represent and manipulate aspects in Reflex.

6.2 Model: the linkset artefact

We introduce *linksets* as a mean to group links that are part of the same higher-level conceptual or semantic unit. Similarly to hooksets gathering hooks, there is no explicit restriction on the set of links contained by a linkset: it is only assumed that they have some concept, source plugin,

purpose, etc. in common. This definition makes linksets appropriate abstractions to gather links from a given aspect, therefore abstractions for representing aspects.

One should notice that the new linkset artefact is not limited to aspects. If for instance aspect translators need to declare *infrastructure links* in addition to aspect bindings resulting from direct aspect translations, they can use specific linksets to gather them. Said differently, these infrastructure linksets will contain links which do not result directly from a user aspect, but share a common purpose reified by their linkset.

Finally, we think that linksets are more appropriate for aspects and plugin infrastructure links than *user links*. User links refer to links built and declared individually in static configuration classes[1]: they are created and configured explicitly by the user and are not supposed to require higher abstraction. Consequently, we choose to bind the concept of linkset to aspect translators: user links remain individual, whereas translators have to declare their links inside a linkset. This mechanism will be seen in paragraph 7.2.2, its influence on composition in part III.

6.3 Implementation

The `Linkset` class consists of a mere link container with a reference to the parent aspect translator. It is checked by construction that each link added to the linkset is a plugin link (opposed to a user link), owned by the same plugin as the linkset.

In order to facilitate the manipulation of plugin links and linksets, particularly in algorithms, they implement the well-known *composite* design pattern. This pattern allows to treat single components and collections of components exactly the same. In practice, clients manipulate instances of an abstract type, `PluginLink`, ensuring that components, primitive or composite, implement the same protocol. Figure 6.1 shows linksets as composite components and *nested links* (links inside a linkset) as primitives.

A consequence of implementing the composite design pattern is that linksets may contain linksets. This capability could be used to explore the perspective of *aspects of aspects*, but also to create conceptual hierarchies of linksets, not necessarily representing aspect hierarchies. A specific study should assess the precise meaning, benefits and drawbacks of linkset hierarchies.

[1] Such as the link configured for the logging example in 4.3.

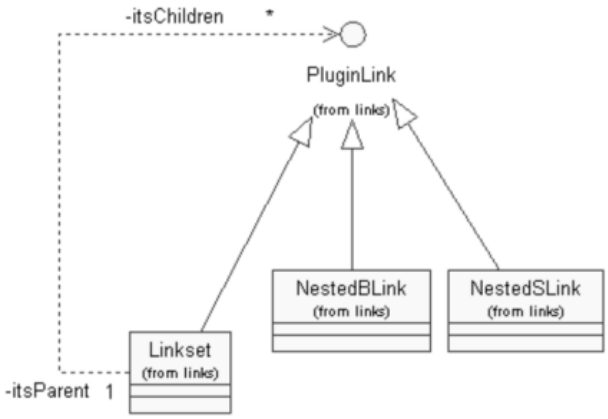

Figure 6.1: The composite pattern applied to plugin links.

6.4 Conclusion

The linkset definition introduced in this chapter provides an appropriate abstraction to group and manipulate links with respect to aspects. By the way, our implementation allows linksets of linksets, which should be studied as a future work. Concerning Reflex frameworks for plugins and composition, the influence of linksets will be seen in the next chapters, which present those frameworks in detail.

Chapter 7

Language layer & aspect translators

This chapter presents a plugin architecture for AOL translators on top of the kernel reflective model. The aim is to increase the levels of abstraction and automatization provided by the AOP kernel.

7.1 Introduction

7.1.1 A layer over reflective primitives

The Reflex AOP kernel is able to apply aspects from different aspect-specific languages using its own primitives, in particular the link abstraction for bindings (4.2), reified by `Link` instances. [Tanter *et al.*, 2003] shows how one can define and manipulate links and other kernel-level primitives inside static configuration classes, or dynamically using dedicated kernel APIs. Nevertheless, one would like to use and compose aspects from different languages without having to translate them manually into kernel primitives. Contrary to Reflex developers, standard kernel users should not necessarily be aware of the internal reflective model, assumed they should rather concentrate on writing aspects and composing them. This chapter addresses the issue translating aspects automatically, in order to increase the user level of abstraction.

7.1.2 Modularizing translators

Secondly, the language layer must not be limited to a closed set of AOLs: it should rather be opened to (even not aspect-specific) future languages. To open the architecture, we decided to encapsulate language-specific translators into explicit reusable entities called *plugins*.

Plugins in general are components designed to fit a specific infrastructure.

As an illustration, three (Java) component models are briefly presented here, to highlight the main characteristics expected from a plugin:

Eclipse plugins [Eclipse.org]. The *Eclipse* platform is probably the most famous example of plugin architecture [Eclipse]. Eclipse plugins are composed of three elements: a *manifest*, *JAR archives*[1] and additional ressources. In practice, implementing classes are gathered in the JAR archives, whereas the manifest describes the plugin in XML. This description includes an identificator, references to external libraries or components, as well as a protocol to manipulate the plugin (creation, deployment, activation): actually, the manifest is responsible for the integration of the plugin with the platform.

Enterprise JavaBeans [EJB]. The *EJB specification* from J2EE defines a business component architecture for multi-tiers client-server systems. In practice, EJB components are made of a *remote interface*, a *home interface* and implementing classes in Java. The remote protocol reflects the services offered by the component, whereas the home interface is used to create or find the component: both participate to integrate the component with the other components and the so-called *EJB container*.

Reflex argument handlers [Tanter & Noyé, 2004]. Reflex *handlers* apply configurations on the kernel through startup arguments. These components implement the `ArgumentHandler` interface, which ensures their integration with Reflex. The protocol forces handlers to declare a set of *tags* to the kernel, which consequently distributes Reflex startup arguments depending on their tag. To integrate a new handler with the kernel, one encapsulates it in a JAR and places it in the dedicated Reflex folder.

Eclipse, EJBs and Reflex argument handlers are very different technologies, however their component models share commonalities: components are made of implementing classes on the first hand, manifests or interfaces to integrate with their environment on the second hand.

This section presents a modular AOL plugin architecture on top of Reflex, then an appropriate framework and its implementation.

[1] A JAR is a Java code archive.

7.2 Model

7.2.1 Reflex plugins

Plugins encapsulate AOL translators and compose the language layer above Reflex primitives.

In practice they are designed as an extension of argument handlers: encapsulated in a plugin JAR and deployed automatically, they declare their tags to the kernel, which forwards them appropriate arguments at load time. Alternatively, they can be called at runtime via their public methods. Finally, plugins use a specific *plugin API* to configure the kernel. The details of this API are presented in the next sections.

7.2.2 Nested links

Nested links are primitive links inside a linkset (6.2), which is equivalent to state they were declared by a plugin. As seen in 6.3, a plugin link is either a nested links or a linkset.

In practice, nested links embed a reference to their owner (i.e. the parent plugin). It enables plugin link traceability, therefore better user feedback quality, particularly in case of link conflict. As an illustration, appendix A shows how error and warning messages from the kernel explicitly refer to plugin names and descriptions instead of bare link identifiers.

7.3 Frameworks and implementation

7.3.1 Kernel configuration

Like standard handlers, plugins have to implement `handleArgs(args)`, which is used by the kernel to dispatch startup arguments to handlers and plugins. Typically, `handleArgs` encapsulates the business part of the plugin.

To configure the kernel with respect to aspect semantics, plugins use the specific `PluginAPI` interface. Configurations relating to operations, attributes and composition are forwarded to the standard Reflex API, whereas the link part of the protocol, namely `PluginLinkAPI`, ensures mechanisms specific to plugins:

- First, `PluginLinkAPI` forces plugins to declare linksets rather than links. Secondly, plugins cannot instantiate links themselves: they have to obtain instances from the kernel, configure them, then send them back to Reflex. Actually, plugins are always returned nested links (but it is transparent from a plugin point of view thanks to the downcast):

as a result, link implementation details are hidden from plugins, besides the binding of nested links with their parent plugin is performed automatically by the kernel.

- Secondly, plugins can only manipulate their own links. The API prevents them from accessing external links. Similarly, one will see in chapter 8 that plugins cannot express composition rules involving other links than their own ones.

Figure 7.1 illustrates how plugin instructions to the kernel are filtered at the level of the plugin API: some configurations are forwarded directly to the standard APIs (for Reflex handlers and configuration classes), whereas the link part of the protocol is implemented specifically for plugins:

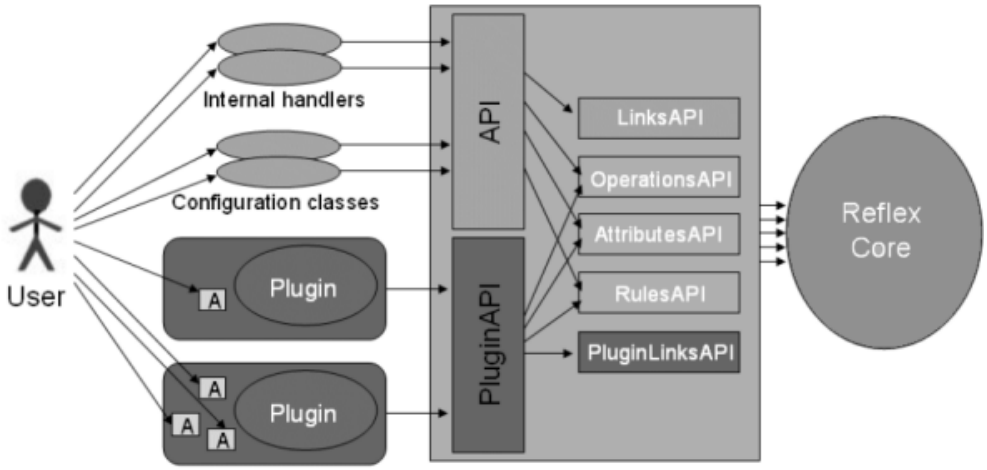

Figure 7.1: Reflex standard and plugin APIs.

7.3.2 Plugin registration

From an implementor point of view, a plugin is very similar to a standard Reflex handler. Indeed, they are encapsulated in a JAR, receive Reflex startup arguments, perform configurations on the kernel... In particular, plugins like handlers must *register* at startup.

The registration phase consists in providing information to the kernel in order to be referenced and activated. The kernel looks for the name of the plugin, its textual description, its specific tags and their description. Tags precede arguments in the Reflex startup line; they can start with "-" (standard tags) or "--" (getting priority over standard tags).

Actually, plugins register by implementing a set of *getters*[2] for each piece of information: `getSupportedTags`, `getTagsExplanation` and `handleArgs` from `ArgumentHandler`, as well as `getName` and `getDescription` from the `Describable` interface also implemented by plugins. These methods are used at startup for registration and argument dispatch, as well as user feedback at runtime (displaying activated plugins, their description and tags, in case of conflict for instance).

7.3.3 Smart feedback

Smart user feedback consists of displaying plugin and aspect references inside error and warning messages, rather than bare links identifiers. Appendix A illustrates this concept with examples.

Traceability being ensured from links to linksets and linksets to plugins (from parent references in nested links and linksets respectively), the kernel can use the plugin getters again to display additional information in case of link conflict, in particular aspect-level descriptions of interactions. By the way, plugins implement a protocol called `RuleOwner` used to get the *source* of a composition rule[3]. The source of a rule, which is generally an aspect, is another piece of information used to improve the quality of user feedback.

To sum up, the plugin architecture ensures a full mapping from bottom to top between the different layers of abstraction in the kernel: links and rules (primitives), linksets (composites) and plugins (translators).

7.4 Conclusion

This section has presented a modular architecture for language translators based on plugins and plugin links. Thanks to this language layer, it is possible to design aspect translators and configure the kernel automatically with respect to aspects. Moreover, traceability and user-friendly feedback are achieved by the framework, in particular the automatic binding of reflective primitives with their aspect and translator.

Chapter 11 shows how to implement and activate plugins through examples.

[2] A getter is a method returning an attribute of the object.

[3] Indeed, one will see in chapter 8 that plugins may declare composition specifications.

Part III

Composition

Chapter 8

Composing links

This chapter presents a first solution to the issue of composing aspects, which is built on top of link composition mechanisms. First, the composition model and primitives are defined, then a dedicated framework and its implementation.

In next chapters, this proposal will be refined to support an explicit consideration of link *controls* defined in 4.2.

8.1 Introduction

As seen in the preliminaries, Reflex is based on a model for partial reflection caracterized by reified metalinks. Links are low-level artefacts used as primitives by the reflective kernel to translate and apply aspects, whatever the aspect language. Like aspects associate cuts and actions in AOP, links reify bindings between hooksets and metaobjects. Aspect interactions reflect the application of multiple links at the same point of the base program. Therefore, the issue of composing aspects is tightly bound to the composition of resulting links.

This chapter presents the design and implementation of a generic model for composing aspects via links. First, a new primitive is introduced to support composition specification. Secondly, a framework is defined to rule how kernel users access and use composition features. At last, we present our implementation, in particular how it supports additional user-defined composition operators.

8.2 Model

8.2.1 Four-step composition process

The Reflex kernel provides link primitives and detects their interactions at runtime. When an interaction occurs, the kernel should be able to resolve the application of these links using composition *rules* (specifications) declared by users or translators.

To reason about composition, we consider a four-step process: *Specification* (of composition rules), *Detection* (of interactions), *Resolution* (of interactions) and *Application* (of the solution), or *SDRA*. Contrary to the other steps of the process being performed automatically (by the kernel), the specification phase directly involves users and plugins. It starts at loadtime and finishes when an interaction is detected: during this phase, rules can be specified dynamically. After the resolution of an interaction, the cycle restarts, and it is allowed again to declare rules until a new interaction is detected.

Clearly, the SDRA model we introduce is supposed to be a convenient scheme for reasoning about interactions and presenting how resolution works, more than a constraining framework.

8.2.2 An artefact for composition rules

To enable composition, it is necessary to introduce a representation for composition rules.

Our proposal is based on *abstract syntax trees* (ASTs) for composition rules. These data structures are often used in compiler and interpreter representations of a program, optimized for representing parsed information and generating code from this information. To reify link composition rules, we make ASTs from *composition operator nodes* and *link leaves* (including user links, nested links and linksets), as shows Figure 8.1.

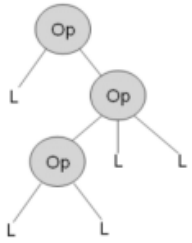

Figure 8.1: AST rules made of composition operators and links.

Assuming links are the lowest-level artefacts composed by the kernel, added a system opened to new operators (seen further in 8.4.3), AST composition rules ensure maximum expressivity for composing aspects.

8.3 Framework

After composition primitives, an appropriate framework must be defined to specify their usage and result in concrete composition features for the AOP kernel. This section describes a *composition rule patterns* system and presents each composition phase in details.

8.3.1 Rule patterns

To ensure plugins cooperate without cancelling each other, they manipulate only their own primitives. As seen in 7.3.1, plugins cannot access external links. Similarly, the composition framework guarantees that plugins express rules involving no other link than their own.

To achieve this requirement, we introduce *rule patterns*. Depending on the type of links involved in a composition rule, as well as the source of these links, different patterns are distinguished. The following table shows who is allowed to declare each type of rule:

	User	Plugin
Links from a unique linkset	-	Ok
Linksets from a unique plugin	Ok	Ok
Linksets from different plugins	Ok	-
User links and linksets	Ok	-
User links	Ok	-

Figure 8.2: User and plugin rule patterns.

According to this grid, the kernel validates newly declared rule: if no pattern is matched or if it is not allowed for the rule owner, an error is raised (see appendix A for examples); otherwise the rule is stored in a repository[1].

One should notice that our current rule pattern set is not appropriate for linksets of linksets. In 6.3 we suggested to study the influence of linkset hierarchies; if such a study is carried out and linksets of linksets used, the rule pattern set should be updated consequently.

[1]Except intra-linkset rules, which become an attribute of the linkset, achieving better object orientation.

8.3.2 Composition phases

Specification phase

Even if the current implementation of Reflex detects link interactions and applies compositions when loading classes, we introduce a public API dedicated to dynamic specification. These composition services are implemented by a new dedicated kernel module[2].

Assumed composition specifications should be overridable during the execution, we allow multiple rules to be declared at runtime. Another choice is to allow partial speicifications: indeed, involving all links for each new rule seems far too constraining from a user point of view and forces to systematically over-specify composition, added links can be added at runtime (what about previous specifications then?).

The drawback with multiple partial rules is the difficulty to merge them when necessary (to solve a particular interaction). To illustrate this, let's introduce the `Seq` operator, commonly used in AOP to express precedence. Assuming two link rules `Seq(L1,L2)` and `Seq(L2,L3)` were declared, if L1 and L3 interact, composing them correctly requires a merging mechanism. Indeed, the solution or `Seq(L1,L2,L3)` (optimally `Seq(L1,L3)`) can be deducted from the rules, since precedence is transitive. If rules are not merged, the kernel cannot compose L1 and L3.
Merging rules is a very complex issue, particularly in an open-operator context. So are consistency checking mechanisms between specifications. We consider that a specific study is required for bringing such features into this Reflex composition model: typically, a logical programming approach could be adopted. Our solution does not provide rule merging or consistency checking features. Consequently, the composition module mentionned in first paragraph is actually very similar to a mere rule repository, where composition specifications are stored and accessed at runtime.

Nevertheless, even if the global consistency of declared rules is not verified, each one is inspected[3] to ensure several of their properties: a rule must never be cyclic or empty, and its pattern must be allowed for its owner (as seen in 8.3.1).

Detection phase

The detection of link interactions is already implemented in Reflex. When an aspect interaction occurs, the list of interacting links is now passed to an *interaction solver*, which is in charge of returning the composition to apply.

[2]One will see that this module consists more or less in a bare rule repository.
[3]The correct term is *visited*, as seen later in 8.4.1.

Resolution phase

Solving a link interaction means here to express and return an appropriate composition specification, which can be interpreted and applied. To express the solution of an interaction, a standard AST rule is used (as in 8.2). As a result, the solver is a module which takes a list of interacting links as input and returns an AST composition rule.

If no or not enough rules specify composition to solve an interaction, the solver returns a warning message and performs default composition: typically, all links are applied successively (as it will be seen in the example 12.3).

As explained previously, our first implementation has no rule merging capability. Therefore, the role of the solver consists of selecting and returning *the most appropriate rule for a given interaction*, which we temporarily define as the smallest[4] specification containing[5] all interacting links. An algorithm extracting this rule for a given interaction is presented in the implementation section (8.4.2).

By the way, we think that the solver should only return strictly necessary links and branches from a rule, in order to simplify further processing. Back to the previous example, we expect the solution rule `Seq(L1,L3)` rather than `Seq(L1,L2,L3)`, i.e. a version of the merged rule minimized for the current interaction. Not surprisingly, the semantics of *rule minimization* depends on both operators and links. For instance, minimizing a `Seq`-based rule consists of removing not interacting links. Conversely, if `If(L1, L2)` is a conditionnal rule[6], minimizing it when L1 does not apply will not result in `If(L2)` (which has no sense a priori). To illustrate correct rule minimization, an example involving a `Seq` and a `If` is presented in Figure 8.3. To sum up, the minimization of a rule depends on the operators semantics as well as the interacting links. In 8.4.3 one will see how to implement minimization mechanisms in the context of an open-operator support.

Application phase

Once a solution has been returned for a given interaction, it must be concretely applied by the kernel. Nonetheless, we are not concerned with this part of the process, since it is part of a specific engineering work at the University of Chile.

[4] In terms of numbers of links involved.
[5] Considering an AST 'contains' its leaves.
[6] Understand *L2 applies only if L1 applies* for instance.

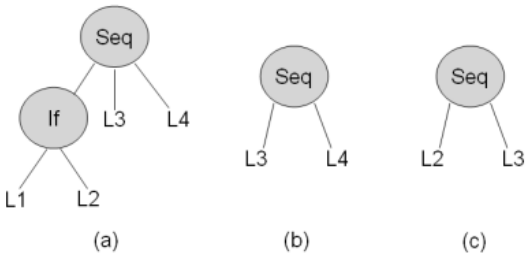

Figure 8.3: Rule minimization is applied from bottom to top, with respect to interacting links as well as operator semantics. The figure shows the result of the minimization of (a), first if L2, L3 and L4 interact (b), or if the interaction involves L1, L2 and L3 (c).

8.4 Implementation

This section first presents the implementation of composition primitives (AST rules), as well as details about the composition phases (specification, detection, resolution and application). At last, it focuses on the open operator support issue.

8.4.1 Rules

AST rules are instances of `Rule`. This class implements an interface called `Visitable`, in order to fit the *visitor* design pattern. The benefit is a clean encapsulation of rule browsing mechanisms.

The first property of a rule is its owner, for traceability. It is set automatically at declaration time (rather than creation time), consequently the owner is not the object who instantiated the rule but the one who added it to the kernel. Besides, a rule owns either a set of children rules (which makes the rule a node associated to an operator in the AST model) or a link (which makes the rule a leaf): indeed, another implementation of the composite design pattern allows to treat rules and link leaves similarly in composition algorithms.

By the way, several rule-specific capabilities, like returning a user-friendly string representation, or the list of wrapped links[7], are implemented using rule-specific visitors in anonymous inner classes.

[7]Direct containees or links from nested rules

8.4.2 Phases

Specification phase

As explained previously, composition rules can be declared at runtime. For this purpose a specific method is introduced in the kernel public API, namely `addRule(rule,owner)`, the owner being of the dedicated type `RuleOwner`.

In the current implementation, rule owners are merely responsible for returning rule sources as strings. When an error occurs with a rule (typically while resolving a link interaction), the kernel gets its source for better user feedback: for the moment it may be a plugin, a user, an aspect filename, actually anything meaningful.

When a composition rule is added to the kernel, a specific initialization process is triggered, consisting of matching a correct rule pattern (8.3.1), setting the owner (a kind of late binding), and verifying several rule properties (the presence of at least one leaf, the non-cyclic nature of the tree, etc.). The implementation of these mechanisms involves specific visitors.

Detection phase

As seen previously, the thesis is not concerned with the detection part, which is already implemented in Reflex.

Resolution phase

Resolving a link interaction consists of selecting appropriate composition rules and returning a minimized version of the most appropriate. To achieve this, a simple algorithm is proposed:

1. First, composition rules are selected from the repository. Rules are stored by pattern, which improves the selection mechanism for a given interaction. Each rule is observed with respect to interacting links[8]: only rules containing all links are kept; if no rule is appropriate, a warning message is displayed and default composition is applied.

2. From remaining rules, only those with a minimal number of links are kept. From what remains after this new selection, the first rule is arbitrarily kept.

3. Lastly, the selected rule is minimized and returned. Minimization consists of removing unnecessary branches and leaves, with respect to operators and interacting links. 8.4.3 will expose how operators implement minimization in practice.

[8]The input of the algorithm

Actually, it would be sufficient if composition rules were only made of primitive links. Since rules can include linksets, a dedicated mechanism is applied over the original algorithm to enable shifts between both levels of abstraction. In practice, linkset-level interactions are processed before lower-level interactions between links. The following algorithm ensures the resolution of link interactions in the context of linkset-level rules:

1. First, nested links are replaced by their linkset, so that the input of the previous algorithm is a list of user links and linksets.

2. Secondly, the previous algorithm is applied, returning a rule invloving user links and linksets[9].

3. Finally, the rule returned is *expanded*, which means that linksets are replaced by their interacting links: if there is only one interacting link from a linkset, the linkset is replaced by this link inside the global solution; if more than one link from this linkset interact, then the linkset is replaced by the most appropriate from its intra-linkset rules (indeed, the selection algorithm is applied recursively with the linkset links and the intra-linkset rules as inputs).

As a result, a minimized link-level rule is returned in any case of interaction, sometimes directly selected from the repository, sometimes built on purpose from existing rules.

Application phase

As said previously, this part of the process is currently the subject of a parallel work at the University of Chile, which is why it is not described here.

8.4.3 Open operator support

Assumed we are not concernet with the application phase, only minimization mechanisms are operator-dependent in the composition process. For this reason, operators have to embed their own minimization strategy.

In practice, operators are forced to implement the `minimize(links)` method, which takes interacting links as input, and returns a minimized version of the rule. To minimize it, the `minimize` method is called on the highest-level operator[10], which consequently minimizes its children rules, may move or remove some of these branches, depending on interacting links and the operator own semantics.

[9] Thanks to the composite pattern, manipulating both links and linksets inside the algorithm is very straightforward.

[10] I.e. the rule itself.

8.5 Conclusion

In this chapter we have built a complete solution (model, framework and implementation) for composing links in Reflex.

First, we decided to reify composition specifications as ASTs made of operators and links. We chose to allow multiple partial specifications, without implementing complex inter-rule merging or checking mechanisms. These features could however inspire future works.
To reason about composition, we proposed a four-step process: specification, detection, resolution, application. In practice we described completely two of them, detection mechanisms being already implemented and application being part of a specific study.
Finally, we proposed an algorithm to select the most appropriate rule in the context of a given link interaction, supporting both link and linkset levels of abstraction inside rules.

However, our assumption that links are the lowest-level entities to manipulate for composing aspects in Reflex will be corrected in chapter 9. Indeed, one will realize the benefits of considering link controls too in the composition model.

Chapter 9

To an explicit consideration of controls

This chapter highlights the benefits of reasoning about *controls* (4.2) as well as links. Several new concepts are introduced; in chapter 10 it will result in new composition primitives, refining the solution built in 8.

9.1 Introduction

As seen in 4.2, links can apply at different positions of a given point in the base program, similarly to aspects themselves. In a language such as AspectJ, the position is made explicit by using a `before`, `after` or `before-after`[1] advice. In Reflex, it is reified by a `Control` object associated to the link. To illustrate the advantage of considering controls explicitly in link composition, this chapter presents a short example: it shows the limits of the model built in chapter 8, then leads to introduce new control-level artefacts, which will be used in chapter 10 to refine Reflex composition mechanisms.

9.2 A composition example

Let's consider two links applying at the same point of a base program, both before and after the joinpoint is reached (i.e. their control is `before-after`). To make it clearer, let's imagine these links correspond to a timer and a scheduler aspect respectively, both applying on the execution of a method called `foo`. The timer triggers a counter before `foo` is called (TB); when `foo` returns, it displays the time elapsed (TA). By the way, the other aspect schedules the calls to `foo` and locks variables before it method executes (SB); when `foo` returns, it unlocks the variables (SA).

[1] With or without call to proceed().

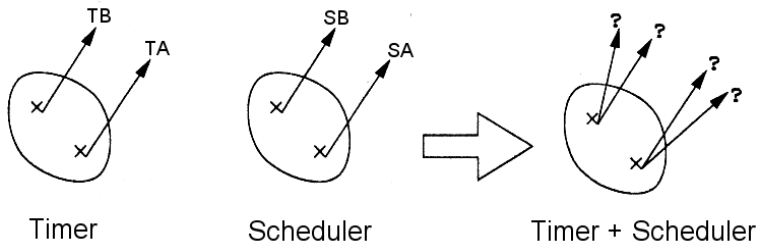

Figure 9.1: Two before-after aspects make four possibilities.

Timer and scheduler are composed of four behaviour fragments: TB, TA, SB and SA. Figure 9.1 shows how they possibly combine:

1. First, the timer can be given priority over the scheduler.
2. Alternatively, the scheduler can execute before the timer.
3. The timer can *wrap* scheduling, which means it starts before the scheduler executes, then stops the counter after variables are unlocked.
4. Finally, it is possible for the scheduler to wrap the timer execution.

9.3 Working with link operators

Let's now see how one can express the four compositions using the mechanisms from chapter 8:

- (1) and (2) are easy to specify: they correspond to standard ordering, relating to the `Seq` operator introduced in 8.3.2.
- (3) and (4) are more difficult to express: indeed, wrapping is a new relation between two links. To distinguish wrapping from ordering, a new operator is required. Let's call `Wrap` this new operator.

If T represents the timer link and S the scheduler link, behaviours (1) to (4) can be expressed using `Wrap` and `Seq` as follows:

1. `Seq(T,S)`
2. `Seq(S,T)`
3. `Wrap(T,S)`
4. `Wrap(S,T)`

Actually, one can only see the difference between `Wrap` and `Seq` by looking at the ordering of *link fragments*, i.e. the behaviour fragments around the joinpoint. To apply these operators correctly, the kernel must make the difference at the level of controls: the link level of abstraction is not sufficient.

Since we are not concerned with the application phase, we may assume that the kernel knows how to apply control-level relations. Nevertheless, the example is useful, considering it shows that operators not only need to express link-level, but also *control-level* relations.

9.4 Introducing control-level operators

Let's now observe the consequences of introducing explicitly control-level operators, i.e. at the level of behaviour fragments associated to specific link control positions.

Considering again the four compositions, and expressing them with control-level relations, one notices only one type of relation is still required, namely ordering. Indeed, if we call `Ord` the control-level operator representing ordering (similarly to `Seq` at the link level), compositions can be written as follows:

1. `Ord(TB,SB)` plus `Ord(TA,SA)`
2. `Ord(SB,TB)` plus `Ord(SA,TA)`
3. `Ord(TB,SB)` plus `Ord(SA,TA)`
4. `Ord(SB,TB)` plus `Ord(TA,SA)`

These relations are control-level translations of previous `Seq` and `Wrap` relations: one may say these link-level rules were *expanded* to control-level relations.

Clearly, explicitly control-level relations seem more expressive than link operators, which hide semantics since they only apparently work at the link level. From a kernel point of view, working at a lower level reduces the number of possible relations, therefore the minimal number of operators required for ensuring the translation of any composition.

9.5 Conclusion

In this chapter we have seen that a composition model based on link-level relations may have serious drawbacks in terms of expressiveness. Actually, link operators lead to the definition and implementation of more operators than control-level relations, to express even basic composition specifications. Additional operator definitions are supposed to balance the poor expressivity resulting from the lack of controls in the model. However, it is complex and clearly not natural to reason about control-level semantics using operators supposed to reflect link-level relations.

As a consequence, next chapter (10) refines the composition model and mechanisms in order to manage link controls explicitly.

Chapter 10

Refining composition with controls

In this chapter, composition is refined by explicitly considering link controls and control-level relations. It leads to identify a set of minimal *kernel operators*, which in the end provides solutions to some of the limits from the previous model.

10.1 Introduction

As seen previously, Reflex links apply at given points in the base program with respect to the cut expressed by their hookset. Precisely, links are specified to apply before, after, before and after or replace the joinpoint, depending on their *control* attribute.

According to chapter 9 showing the benefits of reasonning about composition at a lower level than links, this chapter refines the model designed in chapter 8, extending its granularity to the level of controls. First section presents a new composition model and associated primitives. Secondly, the framework is modified. Third part provides implementation details.

10.2 Model

This section introduces successively a new abstraction for reasonning about links and controls, then explicit control-level relations, lastly a data structure to manipulate the new primitives and express composition solutions.

10.2.1 A new control-level abstraction

In order to reason explicitly at the control level, a new abstraction is used as the composition primitive (i.e. the artefact manipulated by the kernel). We

call *link element*[1] the association of a link and a control; for instance TB, TA, SB and SA from 9.4 (even if we called them *link fragments* at first). In Reflex, controls are accessible from links; therefore, it could be argued that link elements merely consist of a shortcut abstraction. Actually, section 10.3 will show several reasons for manipulating these artefacts instead of links.

10.2.2 Kernel and user operators

Control-level operators being more expressive, the kernel concentrates on them and does not support higher-level specifications explicitly. From this point forward, we call *kernel operators* low-level operators, such as `Ord` from 9.4, which reflects ordering between link elements.

By the way, we prefer to keep some higher-level operators on top of link element operators. Indeed, a link-level operator is generally more convenient to manipulate than its low-level translation: as an example, a `Seq` is generally more convenient to declare than multiple `Ord`. We call high-level operators *user operators*.

To sum up, kernel users and plugins declare composition rules made of user operators and links; these high-level rules automatically expand to low-level relations, made of kernel operators and link elements, which are interpreted by the kernel.

10.2.3 Looking for a minimal set of kernel operators

Expanding link-level relations to control-level rules reduces the number of required operators at the kernel level. For instance, both `Seq` and `Wrap` expand to the same kernel operator relation, namely `Ord`. From this observation, one tries to identify a *minimal set of kernel operators*. By definition, this is the smallest set of operators required for expressing any relation between link elements. By definition, such operators cannot be expanded themselves; besides, they are orthogonal to each other; thirdly, we suppose they are binary, which is equivalent to state that n-ary link element operators can be expanded to a set of binary operators.

[1] The name comes from the association of a Link and a HookElement, which encapsulates the control in the current Reflex implementaton.

For the moment, three minimal kernel operators have been identified:

The ordering operator: `Ord` represents ordering between two link elements.

The nesting operator: `Nest` reflects the same relation as precedence in AspectJ (a nested advice applies if and when its parent around advice invokes `proceed`), the nesting element being associated to a replace control.

The conditional operator: `Cond` expresses that a link element applies only if a given condition is verified. Contrary to `Ord` and `Nest`, `Cond` requires no link element as left argument but a boolean expression.

Conditional relations are already implemented in Reflex as link activations (4.2). Indeed, activations refer to standard links, nonetheless a given control can be checked by adding an appropriate clause to the activation condition: for instance, `Cond(booleanexpr, linkplusbefore)` can be written as the activation condition `(booleanexpr && controlisbefore)`[2]. Consequently, there is no need to implement the `Cond` operator explicitly.

10.2.4 A data structure for link elements

Ordering and nesting can be used as two orthogonal dimensions for sorting link elements and applying their semantics. The following diagram shows how one can sort elements with respect to these concerns:

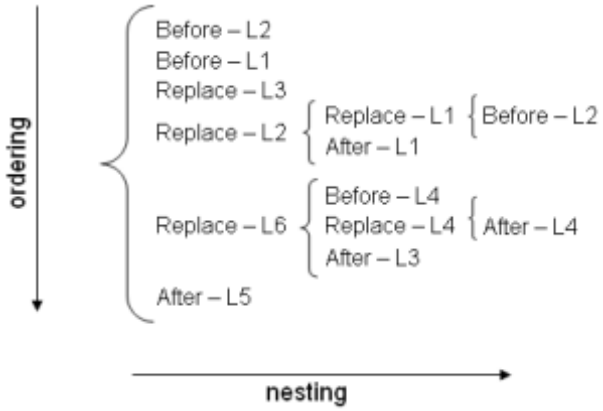

Figure 10.1: A set of link elements sorted with respect to ordering and nesting dimensions; elements should be applied from top to bottom and left to right.

[2]This is obviously pseudo-code.

Ordering and nesting orthogonal dimensions form a kind of 2D link element 'map', which specifies how to apply interacting link elements (Figure 10.1).

We call *hooknode* the corresponding data structure for link elements. A hooknode is composed of three sets: the `before`, the `replace` and the `after` link elements. Each `replace` element nests its own child hooknode, the hierarchy of hooknodes reflecting the levels of nesting between link elements.

As a conclusion, a hooknode is a convenient artefact to represent compositions of link elements, therefore links.

10.3 Framework

This section shows how the modifications on the model and primitives enable an earlier management of composition specifications. Secondly, composition phases are reviewed in this new context.

10.3.1 Early rule management

In chapter 8, the composition operator family was open-ended, which made it complex to merge rules or check consistency among specifications. With the identification of a minimal set of kernel operators, each rule becomes a combination of a small and fixed number of relations (namely: ordering, nesting and the conditionnal relation). As a result, rules are easier to merge, rule contradictions simpler to detect.

For the second version of the Reflex composition model, we decide to process rules and detect contradictions as soon as their declaration (i.e. specification phase). Each rule newly declared to the kernel is instantly translated into kernel rules between link elements.

The problem is, one doesn't know at specification time which link elements will be really involved in an interaction, therefore which control bind to the links when expanding user rules. To illustrate this, let's see how a basic rule `Seq(L1,L2)` may be interpreted differently depending on the interaction context:

- If one considers that L1 and L2 both apply before the given joinpoint, then the user rule `Seq(L1,L2)` means `Ord(L1_with_before,L2_with_before)` at the kernel level.
- If both links apply after the joinpoint, then the rule corresponds to `Ord(L1_with_after,L2_with_after)`.

- Similarly, if links are associated to replace controls, then the rule must be interpreted as `Ord(L1_with_replace,L2_with_replace)`.
- Finally, if L1 and L2 do not apply with the same control, then their interaction is trivial to solve, considering a before applies before a replace, itself applying before an after element.

Actually, high-level rules can have multiple translations depending on the context. Nevertheless, those kernel rules are complementary, since each one is the specification for a different case of control. Consequently, high-level rules can be translated into the sum of their (hypothetic) translations. As an illustration, `Seq(L1,L2)` is equivalent to the sum of the *hypothetic kernel rules* listed above: depending on the control, only the appropriate rule will apply.

However, manipulating hypothetic rules raises a problem with link elements. At specification time, when user rules are translated into hypothetic kernel rules, there is no guarantee that the link elements involved will correspond to an existing link with the right control. To manipulate hypothetic rules, *hypothetic link elements* are required. Actually, this is a good reason to use a specific artefact for the concept of link elements (i.e. a link-control pair): indeed, it would be complicated to instantiate hypothetic links with hypothetic controls without modifying the Reflex link model and implementation.

To conclude, high-level rules expand into hypothetic kernel rules (while it is checked rules keep consistent between each other), then hypothetic link elements are bound to concrete links at resolution time.

10.3.2 Composition phases

Specification phase

The difference with chapter 8 lies in rule processing mechanisms. Even if nothing changes from a user point of view, newly declared rules are interpreted while added to the specification pool. If something wrong is detected with the rule, for instance a contradiction with a previous rule, then a feedback message is displayed very early. Moreover, it improves the resolution of conflicts, in terms of performance as well as complexity.

Detection

Since the granularity of composition gets to a lower level than links, so should the definition of interactions. However, it was chosen not to modify it: indeed, if two links apply at the same point of the program, even if they have different controls (then resolution is trivial and there is no real

conflict), it might be better to warn the user anyway and ask for an explicit specification for their interaction.

Resolution

For manipulating and sorting link elements, as well as expressing the solution to a link element interaction, the hooknode data structure is used:

1. First, a hooknode is created and filled with interacting link elements, corresponding to interacting links associated with their control, placed in the appropriate set inside the hooknode with respect to this control.

2. Secondly, the hooknode inspects its `replace` elements and creates the nesting hooknode hierarchy recursively from nesting kernel rules. In practice, nested hooknodes are created for `replace` elements, and gather their nested elements as nesting rules are being processed. It results in a two dimension link element structure such as Figure 10.1.

3. Thirdly, each link element subset in the hooknodes is ordered with respect to ordering kernel rules.

Application

The way interaction results are applied is still not presented here, however one can expect hooknodes to be easier to apply than a standard composition rule: as a matter of fact, their structure reflects only two types of relations, namely ordering and nesting, whereas AST rules are open-operator specifications.

10.4 Implementation

This section presents the implementation for kernel rule and link element primitives, the hooknode data structure and high-level operators.

10.4.1 Kernel rules and partial specification

AST rules as seen in chapter 8 represent high-level user rules. For low-level kernel rules, no new artefact is introduced.

First, conditional rules are tranlated in terms of activation conditions, which are already implemented in Reflex.

Concerning ordering and nesting, the information is stored in link elements themselves, as *predecessors*: a link element is a predecessor if it applies before another one (*ordering predecessor*) or if it nests another one (*nesting*

predecessor). Each link element has two lists of predecessors, for ordering and nesting respectively. This way the `Link` implementation is not modified, which keeps the original reflective model as pure as possible. By the way, link elements are abstractions dedicated to composition, therefore it seems more natural and object-oriented to make them encapsulate the information about their composition.

Whenever a rule introduces a new predecessor, all its own predecessors are added recursively. Consequently, it is easy to check that there is no contradiction or cyclic specification: for instance, `Ord(LE1,LE2)` added to `Ord(LE2,LE1)` makes LE2 (or LE1 respectively) its own predecessor, which reflects a contradiction. For more illustration, appendix A shows the different possible errors, their associated message and description.

Besides, the object-oriented solution consisting in storing predecessor lists in link elements is better than updating a global list, since it enables *partial specification of composition*. Partial specification means that not all ordering and nesting relations between all link elements of the application must be specified explicitly. First, several trivial rules can be deducted by the kernel. Secondly, the predecessors mechanism allows one to specifies that a given element applies, for instance, before a second one, without specifying anything with respect to a third one; conversely, a single ordered list reflects the specification for each pair of elements.

To illustrate *partial ordering*, let's imagine that links L1, L2, L3 and L4 interact, L4 with a `replace` control, the others with a `before` position. Considering rules `Seq(L1,L2)` and `Wrap(L4,L3)`, the kernel applies L1, then L2, then L4 wrapping L3. Actually, several mechanisms combine to build this solution: first and obviously, the relations explicitly expressed by the rules; secondly, the fact that a `before` applies before a `replace`, which applies before an `after`[3]; thirdly, two elements not at the same level of nesting[4] do not need to be ordered. Both last principles correspond to what we call *trivial rules*, and hopefully users do not have to declare them explicitly. Besides, even if there were other links in the application, no more specification would be required as long as they do not interact with L1, L2, L3 and L4.

Concerning *partial nesting*, the mechanisms are very similar.

[3]Assuming they are at the same level of nesting (L1 and L2 for instance).
[4]L1 and L3 for instance.

10.4.2 Link elements

`LinkElement` instances are composed of a `Link` and a `HookElement` encapsulating the control. Each link element contains two lists of predecessors and exposes two associated methods `addNestingPredecessor` and `addOrderingPredecessor`. Each time a predecessor is added, all its own predecessors are added recursively, and it is verified there is no contradiction.

Three classes extend `LinkElement`, namely `Before`, `Replace` and `After`. In particular, `Replace` elements have an additional attribute which is the nested hooknode used during the resolution phase.

Besides, `LinkElement` overrides the `equals` and `hashcode` methods from `Object`, which helps a `LinkElementRepository` in managing link element instances without duplicates.

10.4.3 Hooknodes

Hooknodes consist of three sets of link elements, namely `Before`, `Replace` and `After` elements. Like kinds of containers, they provide methods to check the existence, add or remove link elements.

More interestingly, the `needsResolution` method implements the detection of interactions. Interactions are detected as the presence of more than one link in the hooknode (10.3.2), nevertheless this definition can be modified (so that for instance a `before` and an `after` elements at the same level of nesting are not considered to interact).

Last but not least, the `solve` method is called at the end of the resolution phase, responsible for filling nested hooknodes as well as ordering elements according to rules. Sorting is performed via specific `Comparator` extensions, looking for predecessors to sort link elements. Consequently, the sorting algorithm is the traditional `sort` algorithm provided by Java. Appendix C explains why we prefered this algorithm to a not-less-traditional Quicksort.

To sum up, a hooknode is a triple list of link elements, created when an interaction occurs, which structures itself to reflect ordering and nesting specifications. As a result, hooknodes compose the data structure to express link element compositions.

10.4.4 Operators and open-operator support

In this second composition model, ASTs do not express low-level compositions anymore; instead they expand to link element relations. Link elements represent compositions once sorted in their specific data structure according to kernel rules. There is no rule minimization anymore, nonetheless operators must specify the way they expand to kernel rules.

In practice, each operator implements the `apply()` method, responsible for translating its semantics into kernel relations. Since many user operators expand by considering each link pair successively to form binary relations, a specific subclass provides an algorithm implementing the decomposition mechanism, allowing inheriting operators to focus on their implementation of `apply(Link,Link)`. By the way, they may subclass other operators or express their semantics in terms of other operators.

To facilitate the implementation of operators, a set of methods is provided by their superclass: `ord`, `nest` and `cond` permit to declare kernel rules between two link elements as standard method calls with two arguments, whereas `before`, `replace` and `after` accept a link argument and create the corresponding hypothetic link element (which is also registered to the repository). As a result, operator implementors can express kernel rules as simply as, for instance, `ord(before(link1),before(link2))`. As a better illustration, chapter 12 exposes the implementation of `Seq` and `Wrap` user operators.

Finally, operators are responsible for translating themselves to kernel relations, either explicitly or by subclassing or composing existing operators.

10.5 Conclusion

From the explicit consideration of controls in composition, we have built an alternative to the solution in chapter 8. We have identified a minimal set of kernel operators and introduced two new artefacts, namely link elements and hooknodes, which are not new concepts but convenient abstractions for the design and implementation of this second solution.

The first benefit is the early processing of composition rules, in particular the early detection of contradictions. User operators, made of other high-level operators or well-known low-level relations, are also easier to implement and reason about. Finally, the application phase should be easier to implement, because of the closed family of kernel operators in comparison with chapter 8.

Part IV

Examples

Chapter 11

Plugin examples

In order to validate the plugin architecture presented in chapter 7, several translators were implemented in parallel to the infrastructure itself. This chapter exposes their implementation as plugin examples.

The chapter is structured as follows. First section provides general information about implementing plugins (11.1). Secondly, a plugin called Profiler is described in details, from the definition of its purpose to its code generation (11.2). Then, two slightly more complex plugins are presented in order to illustrate the various applications for plugins: ReifyCalls facilitates the reification of messages (11.3), whereas the SOM plugin implements a concurrency model (11.4).

11.1 Implementing new plugins

As seen in chapter 7, plugins share information with the kernel, in particular their identity, their loadtime arguments and the source of the composition rules they may declare.

In practice, plugins must implement three respective protocols: `Describable`, `ArgumentHandler` and `RuleOwner`. The following piece of code illustrates how one can implement those protocols to create a new plugin class from scratch:

```
public class DummyPlugin extends Plugin {

    // Constructor
    public MyPlugin() {
        super("my plugin name", "my plugin description");
    }

    // This method is called by the argument dispatcher
    // @param the arguments mapped to their tag
    public void handleArgs(Map aTagsArguments) {}

    // @return supported tags
    public String[] getTags() {return new String[0];}

    // @return the tags description
    public String getTagsDescription() {return "";}

    // @return the source of a composition rule
    // declared by the plugin
    public String getSource(Rule aRule) {return "";}
}
```

The argument dispatcher is responsible for sending dedicated tags to each plugin. It corresponds to `aTagsArguments` in the `handleArgs` signature, which is filled by the dispatcher with tag-argument pairs intended for the plugin.

The example shows that plugins may have no tag, then `supportedTags` returns an empty array. It means that the plugin needs no loadtime argument to achieve its purpose, nevertheless it may receive runtime arguments through any public API it implements (at least `handleArgs` is available for sending instructions to the plugin).

In order to make plugin implementations more convenient, several `Plugin` subclasses are provided as Reflex tools.

For instance, `AntlrParserPlugin` implements plugins using ANTLR technologies [ANTLR.org] to parse string arguments from either startup arguments or a given configuration file. The following code shows how to implement a new ANTLR-based plugin class from `AntlrParserPlugin`:

```
public class MyPlugin extends AntlrParserPlugin {

    // Constructor
    public MyPlugin() {
        super("my plugin name", "my plugin description",
                new String[0], // Tags
                "no tag for this example");}

    // @param a tag supported by the plugin
    // @return the associated lexer class
    protected Class getAntlrLexerClass(String aTag)
    {return MyLexer.class;}

    // @param a tag supported by the plugin
    // @return the associated parser class
    protected Class getAntlrParserClass(String aTag)
    {return MyParser.class;}

    // @return the input source of a composition rule
    // declared by this plugin
    public String getSource(Rule aRule) {return "";}
}
```

The abstract `AntlrParserPlugin` class is in charge of handling arguments and parsing commands. It handles multiple tags (possibly associated to different lexers and parsers) and multiple arguments. Each argument is recognized either as an individual command or a filename (a bundle of commands). Finally, each command is parsed using the appropriate lexer and parser, the process being triggered by a call to `start`, which is implemented by ANTLR parsers. On the other hand, the parser is responsible for calling appropriate methods from the plugin API to make it perform kernel configurations with respect to parsed commands.

Appendix B explains the reasons why we chose ANTLR to implement our first plugin examples. Nevertheless, only the call to `start` is specific to ANTLR, besides it is embedded in a tool class rather than the core plugin architecture. Obviously, one can easily implement tools based on other existing parsing technologies.

11.2 The Profiler plugin

Purpose, usage

The profiler plugin is an extension of the timer introduced in 9.2. It logs method calls and their return time, being triggered by commands like:

`profile: "`*methodName*`" fromClass:` *className*`;`

Single commands can be sent to the plugin using a specific tag, `-profile`, followed by the command string. For multiple profiling, commands can be gathered in a text configuration file, then the filename follows the tag.

The following trace shows a typical Profiler log:

(...)
```
[ProfilerPlugin] Method 'foo' is called (call #1).
```
(method 'foo' executes)
```
[ProfilerPlugin] Method 'foo' returns (after 862 milliseconds).
```
(...)

Design

`ProfilerPlugin` subclasses `AntlrParsersPlugin` (11.1). The plugin declares its name, its description, the `-profile` tag and its description, as well as `ProfilerLexer` and `ProfilerParser` ANTLR classes. It gets its arguments from the Reflex dispatcher through `handleArgs`, which parses commands and filenames using ANTLR, then configures the kernel. Lexers and parsers are embedded inside the Profiler JAR archive: `ProfileParser` is implemented so that correct commands (matching the Profiler-specific grammar) trigger calls to `profile(String aMethod, String aClass)` from the plugin API. This method performs configuration as follows:

1. Register the *message receive* operation if necessary.
2. Create a hookset for the appropriate method-class cut.
3. Get a new link, configure it as *before-after*, binds it to a counter.
4. Wrap the link into a linkset, then send it to the kernel.

The counter is a metaobject. The `Counter` class inside the JAR archive implements two methods, called respectively before and after the execution of the method, performing the log.

Source code

Four classes compose the Profiler JAR: the plugin itself, the counter, the lexer and parser.

The following code implements the plugin:

```
public class ProfilerPlugin extends AntlrParsersPlugin {

    public ProfilerPlugin() {
        super("ProfilerPlugin", "A plugin to log invocations
            on a method and their return time",
            "-profile",
            "[file1:file2:...] \\n\\t to log invocations on given
            methods and their return time, directly as a string
            command or in a text file (syntax: 'profile:
            \"method_name\" fromClass: class_name;')");}

    public String getSource(Rule aRule) {
        return null;} //This plugin does not declare composition rules

    protected Class getLexerClass() {
        return ProfilerLexer.class;}

    protected Class getParserClass() {
        return ProfilerParser.class;}

    public void profile(String aMethod, String aClass) {
        System.out.println("[" + getID() + "] Applying: 'profile: "
            + aMethod + " fromClass: " + aClass + "'.");

        //Declare the MsgReceive operation if necessary
        if (PluginAPI.operations().existOperationSupportFor(MsgReceive.class)
            && !(PluginAPI.operations().getOperationSupport(MsgReceive.class)
                instanceof MsgReceiveInstaller))
            System.err.println("[ProfilerPlugin] Warning: a different
                installer is already defined for operation MsgReceive.");
        } else
            PluginAPI.operations().addSupport(new OperationSupport(
                MsgReceive.class, new MsgReceiveInstaller()));
        }

        //Create yhe hookset (cut)
        PrimitiveHookset theHookset =
            new PrimitiveHookset(MsgReceive.class,
                new NameCS(aClass),
                new NameOS(aMethod));
```

```
        //Get a new link
        String theName = "PROFILER_" + aClass.substring(
            aClass.lastIndexOf('.')+1) + "_" + aMethod;
        BLink theLink = PluginAPI.links().getBLink(theHookset,
            new MODefinition.MOClass(Counter.class.getName()), theName);

        //Set up the link
        theLink.setInitialization(Initialization.EAGER);
        theLink.setControl(Control.BEFORE_AFTER);

        //Wrap and declare the link
        Linkset theLinkset = new Linkset(this, theName+"_Linkset");
        theLinkset.addLink(theLink);
        PluginAPI.links().addLinkset(theLinkset);
    }
}
```

The following class implements the counter metaobject:

```
public class Counter implements BeforeMsgReceive, AfterMsgReceive {

    private int itsCounter = 0;
    private long itsInitialDate;

    public void beforeMsgReceive(Object[] aReifiedData) throws MOPException {
        String theMethodName =
            DMsgReceive.convert(aReifiedData).getMethod().getName();
        System.out.println("[ProfilerPlugin] Method '" + theMethodName
            + "' is called (call #"+(++itsCounter)+").");
        itsInitialDate = System.currentTimeMillis();
    }

    public void afterMsgReceive(Object[] aReifiedData) throws MOPException {
        long theTimeElapsed = System.currentTimeMillis() - itsInitialDate;
        String theMethodName =
            DMsgReceive.convert(aReifiedData).getMethod().getName();
        System.out.println("[ProfilerPlugin] Method '" + theMethodName
            + "' returns (after "+theTimeElapsed+" milliseconds).");
    }
}
```

Concerning the lexer and parser, they are generated from the following ANTLR grammar specification:

```
class ProfilerParser extends Parser;

start :( rule )+ ;

rule
    {itsMethod = null; itsClass = null;}
    :PR m:METH SP FC c:CLASS SEMI ( NEWLINE )*
        {itsClass = c.getText();
        String theString = m.getText();
        itsMethod = theString.substring(1, theString.length()-1);
        itsPlugin.profile(itsMethod, itsClass);};


class ProfilerLexer extends Lexer;

options { k=10; }

PR  :"profile" COL SP;

FC  :"fromClass" COL SP;

CLASS: ( ('a'..'z') ('a'..'z'|'A'..'Z'|'0'..'9')* PT )*
    ('A'..'Z') ('a'..'z'|'A'..'Z'|'0'..'9')*;

METH :'"' ('a'..'z') ('a'..'z'|'A'..'Z'|'0'..'9')* '"';

NEWLINE :'\t'|'\n'|'\r';

COL :':';

PT  :'.';

SEMI:';';

SP  :' ';
```

11.3 The ReifyCalls plugin

ReifyCalls provides facilities to reify method calls on Java objects and perform shifts to metaobjects. The specific syntax is:

```
reifyCallsOn: className toMO: MOClassName;
reifyCallsOn: className toBody: @body@;
reifyCalls: "methodName" on: className toMO: MOClassName;
reifyCalls: "methodName" on: className toBody: @JavaBody@;
```

Similarly to Profiler, `ReifyCallsPlugin` extends `AntlrParsersPlugin` and deals with both individual commands and configuration files. The following commands could be a typical input file:

```
reifyCallsOn: test.Foo withMO: test.FooMO;
reifyCalls: "bar" on: test.Bar withBody:
    @System.out.println("bar");
    return null;@;
reifyCallsOn: test.Zorg withBody:
    @return DMsgReceive.convert($1).perform();@;
```

As seen in class `Counter` (11.2), metaobject are provided a reified data argument from the MOP. Here, the $1 variable is a Javassist shortcut refering to the first argument of the method, i.e. reified data. Calling the `perform` method on those data executes the original target method. Said differently, the third command in this configuration file actually does nothing: it reifies a method call then executes the method.

The ReifyCalls implementation is slightly more complex than Profiler. First, the lexer introduces method body tokens, composed of a Java body wrapped with `@` characters[1]. Secondly, ReifyCalls has a second argument which consists in the metaobject: if a class is specified, the plugin first verifies this class is valid (i.e. the metaobject implements `ReplaceMsgReceive`); in case a method body is given, it creates an appropriate metaobject from scratch (i.e. a class is created, it is made implement `ReplaceMsgReceive`, then a method is created from the given body and introduced in the class).

[1] Chosen because they are not valid in a Java expression.

11.4 The SOM plugin

The SOM plugin achieves an implementation of *Sequential Object Monitors* [Caromel *et al.*, 2004], a scheduler-like approach to concurrency in Java.

As presented in [Tanter & Noyé, 2004], the standard SOM implementation is a Reflex-based library. The SOM library is composed of a main class called `SOM`, an API class and a set of relating classes used by `SOM`.

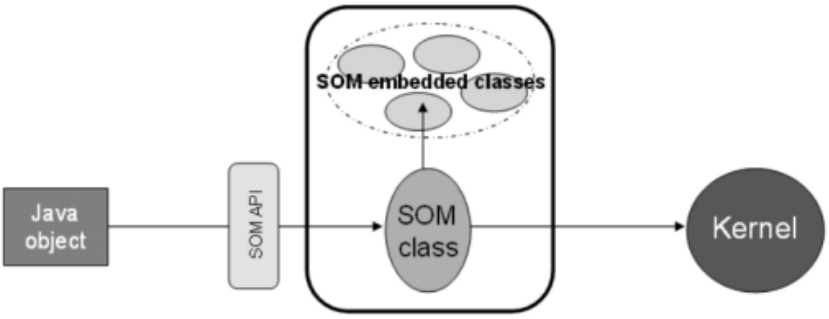

Figure 11.1: SOM library classes and API.

The SOM plugin extends the library implementation, achieving in particular a configuration language for SOM. This language is an evolution of the one proposed in [Tanter & Noyé, 2004]. Its syntax is:

```
schedule: className withScheduler: SchedulerClassName;
schedule: className withBody: @JavaBody@;
```

The parser consists in a simplified version of the ReifyCalls parser. The `SomPlugin` class replaces `SOM`, configuring Reflex with respect to the plugin framework. `SOM` still exists, but does not configure the kernel anymore, instead it acts like a public API and forwards calls to the plugin core class.

Figure 11.2 shows there are now different types of input for SOM: first, direct commands and configuration files at startup in the SOM plugin configuration language; on the other hand, calls to `SomPlugin` and `SOM` which provide the SOM programmatic access API.

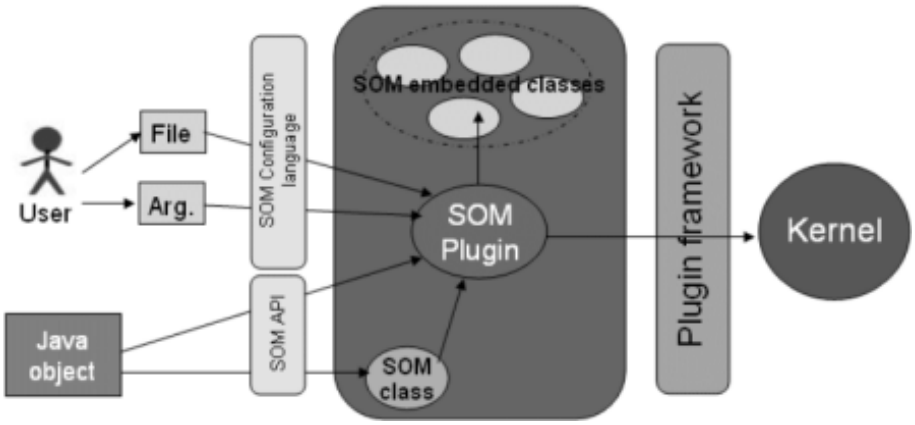

Figure 11.2: The SOM plugin classes, inputs and output.

Chapter 12

Composition examples

This chapter is concerned with illustrating composition mechanisms, in particular the implementation of operators and the usage of composition rules.

First are exposed the implementation of Seq (12.1) and Wrap (12.2) user operators as presented in chapter 9. Then, a basic interaction example is described, how default composition applies, then how one can adapt the behaviour of the application by specifying appropriate rules (12.3).

12.1 The Seq operator

The `Seq` operator reflects precedence between links. A `Seq` tree accepts any number of nested rules and link leaves.

As seen in 9.4, a link rule `Seq(link1, link2)` is equivalent for the kernel to the following control-level relations:

- `Ord(link1_with_before, link2_with_before)`
- `Ord(link1_with_replace, link2_with_replace)`
- `Ord(link1_with_after, link2_with_after)`

Consequently, the `Seq` operator is implemented as follows:

```
public class Seq extends OperatorNode {

    public Seq(Rule[] aChildren) {
        super(aChildren);}

    //Apply kernel rules
    protected void apply() {
        //Use the rule expansion algorithm
        super.apply(getChildren());}

    //Apply kernel rules for a pair of links
    protected void apply(Link aL1, Link aL2){
        //Instantiate link elements, declare kernel rules
        ord(before(aL1), before(aL2));
        ord(replace(aL1), replace(aL2));
        ord(after(aL1), after(aL2));}
}
```

The `apply()` method is called when the rule needs to be translated, i.e. at declaration time.

Since a `Seq` can be interpreted by taking its containees two by two in a certain order, `apply(getChildren())` is used to expand the rule into binary relations. The algorithm, which is implemented by the superclass, is in charge of extracting primitive links from both nested rules and linksets, making pairs in a valid order, as well as triggering calls to `apply(Link,Link)` for each pair. Actually, this last method encapsulates most of the operator semantics.

12.2 The Wrap operator

The `Wrap` user operator implementation differs from `Seq` by its kernel-level semantics considering two links (i.e. its `apply(Link,Link)` implementation). As a matter of fact, `Wrap(link1, link2)` must be translated into:

- `Ord(link1_with_before, link2_with_before)`
- `Nest(link1_with_replace, link2_with_before)`
- `Nest(link1_with_replace, link2_with_replace)`
- `Nest(link1_with_replace, link2_with_after)`
- `Ord(link2_with_after, link1_with_after)`

The following source code implements the `Wrap` operator:

```
public class Wrap extends OperatorNode {

    public Wrap(Rule[] aChildren) {
        super(aChildren);}

    //Apply kernel rules
    protected void apply() {
        //Use the rule expansion algorithm
        super.apply(getChildren());}

    //Apply kernel rules for a pair of links
    protected void apply(Link aL1, Link aL2){
        //Instantiate link elements, declare kernel rules
        ord(before(aL1), before(aL2));
        nest(replace(aL1), before(aL2));
        nest(replace(aL1), replace(aL2));
        nest(replace(aL1), after(aL2));
        ord(after(aL2), after(aL1));
}
```

12.3 Solving interactions

This section introduces an interaction example based on Profiler (11.2) and SOM (11.4). To obtain a conflict, both plugins are asked to intercept the same message: `foo` sent to class `Foo`. Reflex is started with the following arguments:

```
--pluginClasses SomPlugin ProfilerPlugin
-profile "profile: \"foo\" fromClass: Foo;"
-som "schedule: Foo withScheduler: MyScheduler;"
Main
```

The first line loads both plugins[1]; it could have be done automatically by placing them as JARs in the plugin autoload folder. Second and third lines configure Profiler and SOM to intercept calls to `Foo.foo()`. Finally, the application main class is called (fourth line), creating a `Foo` instance then calling `foo` to get the conflict.

Default composition

When Reflex starts with the arguments above, the kernel detects the interaction of both links, which try to apply at the same point. By default, links are composed in sequence and the following warning is displayed:

```
[HookFactory] Warning: no composition specification to sort
beforeafter(profiler_link) and beforeafter(som_link). Links
will be installed in sequence.
```

In practice, links apply as follows:

1. Profiler records the current date as the starting of `foo`.
2. SOM calls the appropriate scheduler before `foo` is called.
3. (`foo` executes)
4. Profiler considers that `foo` returned and traces the time elapsed.
5. The scheduler returns.

Consequently, the following trace displays:

```
(...)
[ProfilerPlugin] Method 'foo' is called (call #1).
[Scheduler] Scheduling 'foo'.
(foo executes)
[ProfilerPlugin] Method 'foo' returns (after 1194 milliseconds).
[Scheduler] Scheduling of 'foo' complete.
(...)
```

Semantically, the default application order does not make much sense here. In next section, we show how one can adapt composition by specifying rules.

[1] Thanks to a new handler added to Reflex for this purpose.

Adapting composition

To illustrate how one can modify the behaviour of the application in case of aspect interaction, we use two successive composition rules and check the composition trace.

First, let's specify `Wrap(profiler_link, som_link)` by adding the rule to the kernel startup arguments:

```
-rule "Wrap(profiler_link, som_link)"
--pluginClasses SomPlugin ProfilerPlugin
-profile "profile: \"foo\" fromClass: examples.Foo;"
-som "schedule: Foo withScheduler: MyScheduler;"
Main
```

The first line specifies the rule; indeed, a dedicated handler allows users to specify composition rules at startup. As a result, Profiler counts the time `foo` takes to execute as well as the time taken by the scheduler:

```
(...)
[ProfilerPlugin] Method 'foo' is called (call #1).
[Scheduler] Scheduling 'foo'.
(foo executes)
[MyScheduler] Scheduling of 'foo' complete.
[ProfilerPlugin] Method 'foo' returns (after 1286 milliseconds).
(...)
```

As an alternative, let's try `Wrap(som_link, profiler_link)`:

```
--pluginClasses SomPlugin ProfilerPlugin
-profile "profile: \"foo\" fromClass: examples.Foo;"
-som "schedule: Foo withScheduler: MyScheduler;"
-rule "Wrap(som_link, profiler_link)"
Main
```

Consequently, the counter measures the time taken by `foo` to return, wrapped by the scheduler:

```
(...)
[MyScheduler] Scheduling 'foo'.
[ProfilerPlugin] Method 'foo' is called (call #1).
(foo executes)
[ProfilerPlugin] Method 'foo' returns (after 145 milliseconds).
[MyScheduler] Scheduling of 'foo' complete.
(...)
```

As a conclusion, one can easily adapt composition semantics, declaring composition rules at startup or anytime before the interaction occurs. One must notice however, that Reflex applies these semantics only once when classes are loaded, which limits the actual scope of runtime composition specifications.

Part V

Conclusion

Chapter 13

Related work

As seen in the introduction, a variety of approaches exist for separating crosscutting concerns. Among them, several relate directly to this work by promoting versatile solutions to AOP. This section presents those technologies through their similarities and differences with the Reflex AOP kernel.

13.1 CME

The *concern manipulation environment* ([CME]) developed at IBM is an ambitious large-scale project, aiming to support AOSD at any level (analysis, design, implementation, etc.) with respect to any object-oriented computing environment (programs in various languages as well as artefacts like UML diagrams), targeting both asymmetric and symmetric approaches.

In particular, the *concern assembly toolkit* (CAT [Harrison *et al.*, 2002]) provides as part of the CME abstractions and support for *concern assembly*. Concern assembly is defined as the low-level manipulation of software artefacts to implement "composition and weaving" from AOSD approaches. Actually the paper does not refer to aspect composition, but the composition of class hierarchies and combination of methods. As a matter of fact, the IBM solution is not concerned with the detection of aspect interactions, their resolution nor the feedback issues emphasized by [Tanter & Noyé, 2004] and addressed by this thesis.

Even if the CAT proposal does not focus on the same issues as Reflex, it promotes the same idea that "an abstraction layer and toolkit supporting low-level concern assembly would enable AOSD tool builders to leverage many of the similarities, reducing development effort and increasing the scope and interoperation of their tools". In the following paragraphs, Reflex and the CME are compared with respect to the issues addressed by both solutions: plugin architecture, back-end model and frameworks.

In practice, the CAT toolkit is a set of interfaces, frameworks and *concern assemblers*. Concern assemblers are kinds of plugins, intended to take any type of AOSD artefacts as input and produce *assembly directives* as output. Assembly directives are independent of the output language and result in low-level modifications of the program. In comparison, Reflex plugins and the kernel itself target only asymmetric approaches, besides they are limited to Java bytecode output.

Secondly, assembly directives are expressed either in a specific *concern assembly language*, or as calls to a dedicated API. Despite a similar public API, Reflex does not promote an intermediate language. IBM designers justify this layer by the serialization of low-level assembly specifications. Indeed, once serialized specifications can be used for analysis, for instance detecting behavioural interference. However, it is not clear in the paper if the translation to the intermediate language is required, since concern assemblers can perform direct calls to the API: "[Assembly directives] can be written in the concern assembly language, a usage of XML. Alternatively, they can be issued as calls to a Java API implemented by all concern assemblers". Actually, one may wonder about the feasibility of a global analysis without a uniform process[1] for all assemblers.

By the way, IBM designers use open-ended string directives in the CAT API, since it allows any type of specification without being bound to a particular software artefact. Conversely, Reflex promotes object-oriented protocols and high-level artefacts such as hooksets or linksets.

Finally, the IBM proposal is based on a specific model for representing object-oriented concepts and directives. Indeed, the CAT paper introduces its own object-oriented and organizational elements, as well a set of directives and protocols. In comparison, Reflex is based on a reflective model, over which are defined additional abstractions such as hooksets and linksets.

To conclude, the CME provides an alternative to the language layer proposed in this thesis, intended to be as generic as possible. However, the IBM proposal does not consider explicitly aspect composition issues, which are addressed by Reflex.

[1] For instance: translating semantics into written directives, then calling the API.

13.2 AOLMP

Aspect-oriented logic meta programming (AOLMP) is an expressive approach to building composable aspect-specific languages. It results from the unification of aspect declarations and weaver implementations in a common logic programming language. The solution promoted is very powerful and pure, in particular for aspect composition: indeed, only one (general-purpose logic meta) language is used to reason about aspects, both from a user and an implementor point of view.

The main difference with Reflex lies in that no aspect-specific syntax is provided beyond the kernel (logic) framework. In comparison, a user-friendly language layer as introduced in 4.4 provides the following benefits:

- No need to know the kernel underlying model to use aspects from a given AOL: they can simply be 'plugged in'.
- Using an aspect-specific syntax guides the aspect writer, who is also shielded from the complexity and the expressiveness of the kernel model.

As a conclusion, AOLMP makes a powerful solution to composing aspects, however it does not address the language layer issue.

13.3 XAspects

XAspects [Shonle *et al.*, 2003] is a plugin mechanism that integrates DSALs and Domain-Specific Component Languages (DSCLs) with AspectJ.

The XAspects architecture is intended to make plugins cooperate with each other. In practice, each plugin performs changes to the base program in a 6-phase conversation with the XAspects compiler, which is actually composed of two main steps :

- First, each plugin sends changes[2] to the compiler, from the analysis of its input aspect body.
- Secondly, plugins may send new instructions, from the analysis of the bytecode produced by the compiler after phase one.

Actually, aspects cooperate with each other as plugins do to apply their changes. However, XAspects provides neither analysis nor control over aspect interactions, no more than user feedback in case of conflict (not even the names of involved plugins or aspects).

[2] As Java/AspectJ code.

Turning on to XAspects limitations, [Shonle *et al.*, 2003] describes several requirements on aspect bodies, naming restrictions (in particular for introductions), as well as more serious restrictions: in particular, structural changes are forbidden during the second phase, and composition is limited to (AspectJ) precedence.

A look at the source code examples shows that there is no real translation mechanism in XAspects: provided examples are either trivial or based on DAJ tools [DAJ.sf.net, Lieberherr] to create Java/AspectJ files. Basically, the XAspects compiler wraps ajc[3]. If one compares XAspects with using DAJ and ajc directly, the benefits are:

- XAspects theoretically prevents from having to weave aspects in a special order, since each plugin gets the bytecode resulting from the application of all aspects in phase one.
- It is possible to gather multiple aspects and associate them with a particular plugin inside a single XAspects source file.
- The programmer needs only one command to perform the whole compilation, since DAJ is executed behind the XAspects compiler.

Actually, one sees that XAspects deals mainly with the presentation issue caused In terms of source code artefacts and weaving commands by multiple aspects in different languages.

Comparing XAspects and Reflex, three issues are highlighted:

Aspect presentation: XAspects provides a solution to gathering multiple aspects written in various languages. Conversely, Reflex does not deal with this issue for the moment.

Aspect translation: the responsibility of translating aspect semantics into an intermediate lodel or language is left by XAspects to DAJ. Conversely, this thesis directly addresses the issue with plugins.

Intermediate language: XAspects uses AspectJ as the intermediate language, which is a serious limitation (in particular for composition). In comparison, Reflex uses reflexive primitives and an open composition model.

As a conclusion, one should state XAspects and Reflex address different issues.

[3]The AspectJ compiler.

13.4 Josh

Josh [Chiba & Nakagawa, 2004] is an open-AspectJ-like language that allows to experiment new AOLs. Generally speaking, both Josh motivation and realization are very close to Reflex.

Concerning the differences between both proposals, Josh inlines aspect bodies, which contrasts with Reflex representing aspects via objects. Inline weaving however disables a variety of features relating to Java security, as well as per-object aspects and aspects of aspects capabilities.

Besides, Josh is aimed at experimenting languages, rather than combining them. As a result, no support is provided for aspect composition, whereas Reflex ensures the detection of interactions and provides means to resolve them.

Chapter 14

Thesis overview

As highlighted by the structure of the dissertation, this thesis exposes a solution built incrementally.

First, chapter 6 introduces a new abstraction in the Reflex kernel, namely *linksets*, which reify semantic groups of links. As such they can typically reflect crosscutting concerns, and provide a convenient abstraction for reasoning about, tracing and composing aspects.
Secondly, chapter 7 proposes a *plugin architecture* to cleanly encapsulate and integrate aspect language-specific translators in the Reflex AOP kernel. This infrastructure implements the language layer introduced in 4.4 and [Tanter & Noyé, 2004]. Moreover, its design supports and promotes traceability and smart user feedback. Finally, the way translators are defined, i.e. extending Reflex standard handlers, facilitates their design and implementation.

Turning on to aspect composition issues, chapter 8 promotes a generic model for *link composition*. Design and implementation are exposed successively, as well as a dedicated user framework to specify composition at load time or dynamically.
Chapter 9 highlights the benefits of considering *link controls* explicitly in the composition model, and shows how it promotes a cleaner design of composition operators.
As a consequence, chapter 10 proposes a new composition model, based on explicit controls as well as links. Reasoning about controls leads to distinguish between two types of operators, called *kernel* and *user operators*. Finally, a minimal set of operators is identified, which enables kernel composition features, such as the early detection of rule contradictions.

Finally, a set of working examples illustrates and validates the implementation of the thesis.

Chapter 15

Perspectives

This last section presents several future works and perspectives for the solution built in this thesis, and the Reflex AOP kernel in general.

First, a minimal set of operators was identified to express composition in an open manner. Assuming the correctness of its specification, a variety of operators and language translators are expected to be released in the next future, similarly to the recently implemented AspectJ plugin for Reflex.

Secondly, advanced rule consistency analysis and checks could be implemented over composition mechanisms. Even if the framework built for this thesis implements several verifications over rules (cyclic rules, sequence contradictions, etc.), interactions between aspects are far from being limited to these cases, and should be the subject of a specific study.

Thirdly, it could be studied how the implementation of linksets can influence aspect-oriented programming with Reflex. In particular, since linksets of linksets permit complex hierarchies over links, such issues as the possibility to compose linksets that do not reify a standalone concern should be studied (for instance if a linkset composed individually is part of a higher-level linkset reflecting a concern).

Focusing on AOL modules, their infrastructure could be refactored so that it accepts various layers of plugins. Said differently, the architecture could be extended to allow plugins to use other plugins, making them act more like components than standalone translators.

Finally, the composition frameworks presented in this thesis will be extended to structural kernel mechanisms.

Appendix A

Error messages

This section lists Reflex handled errors relating to linksets, plugins and composition features. Each error is presented with a specific message which illustrates Reflex traceability capabilities.

Double tag declaration

Explanation: a tag is declared twice, by two plugins or twice the same plugin.

Kernel policy: exception

Message: "Error: Tag A was declared twice (by B and C)." where A is the string tag, B and C plugin IDs.

Incorrect tag declaration

Explanation: tag is not well-formed, it should be prefixed with '-' or '--'.

Kernel policy: exception

Message: "Error: Tag A declared by B is incorrect. Tags are prefixed with '-' for basic tags or '--' for server tags." where A is the string tag and B the plugin ID.

Invalid plugin JAR

Explanation: a JAR in the plugin folder contains no subclass of Plugin.

Kernel policy: warning

Message: "Warning: File A is not a correct JAR plugin file. File will be ignored." where A is the file name.

Double plugin link nesting

Explanation: a plugin tries to add an already owned PluginLink (this includes nested links and linksets) to another parent.

Kernel policy: exception

Message: "Error: Plugin A is adding link B to parent C. B is already owned by D. A plugin link cannot have more than one parent." where A is the plugin ID, B the link ID, C and D the parents IDs.

NB: here only one plugin is mentioned. As a matter of fact, both parents (the one already set and the new one) are owned by the current plugin, since plugins cannot access links / linksets from other sources (see framework part).

Double linkset declaration

Explanation: a linkset is declared twice.

Kernel policy: warning

Message: "Error: A was declared twice by B. Second declaration will be ignored." where A is the linkset ID, B the plugin.

NB: here we already know that the linkset was declared twice by the same plugin, since plugins cannot access linksets from other sources (see framework part).

Double link ID usage

Explanation: a plugin link is declared with an already existing ID.

Kernel policy: exception

Message: "Error: Plugin A tries to use name B already used by a user link." or "Error: Plugin A tries to use name B already used by a link from plugin C." where A is the plugin ID, B the link ID, C a plugin ID.

Composition sequence contradiction (between rules)

Explanation: rules are in sequence contradiction.

Kernel policy: exception

Message: "Error: composition sequence contradiction between rule A and rule(s) B implies that C is declared anterior to itself!" where A is the string representation of the rule being applied, B the string representations of other rules in contradiction, C the string representation of the link element.

Example: "Error: composition sequence contradiction between
rule Seq(l1,l2) declared by TestPlugin [source: aspect TestAspect.rfx]
rule Wrap(l2,l3) declared by TestPlugin [source: aspect TestAspect.rfx]
rule Seq(l3,l1) declared by CompositionHandler [source: file TestInput.txt]
implies that before(l1) is anterior to itself!"

Composition sequence contradiction (within rule)

Explanation: a rule is in sequence contradiction with itself.

Kernel policy: exception

Message: "Error: composition sequence contradiction within rule A =¿ B is declared anterior to itself!" where A is the string representation of the rule and B the string representation of the link element.

Example: "Error: composition sequence contradiction within
rule Seq(Seq(l1,l2), Seq(l2,l1)) declared by CompositionHandler [source:
command line argument of tag -compositionRules]
implies that before(l1) is anterior to itself!"

Composition nesting contradictions

Nesting related error messages are similar to sequence errors. See sequence contradiction messages.

Lack of sequence rule

Explanation: no rules specify an ordering relation between two elements.

Kernel policy: warning

Message: "Warning: no composition specification to sort A and B. Links will be installed in sequence." where A and B are the string representations of both link elements.

Example: "Warning: no composition specification to sort after(l4) and after(l5). Links will be installed in sequence."

Appendix B

Parsing technologies

In order to implement several plugin examples with their configuration language, a parsing technology had to be chosen. A parser is a program that can recognize a particular language, with respect to its specific grammar. A parser generator is a program that reads a grammar, then outputs a parser to recognize the corresponding language. ANTLR [ANTLR.org] is the popular parser generator we decided to use. This section presents it and compares with alternative solutions.

ANTLR is a second generation parser generator, the first generation being the popular PCCTS also by Terence Parr. Implemented in Java, ANTLR generates Java, C++ and C# parser. Besides, ANTLR provides several advanced features which make it popular among recent parsing technologies: lookaheads bigger than 1, tree parser generation, etc.

As a parser generator, ANTLR can first be compared to hand craft parsers. The benefits of a parser generator are quite obvious: shorter development time, prevention of mistakes since most of the implementation is automated, higher level of abstraction for both the programmer and any code reader. On the other hand, generated parsers are traditionaly considered as slow, using lots of memory and providing poor error reporting. Concerning the amounts of time and memory needed by generated parsers, most claim they are insignificant on modern computers. By the way, it is generally admitted that ANTLR has a fairly good error reporting.

Moreover, we compared ANTLR with the most famous UNIX program YACC [YACC] and the recent JavaCC by Sun Microsystems [JavaCC]. First, the YACC table driven solution makes parsers difficult to debug: indeed, parser state transitions result from values in a table. Besides, YACC is limited to a lookahead of 1, it cannot produce C++ parsers, and its er-

ror reporting is generally considered as poor[1]. Turning on to JavaCC, the benefits are less obvious: even if several feedbacks are in favor of ANTLR on the internet, it is generally accepted that few differences exist. Nevertheless, JavaCC does not generate C++, C# nor tree parsers. The ANTLR licence is also less restrictive (pure open source). Finally, Sun now supports ANTLR, after having supported JavaCC for a while.

To conclude, we thought ANTLR was both a convenient and promising parsing technology, therefore we chose it for implementing plugin examples.

[1]Particularly when compared to ANTLR.

Appendix C

Ordering algorithms

In order to sort link elements with respect to their ordering or nesting predecessors, a specific algorithm had to be used or implemented. This section presents the `sort` polymorphic algorithm from the Java JDK [algorithms], which we decided to use rather than a traditional quicksort.

"The `sort` algorithm reorders a `List` so that its elements are ascending order according to some ordering relation". Typically, objects to be sorted are made implement `Comparable`, which reflects ordering relations between elements. In the case of link elements, the distinct ordering and nesting relations require the use of two specific `Comparators`[1]: one looks at ordering predecessors from each element, whereas the other is concerned with nesting predecessors.

According to Sun, the Java `sort` is a slightly optimized *merge sort* algorithm. Concretely, it makes the algorithm:

Fast: It is guaranteed to run in `n.log(n)` time, and runs substantially faster on nearly sorted lists. Moreover, "empirical studies showed it to be as fast as a highly optimized quicksort. Quicksort is generally regarded to be faster than merge sort, but isn't *stable*, and doesn't *guarantee* `n.log(n)` performance.

Stable: It does not reorder equal elements. In practice it is important when a list is sorted repeatedly on different attributes, which is not our case.

Besides, the `sort`-based implementation with element predecessors is not less object-oriented than an OO quicksort. Finally, using the algorithm provided by Java prevents from the risk of mistakes inherent to implementing one's own algorithm.

[1] By the way, comparators are claimed to be better than traditional type-unsafe compareTo() methods.

Bibliography

[AOSD.net] *Aspect Oriented Software Development* website http://aosd.net.

[Kiczales *et al.*, 1997a] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin *Aspect-oriented programming*. In *ECOOP'97 11th European Conference in Object-Oriented Programming*, LNCS 1241, pages 220-242, 1997.

[Harrison *et al.*, 2002] W. Harrison, H. Ossher and P. Tarr *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*, IBM Research Report, 2002.

[Filman *et al.*, 2000] R.E. Filman and D.P. Friedman *Aspect-Oriented Programming is Quantification and Obliviousness*, Workshop on Advanced Separation of Concerns OOPSLA 2000, Oct. 2000, Minneapolis, USA.

[Bergmans *et al.*, 2001] L. Bergmans and M. Aksit *Composing Crosscutting Concerns using Composition Filters*, Communications of the ACM, october 2001/vol. 44, no. 10, 2001.

[Tarr *et al.*, 1999] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns* Proceedings of the International Conference on Software Engineering (ICSE'99), May, 1999.

[Kiczales *et al.*, 1997b] G. Kiczales, et al. *D: A Language Framework for Distributed Programming*, technical report, no. SPL-97-010, 1997.

[Irwin *et al.*, 1997] J. Irwin, J.M. Loingtier, J.R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, T. Shpeisman *Aspect-Oriented Programming of Sparse Matrix Code*, International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE), LNCS 1343, Springer-Verlag, 1997.

[Kiczales *et al.*, 1997c] G. Kiczales, et al. *RG: A Case-Study for Aspect-Oriented Programming*, technical report, no. SPL-97-009, 1997.

[Kiczales *et al.*, 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold *An Overview of AspectJ*, in Proceedings of ECOOP, Springer-Verlag, 2001.

[De Volder, 1999] K. De Volder *Aspect-Oriented Logic Meta Programming*, Meta-Level Architectures and Reflection, 2nd International Conference on Reflection, LNCS 1616, Springer-Verlag, 1999.

[Salinas, 2001] P. Salinas *Adding Systemic Crosscutting and Super-Imposition to Composition Filters*, EMOOSE MSc. thesis, Vrije Universiteit Brussel, 2001.

[Rashid, 2001] A. Rashid *A hybrid approach to separation of concerns: The story of SADES*, in Yonezawa and Matsuoka *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, Sept. 2001, Springer-Verlag.

[Tanter & Noyé, 2004] E. Tanter and J. Noyé *Versatile Kernels for Aspect-Oriented Programming*, INRIA Research Report RR-5275, Jul. 2004.

[Wand, 2003] M. Wand *Understanding aspects (extended abstract)*, in *Proceedings of the International Conference on Functional Programming (ICFP 2003)*, ACM Press.

[CME] *The Concern Manipulation Environment* website http://www.research.ibm.com/cme.

[Shonle *et al.*, 2003] M. Shonle, K. Lieberherr and A. Shah *XAspects: An Extensible System for Domain-Specific Aspect Languages*, in *Proceedings of the 18th ACM SIGPLAN Conference on OOPSLA*, pages 26-30, Anaheim, CA, USA, 2003.

[Chiba & Nakagawa, 2004] S. Chiba and K. Nakagawa *Josh: An open Aspect-J-like language*, in AOSD2004.

[Tanter *et al.*, 2001] E. Tanter, N. Bouraqadi and J. Noyé *Reflex - towards an open reflective extension of Java*, in Yonezawa and Matsuoka *Proceedings of the 3rd International Conference on Metalevel Architectures and Advanced Separation of Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, Kyoto, Japan, Sept. 2001, Springer-Verlag.

[Tanter *et al.*, 2003] E. Tanter, J. Noyé, D. Caromel and P. Cointe *Partial behavioral reflection : Spatial and temporal selection of reification* in *Proceedings of the 18th ACM SIGPLAN Conference on OOPSLA*, pages 27-46, Anaheim, CA, USA, 2003.

[Malenfant & Cointe, 1996] J. Malenfant and P. Cointe *Aspect-oriented programming versus reflection: a first draft*, Position Statement for OOPSLA 96.

[Sullivan, 2001] G. Sullivan *Aspect-oriented programming using reflection*, OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems, October 2001, Tampa Bay, FL, USA.

[Douence *et al.*] R. Douence, P. Fradet, and M. Südholt *A framework for the detection and resolution of aspect interactions*, course slides EMN, Irisa.

[Eclipse.org] *Eclipse* website http://www.eclipse.org

[Eclipse] *Notes on the Eclipse Plug-in Architecture* http://www.eclipse.org/articles/Article-Plug-in-architecture/

[EJB] *Enterprise JavaBeans Technology* website http://java.sun.com/products/ejb/

[ANTLR.org] ANTLR website http://www.antlr.org/

[Caromel *et al.*, 2004] D. Caromel, L. Mateu, E. Tanter *Sequential Object Monitors*, Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004), LNCS 3086, pp. 316-340, June 2004, Oslo Norway.

[Harrison *et al.*, 2002] W. Harrison, H. Ossher, P. Tarr, V. Kruskal and F. Tip *CAT: A Toolkit for Assembling Concerns*, IBM Research Report, 2002.

[DAJ.sf.net] *DAJ website* http://daj.sf.net.

[Lieberherr] Karl J. Lieberherr *Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP).*

[YACC] *YACC's Home* http://www.linuxtrent.it/Members/yacc/.

[JavaCC] *JavaCC Home* https://javacc.dev.java.net/.

[algorithms] *Algorithms* http://java.sun.com/docs/books/tutorial/collections/algorithms/.