Vrije Universiteit Brussel - Belgium

Faculty of Sciences

In Collaboration with École des Mines de Nantes - France

and

DCC, University of Chile - Chile

2006

Declarative Composition of Structural Aspects

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: Benoit Kessler

Promoter: Prof. Théo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Éric Tanter (DCC, University of Chile)

Abstract

While designing applications with multiple crosscutting concerns, implemented as aspects, interactions can occurs among these aspects. To resolve these interactions, aspects need to be composed.

Admittedly, a crucial part in aspect composition lies in the *detection* of the interactions. As a matter of fact, aspects typically use *intensional* definitions of their cuts, so the developer can have trouble foreseeing the possible interactions between a given aspect and the base code, and between several aspects. It is therefore crucial that the aspect-oriented system *detects* and *reports* on interactions.

Once the interactions have been detected and identified, they shall be resolved using declarative composition mechanisms. A common example of such mechanism is tools for ordering the aspects at a shared join point.

In this report, we propose a solution for detection of interactions among structural aspects, that is to say aspects modifying the structure of classes, in Reflex. This solution is based on a logic engine connected to Reflex and which is used to reason about the aspects and how they interact. We also propose some advanced tools for interaction resolution.

Acknowledgments

First of all, I would like to thank Prof. Theo D'Hondt for promoting my researches and therefore for allowing me to conduct this thesis.

I also wish to thank my supervisor Éric Tanter for his help and for everything he taught me.

I also thank Raquel, Julio, Patricia and Dieter for their support at all time, and especially during the most stressful parts of this thesis.

Nomenclature

AOP Aspect-Oriented Programming

AOSD Aspect-Oriented Software Development

BLink Behavioral Link

DSL Domain Specific Language

Mutex Mutual Exclusion

SLink Structural Link

SoC Separation of Concerns

VM Virtual Machine

Contents

Chapter 1

Introduction

Aspect-Oriented Programming (AOP) has be designed to achieve separation of concerns in object-oriented languages. Each concern is modularized by an aspect which defines the appropriate behavior independently from the base program (*i.e.* business logic). Yet, at some point, the aspects and the base program have to be weaved together so as to build the final application.

These aspects are designed independently from each other, and possibly by different programmers. Therefore possibilities of interactions and conflict appear, leading to errors or bad or unwanted behavior. To resolve these interactions, aspects need to be composed thanks to declarative composition tools. Composing the aspects allows to resolve the interactions and to ensure proper behavior of the application.

However, in order to resolve the interactions, one must first be aware of their existence. This means that tools should be provided for detection of the interactions. These tools should detect the interactions independently from the base program used and shall report the interactions found the the programmer. Afterwards, the interactions can be resolved by declaring composition rules among the interacting aspects.

In the context of Reflex, a versatile kernel for multi language AOP, the issue of composing structural aspects have been studied. We propose detection tools based on a logic engine connected to Reflex and on detection rules previously defined to detect all kinds of interactions between structural aspects. Yet this study is limited to the scope of structural aspects adding structural elements (*i.e.* classes, methods, fields, annotations, etc.) The aspects which modify the existing elements, such as modifying the class hierarchy or a method access modifier, are left for future study.

We also propose a set of composition mechanisms that one could expect from a comprehensive system truly supporting structural aspects. These mechanisms are sometimes inspired

from existing mechanisms in other domains, such as from traits, while other are original to Reflex, such as the ability to control the visibility of the structural modifications done by structural aspects.

In a first part, we review the state of the art of aspect composition and approaches. We review in particular several well known AOP approaches, such as AspectJ, or approaches which propose advanced mechanisms for interaction detection and/or aspect composition, such as JAsCo and Compose*. We also review in details the Reflex AOP kernel as the background for this study. In a second part, we introduce our proposal for interactions detection and resolution. We first identify and classify the different kinds of interactions possible between structural aspects and list the resolution mechanisms which would be expected for each kind. We then present the composition process used in Reflex. This process is iterative and is based on succession of detections and declarations of composition rules. Afterwards, we describe our proposal for automatic detection of interactions among structural aspects, based on a logic engine. After that, we describe the composition mechanisms introduced in Reflex. Then follows a description of the implementation details concerning the detection tools and the aspect composition process. Then, in a third part, we open discussions about related works and possible future extensions and ameliorations. We also discuss about the value and limits of our proposal. Finally we end with a conclusion on the present proposal.

Part I

Concepts and Context

Chapter 2

State of the art

Composition of aspects is a wide and complex topic and has been studied a lot over the past few years. We here review what has been done so far.

In a first part, we introduce the concept of aspects and the problems caused when several co-exist in a single program. Then in a second part we study some of the different existing systems for AOP so as to build the context of this study.

2.1 Introduction to aspects

In software engineering, *aspect-oriented programming* (AOP) and *aspect-oriented software development* (AOSD) paradigms attempt to aid programmers in the *separation of concerns*, specifically crosscutting concerns, as an advance in modularization. AOP does so using primarily language changes, while AOSD uses a combination of language, environment, and methodology.

Separation of concerns (SoC) is the process of breaking a program into distinct features that overlap in functionality as little as possible. A concern is any piece of interest or focus in a program. Typically, concerns are synonymous with features or behaviors.

In this section we present the different kinds of aspects that exist and introduce the problem of composition of aspects.

2.1.1 Model and types of aspects

AOP languages usually follow the pattern of *pointcut/advice*. Pointcuts allow a programmer to specify join points (well-defined moments in the execution of a program, like method call, object instantiation, or variable access). All pointcuts (*aka* cut) are expressions (quantifications) that determine whether a given join point matches. Advice (*aka* action) allows a programmer to specify code to run at a join point matched by a pointcut. The actions can usually be performed before, after, or around the specified join point. The result is commonly called a *behavioral* aspect as it adds, or modify, some behavior to the base program.

Another kind of aspect exists, usually known as *inter-type declaration*(AspectJ), *structural aspect* or *introductions*. Inter-type declarations allow a programmer to add structural elements (*i.e.* methods, fields, interfaces, annotations, etc.) to existing classes from within the aspect.

2.1.2 Composition of aspects

Aspects can interact when they are applied on a program. Two aspects are said to interact if they apply at a common join point or if one aspect modifies the behavior or the scope of another aspect. Modifying the scope of another aspect can be done by adding new members which are possibly part of its cut. Then, from a semantic point of view, interacting aspects can be conflicting or non-conflicting depending whether the interaction is semantically correct or not.

Using the example of [8]: the `loggingAspect` prints the arguments of designated method calls, while the `encryptionAspect` encrypts and decrypts communications between distant objects. Both aspects apply to the methods `sendData(String)` and `receiveData(String)` which are part of the protocol for communication between distant objects. The two aspects interact in the sense that the data sent between the distant objects are logged and encrypted. Depending on the programmers will, this interaction can be conflicting or not. Let's assume the wanted and correct behavior for the application is to log the encrypted data and not to log the clear data. Then, encrypting the data and then logging it is a non-conflicting interaction. On the contrary, logging the clear data and then encrypting it is a conflict as it is against the programmer's design/will.

Besides, there is a specification dimension: has the interaction been resolved or not? We say that an interaction is resolved if resolution mechanisms have been specified for this interaction, and unresolved otherwise. An interaction can therefore be resolved and conflicting if, by mistake, the opposite order is defined for the resolution. In the above example, this would be specifying that logging should happen before the encryption: the interaction

is resolved but conflicting semantically.

Douence *et al* [7] advocate a general approach for *automatic* detection and *explicit* resolution of aspect interactions. They state that aspects should be written independently from each other, then automatic tools shall detect and report the aspects' interactions to the user and finally the interactions shall be resolved using dedicated composition language. Treatment for the interactions shall also be separated from the aspects themselves.

In the case of behavioral aspects, interactions may occur when several aspect apply at the same join point at the same time. In that case, aspects can be composed using sequencing. This sequencing can be simple, like for instance a simple ordering ("one aspect applies prior to the other"), or more elaborated, using a mix of ordering and nesting ("one aspect surrounds the other") for example.

For structural aspects, the problem is more simple as join points are classes: the aspects have a set of classes which cut their specifications and on which they are to bring structural modifications. A basic analysis is to say that two aspects interact if the intersection of the sets of their classes is not empty. This is not fine-grained enough as modification can be orthogonal and therefore don't create any conflict at all. Sequencing is also more simple as there is no possible nesting: only ordering, using precedences, is possible due to the nature of such aspects.

2.2 Existing systems

Many approaches for AOP exist and it would be impossible to review them all here. Still, we review some of them, the most famous and/or interesting for this report. We specifically put the accent on aspect composition and interactions detection as it is what this contribution is all about.

2.2.1 AspectJ

AspectJ extends the Java programming language with a new construction: aspects. It supports both behavioral and structural aspects, although differently.

Behavioral aspect are known as the *dynamic crosscutting concerns* as they combine a dynamic poincut and an advice to be executed before, after or around the pointcut. The following code gives an example of a package-visible pointcut that exposes an `int` and an advice which runs before reading the field `int Foo.y`, both taken from the AspectJ 5 quick reference guide:

```
pointcut pc(int i) : set(int Foo.x) && args(i) ;

before () : get(int Foo.y) { ... }
```

In the earlier versions of AspectJ, aspect were woven in the base code at compile time. With the version 5 of AspectJ which supports Java 5, it is also possible to define load-time weaving, using a xml file for the configuration.

AspectJ uses precedence (ordering) declarations to deal with conflicts or interactions. Concerning the different advices in a same aspect file, they are applied in the order they are declared within the file. When *around* advices are used, AspectJ forces nesting[1] and *before* and *after* advices are nested inside as well. The default behavior when several aspects apply to the same join point is given in [14]:

1. First, any around advice are run, most-specific first. Within the body of an around advice, calling proceed() invokes the next most specific piece of around advice, or, if no around advice remain, goes to the next step.
2. Then all before advice are run, most-specific first.
3. Then the computation associated with the join point proceeds.
4. Execution of after returning and after throwing advice depends on how the computation in step 3 and prior after returning and after throwing advice terminate.
   - If they terminate normally, all after returning advice are run, least specific first.
   - If they terminate by throwing an exception, all after throwing advice that match the exception are run, least specific first. (This means after throwing advice can handle exceptions thrown by less specific after returning and after throwing advice.)
5. Then all after advice are run, least-specific first.
6. Once all after advice have run, the return value from step 3, if any, is returned to the innermost call to proceed from step 1, and that piece of around advice continues running.
7. When the innermost piece of around advice returns, it returns to the surrounding around advice.
8. When the outermost piece of around advice returns, control continues back from the join point.

[1] Nesting is when an around advice surrounds another advice.

We can see that means for composing behavioral aspects are limited in AspectJ. The developers cannot define their own ordering and nesting when *around* and *before/after* advices are mixed.

AspectJ does not provides any mean of detection of interactions and conflicts. Still, AspectJ on Eclipse (AJDT) provides a view showing the cross references that is to say all the join points where several aspects apply at the same time, or where *around* aspects apply (See Fig. 2.1.)

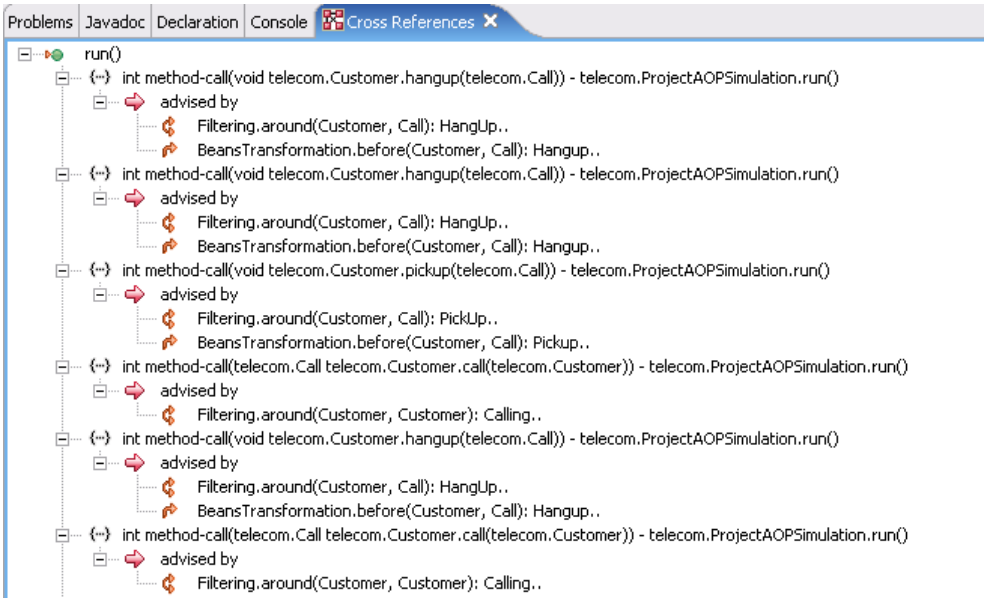

Figure 2.1: Cross References view in Eclipse

This is how interactions are detected and reported by AspectJ.

AspectJ supports structural aspects by means of intertype declarations (*aka* introductions). These are also known as *static crosscutting mechanisms* as it modifies the static structure of classes. It allows for example to add some methods and fields to a class or to modify the hierarchy of classes. The following code example shows how to add a method `m` to the class *Foo*, visible anywhere in the defining package:

```
int Foo.m( int i ) { ... }
```

Problems of composition arise when, for example, two aspects try to add fields or methods with the same signature to a same class, or try to modify the hierarchy of a same class. Still, AspectJ does not provide any means to resolve this conflict. The only solution for the programmer is either to remove one of these methods or to rename one of them. Moreover, intertype declaration do not use pointcuts to define their cut. They instead designated

directly the class where elements shall be added. As a consequence, the modifications brought by an aspect cannot affect the cut of another structural aspect, thus preventing interactions. In fact, AspectJ relies on the compiler and some limited possibilities to detect conflicts among structural aspects.

2.2.2 JAsCo

JAsCo [19, 24] is another aspect-oriented approach, tailored for component-based development. It introduces two concepts: *Aspect Beans* and *Connectors*. The Aspect Beans can be seen as an extension of the actual Java Beans. They describe the crosscutting concerns independently from the actual implementation of the components and define *Hooks* which are responsible for capturing the crosscutting behavior. Hooks contain a triggering condition with both an abstract pointcut definition and a dynamic condition of application thanks to a method `isApplicable`, one or more advice methods and any number of ordinary Java class members, local to the hook. Connectors connect the aspect beans to the components together, by instantiating logically related aspect beans hooks, so as to define behavioral aspects. Figure 2.2 summaries this concept by showing a connector instantiating two hooks of an aspect bean and connecting them to the base code.

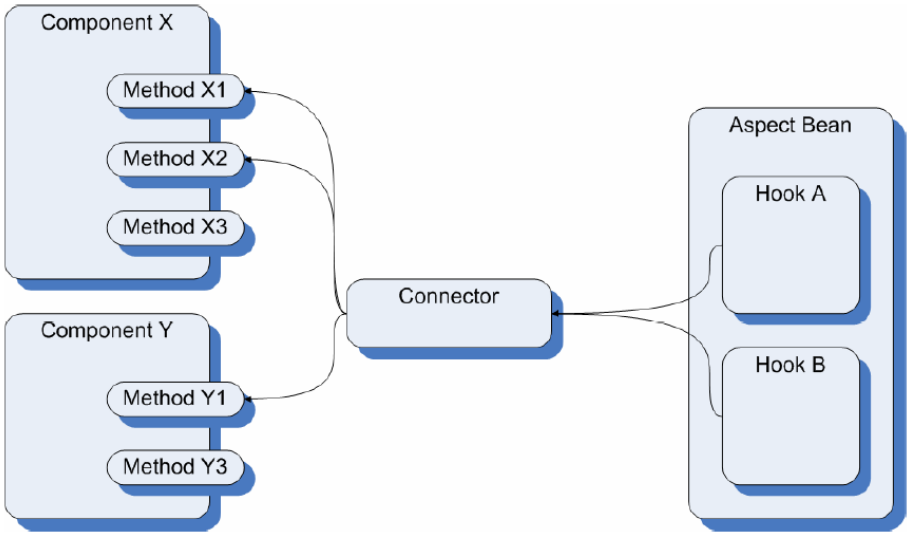

Figure 2.2: Schematic overview of JAsCo

When a connector instantiates several hooks, the default behavior is to apply them in the order they are declared if several are applicable at the same joint point. However, it is possible to define a *precedence strategy* to define a specific order in which associated advices are executed, leading to fine-grained control. Still, this is only available for hooks instantiated within the same connector. If one wants to order hooks which are declared

in different connectors, the only solution is to move these declarations within a single connector. Moreover, JAsCo provides means to define precedences among connectors. There are therefore two distinct levels of ordering: among connectors and among hooks within a same connector. Regarding *around* advices, JAsCo forces a nesting relation, like AspectJ.

JAsCo provides another advanced composition mechanism known as the *Combination Strategies*. This works as a filter on the list of applicable hooks at a certain point of the execution. It allows to define simple composition behavior such as *Mutual Exclusion* (*"If A applies then exclude B"*), also known as "mutex", and *Implicit Cut* (*"If A applies, then apply B as well"*), as well as more complex one. It is then possible to define advanced behavior so as, for instance, to remove `hookB` from the applicable hooks at a certain join point whenever `hookA` is not applicable. The following example defines a combination strategy *TwinCombinationStrategy* which states that if a hook `hookA` is not applicable, then `hookB` shall not be applied either:

```
class TwinCombinationStrategy implements ICombinationStrategy
{
   private Object hookA, hookB;

   TwinCombinationStrategy(Object a,Object b)
   {
      hookA = a;
      hookB = b;
   }

   HookList verifyCombinations(HookList hlist)
   {
      if (!hlist.contains(hookA))
         hlist.remove(hookB);

      return hlist;
   }
}
```

Although this tool is powerful and interesting, it remains dangerous as unexpected behavior may occur. For instance, if a first combination strategy CoS1 defines that if `hookA` and `hookB` are applicable, then `hookC` shall be removed, and if another strategy CoS2 states that if `hookD` is applicable then `hookC` shall be added. Then there is an obvious conflict when `hookA`, `hookB`, and `hookD` are applicable at the same time: if `hookC` is applied, it conflicts with the first rule, while if it does not, it conflicts with the second rule.

JAsCo does not support structural aspects, most probably because they do not fit with the component-based model.

2.2.3 Compose*

Compose* is a aspect-oriented approach based on the concept of *Composition Filters* [2]. Composition Filters model is an extension of the Object-Oriented model whose base idea is to surround every object, referred as the *kernel object*, with an interface layer which intercepts entering and out-going message so that they can be modified. Using this concept, Compose* is able to do an aspect-like language. (See Fig. 2.3)

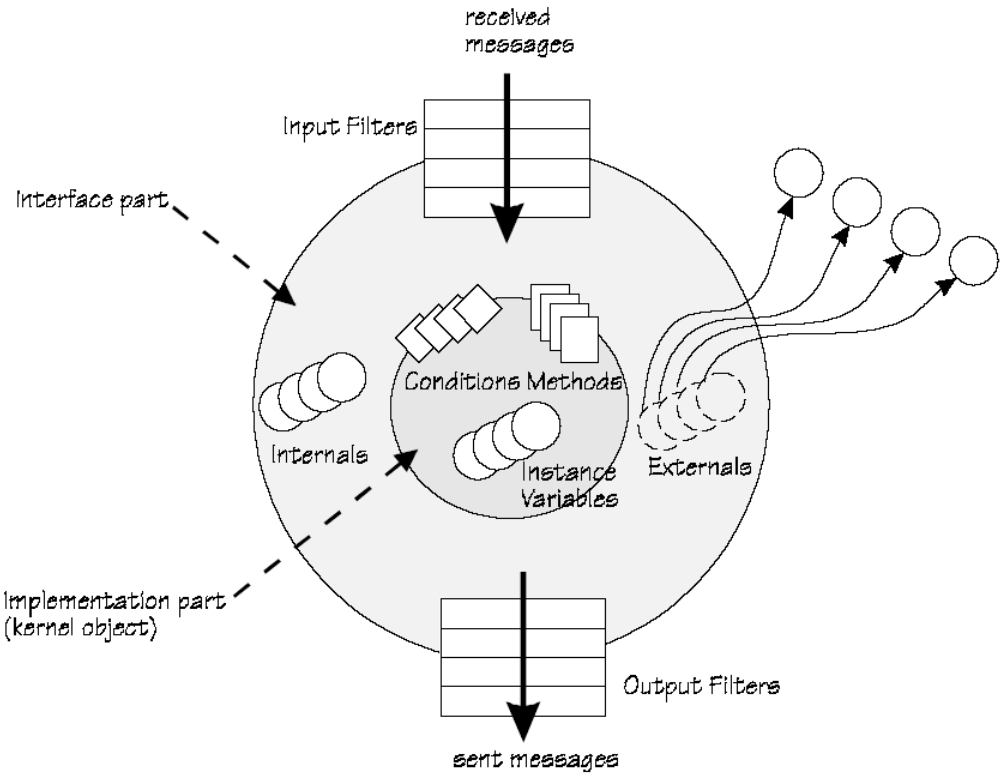

Figure 2.3: The components of the composition-filters model.

Compose* supports both structural (*syntactical*) and behavioral (*semantic*) aspects. Behavioral aspects are done using the composition filter introduced above, while the structural aspects are done by declaring syntactical introductions.

Pascal Durr *et al* [9, 8] propose a language independent technique to detect behavioral conflicts among aspects that are superimposed on the same join point. Their base idea is that some resource must be shared among advices for them to conflict. They are therefore

limiting their study to the -most relevant- case of conflicts on shared join points. They identify *operations* that can be done on *resources* and use *detection rules* to detection conflicts. These rules are created by the programmer and can be an assertion pattern, a combination of operations that must occur on a resource, or as conflict pattern, a combination of operations that must not occur. Then, at a certain join point, they can generate the tree of possible execution traces and select one that is not conflicting with the rules. If this is not possible, they report a warning to the user and continue the compilation process. This system therefore acts both to help the compiler to weave the aspects in a suitable order and as a conflict detection mechanism.

They use as an example the case of two aspects applied on communications between distant objects: one which does logging and one which does encryption and decryption. Both aspect apply on the same join points, for instance when messages are sent and received between the distant objects. There is therefore an obvious interaction which can create unwanted behavior. Indeed, if logging is applied before encryption, the clear message will be logged, enabling debugging but creating an obvious hole in security. In the other case, the message is logged encrypted, which is bad for debugging as the message is no longer available but better in terms of security. Depending on the choice of the developer, rules to detect conflicts can be specified. For instance, the rule "if a read (logging) operation occurs after an encrypt operation on the same resource, then it is considered as a conflict." Then, by specifying multiple rules like this one, the developer can detect semantic conflicts among the aspects he uses.

The limit of this system is that the programmer has to be aware of all the aspects used and that some of them are conflicting in order to think about designing these rules. He for instance has to check manually all the points where the logging can interact with other aspects so as to design the appropriate rules.

Concerning structural aspects, they do not detect the interactions as they state that they are usually captured by the typing system of the underlying language [9]. This assertion is somehow a bit light as when structural aspect have a dynamic cut, they can be influenced by the actions of other aspects, which is not detected by the compiler or the language.

However, in [10] they introduce an algorithm to reason about inter-dependent structural aspects, that is to say aspects whose cut is influenced by the action of another aspect. This can build circular dependencies: (1) an aspect A1 adds the annotation `PersistentClass` to every class containing at least one method with the annotation `Persistent` and (2) aspect A2 adds the annotation `Persistent` to every method of classes which have the annotation `PersistentClass`. Depending on the order of the application and of the class, we can end up with different results. Their algorithm is based on iterations over every possible ordering, but is limited to the case where there are no aspects with what they call *"negative feedback"*, that is to say aspects whose cut is based on the *absence* of elements. In these conditions, the algorithm is sure to reach a fix point, and therefore terminate. For

example, adding an aspect A3 which adds the annotation `Transient` to any method which doesn't have the annotation `Persistent` would be considered as an error and reported as such. However, using aspect whose cut is based on the absence of some elements is still possible as long as it does not interact with other aspects.

2.2.4 LogicAJ

LogicAj stands for *Logic Aspects for Java*. It is being designed at the university of Bonn and uses Prolog as the base elements to modify the Java program [15, 17]. Quoting the website:

> LogicAJ enables the use of meta-variables for base program elements (fields, methods, parameter lists, argument lists, method bodies, etc.). Meta-variables can be used uniformly in pointcut definitions, introductions and advice. Generic introductions and generic advice can both be subject to prior pointcut evaluation. This enables implementation of aspect effects that can vary depending on the values produced for meta-variables during the evaluation of pointcuts.
>
> Generic introductions make it easy to introduce new types, determine new class members, and vary the code of introduced methods by evaluating predicates. Generic advices replace otherwise redundant or reflective advices and reference methods and fields created by generic introductions.

LogicAJ uses *Condor* [5] (for *Conflict Detector*) for interactions detection. To our knowledge, there hasn't been any publication on Condor so far, but on their web site we can read that Condor provides:

- the ability to identify a well-defined class of interferences
- the ability to determine an interference-free order of execution, if one exists
- the ability to determine the most suitable weaving algorithm for a given set of aspects

And the interaction analysis is

- independent of the base programs to which the aspects refer (only aspects need to be analyzed)
- independent of any special annotations of the analyzed aspects

However, no more information is provided, and no download found, so no further investigations have been done on this.

2.2.5 Logic Metaprogramming

Brichau *et al* [3] propose a way to combine aspects written in different aspect languages using *Logic MetaProgramming* (LMP). Aspects and aspect languages are represented as modularized logic metaprograms which are combined using various *combination modules* and/or *interaction modules*.

Combination modules take as parameters several other modules and contain rules that tell how the functionalities of these modules are meant to be combined.

Interaction modules implement a dependency or an interaction between aspects. Contrary to combination modules, they do not combine other modules but instead modularize a crosscutting aspect.

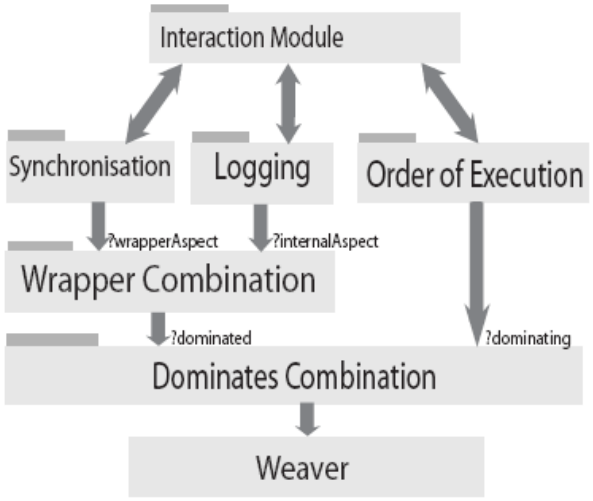

Figure 2.4: Example of composition of logic modules

Figure 2.4, taken from [3], shows an example of this system. Three aspects are represented as modules: *synchronization*, *logging* and *order of execution*. These three aspects interact together, as modularized by the Interaction Module, and are composed as follow: *logging* is wrapped inside *synchronization*, thanks to the Wrapper Combination, and the resulting combination is combined with *order of execution* thanks to the Dominates Combination, which states that *order of execution* is to be applied first. The resulting combination is then given to the weaver.

This LMP approach allows to abstract the aspects in logic modules and then to combine them. It also allows to define complex behavior for the composition by combining modules together. Hence it provides interesting means of composition, especially for aspects from different aspect languages. However, this approach does not address the problem of interaction detection.

2.2.6 Reflex

Reflex is a versatile kernel for multi-language AOP which uses reflection in order to achieve aspects. Its purpose is to be able to support any other AOP approach. Still the study of this approach is postponed to chapter 3 as it will be the base for the rest of the study.

2.3 Summary

A lot of work has been done on composition of behavioral aspects. Some approaches like AspectJ prefer to restrain the possibilities so as to have an easy but efficient system and to avoid complications, while other, like JAsCo, try to come up with advanced solutions to combine and compose aspects. Still, ordering and nesting is usually not very flexible and often limited.

On the other hand, composition of structural aspects seems to have been quite neglected and these aspects are not even always supported.

Concerning detection of conflicts and interactions, it seems very little work has been done so far. Compose* brings some interesting ideas based on specifications provided by the developers to state what is a conflict and what should be happening. However this is not really *automatic* detection as it requires a lot of work from the developer and a clear vision of the entire system, especially of all the different aspects present, which is not always feasible in the case of big teams of development.

Still, they provide a kind of detection mechanism for interactions among structural aspects. This mechanism detects and resolves automatically the circular dependencies among structural aspects, in specific case where there is no negative feedback. However they do not provide tools for the programmers to see the list of all the detected interactions and do not provide them with tools to resolve the interactions manually, as it would be wanted in certain cases.

We now turn to the study of Reflex, which will be the base for the rest of this study.

Chapter 3

Diving inside Reflex

In this chapter we introduce Reflex, a versatile kernel for AOP in Java, as it is the base for the present study. This presentation summarizes the papers [22, 20]. The reader is invited to read them if he wants to learn more about this system. Nevertheless, the necessary information for the rest of the report is presented here.

This presentation is the state of Reflex as it was before the present contribution to the system.

In this chapter, we briefly review this multi-language AOP kernel. In the first section, we describe the idea behind it and how it works. Then we review the aspects of composition in Reflex: aspects of aspects (Sect. 3), aspect dependencies (Sect. 4), ordering/nesting of aspects (Sect. 5), and visibility of structural changes (Sect. 6). Finally, we describe the what happens when a class is loaded in the Java Virtual Machine, as it will be used further in the present document.

3.1 Introduction

Reflex makes it possible to translate aspects from any AOP approach, even domain specific ones, into Reflex [21, 22]. This is done thanks to a model based on *reflection* and *links* to build aspects. Reflex supports both *behavioral* aspects and *structural* aspects and provides means for compositions of aspects. Reflex supports:

- automatic detection of aspect interactions limiting spurious conflicts;
- aspect dependencies, such as implicit cut and mutual exclusion;
- extensible composition operators for ordering and nesting of aspects;

- control over the visibility of structural changes made by aspects;
- aspects of aspects.

3.2 Multi-language AOP and Reflex

This section briefly introduces the necessary background concepts on multi-language AOP and the Reflex AOP kernel.

3.2.1 Multi-language AOP

Reflex is based on a flexible model of partial behavioral reflexion [23] along with structural abilities. It is based on a three level model (cf. Fig. 3.1). The lower level, the transformation layer, is in charge of basic weaving for both behavioral and structural modifications. The second layer, the composition layer, is for detection and resolution of aspects interactions. The language layer is there for modular definition of aspect languages (as plugins).

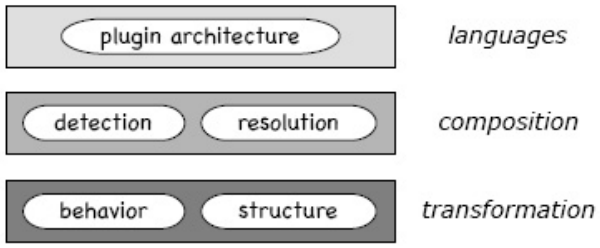

Figure 3.1: Architecture of a versatile kernel for multi-language AOP

3.2.2 Reflex in a nutshell

Reflex's view of AOP is inherently related to metaprogramming: the central notion is about *links* binding a set of program points (a *hookset*) to a *metaobject*. A link is characterized by a number of attributes, among which the control at which metaobjects act (before, after, around), and a dynamically-evaluated activation condition. The cut of aspects are done through *introspection* of the program, while the action is done by behavior/structural modifications (*intercession*). Figure 3.2 shows two links, one subject to activation unlike the other one, and the correspondence to AOP concepts of the pointcut/advice model.

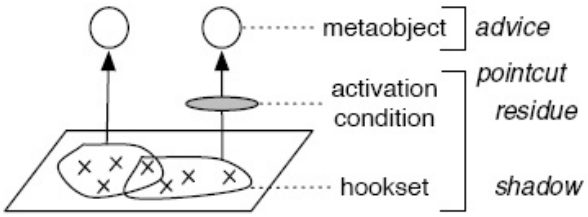

Figure 3.2: The link model and correspondence to AOP concepts

Reflex uses Javassist [4] to perform bytecode transformation at load-time, according to the aspects declared. As one could have figured, there are two kinds of links for the behavioral and structural modifications.

The behavioral links, also refered as BLinks, follow this pattern of hookset with a possible activation test and a metaobject. During their installation, *hooks* are are inserted in class definitions at the appropriate places in order to provoke reification at runtime.

The structural links, or SLinks, on the contrary, are not subject to activation as they are only used at load time. Each structural link binds a structural cut to a metaobject. In Reflex, a structural cut is a *class set* defined intentionally by a *class selector*. For instance the following class selector defines a cut consisting of the `Buffer` class only.

```
bufferSelector = new ClassSelector(){
  boolean accept(RClass aClass){
    return aClass.getName().equals("Buffer");
}};
```

The metaobject bound to the class set can modify the structure of the elements, following the model of Javassist [4]: an `RPool` object gives access to `RClass` objects, which in turn give access to their members as `RMember` objects (either `RField`, `RMethod`, or `RConstructor`), which in turn give access to their bodies as `RExpr` objects (with a specific type for each kind of expression). These objects provide and abstraction of the byte-code and facilitate the manipulations.

Using these links, we can rebuild aspects as they are known in other AOP approaches: an aspect can be represented as a set of structural and behavioral links. The transformation from one side to the other is the matter of the language layer introduced earlier.

3.3 Aspects of Aspects

Designing aspects of aspects is possible in Reflex: A link A can apply to the action of another link B by having the cut of A matching operations that occur in the metaobject associated to B. Metaobjects are indeed normal objects and therefore can be modified as any other object.

The difference is for aspects acting *around* an execution point: in Reflex, the call to the method `proceed` is visible to other aspects, unlike in AspectJ, and therefore can be used in the activation of other aspects.

3.4 Aspect Dependencies

Aspect dependencies are of three kinds in Reflex: implicit cut (*"apply A whenever B applies"*), mutual exclusion (*"never apply A if B applies"*), and forbidden interactions, which is an error mechanism to forbid two aspects to interact.

The implicit cut is defined by providing a link the same cut as another link. In the case of behavioral links, it would be done by

```
BLink trace = Links.getSameCut(discount, <mo>);
```

which is a convenient procedure to declare both links in the same hookset and sharing the same activation condition. (`<mo>` stands for the metaobject specification, not relevant here.)

Mutual exclusion between two aspects is obtained in Reflex by declaring that a link should not apply if another one does. It is defined by the code

```
Rules.declareMutex(aLink1, aLink2);
```

and ensures that if `aLink1` is to be applied, then `aLink2` is not. In the case of BLinks, the resolution of mutual exclusion is subject to the activation condition, if present: if there is no, then the mutual exclusion can be resolved at weaving time. In the other case, resolution is done dynamically at runtime. See [20], section 4.2 for more details on this matter.

A particular case of mutual exclusion is when interaction between two aspects should be considered an *error*. In that case, it simply raises an error at the attention of the developer whenever both links are applied both. The following code is used to declare a forbidden interaction between `discount` and `bingo`:

```
Rules.declareError(discount, bingo);
```

3.5 Ordering and Nesting of Aspects

The Reflex AOP kernel follows the general approach advocated by Douence *et al.*, of *automatic* detection and *explicit* resolution of aspect interactions [7]:

- The kernel ensures that interactions are detected, and reported to users upon under-specification (Sect. 3.5.1).
- The kernel provides expressive and extensible means to specify the resolution of aspect interactions (Sect. 3.5.2).
- From such specifications, it composes links appropriately (Sect. 3.5.3).

3.5.1 Interaction Detection

Reflex adopts different definitions of interactions depending on the case: behavioral and structural.

Two behavioral links interact *statically* if the intersection of their hookset is not empty. As the cut of a link is subject to a dynamic-evaluated condition, it is said that they interact *dynamically* if they interact statically and they are both active at the same time. But since link ordering is resolved statically (when introducing hooks) and activation conditions can be changed dynamically, Reflex adopts a defensive approach: any static interaction is reported, and must be considered by the developer, so that a dynamic interaction is never under-specified [20]. This approach has the advantage to limit the spurious conflicts.

Two structural links are said to interact if the intersection of their class set is not empty. This approach is too coasrse-grained as two links can bring orthogonal modifications and therefore not create any conflict. This approach shall be reviewed, which is part of the present work.

3.5.2 Ordering and Nesting

Reflex provides two basic sequencing: ordering and nesting. Ordering applies two elements one before the other. Nesting applies two elements one "inside" the other. This is exclusively the case for *around* advices: the nested action will be executed only if the previous one calls the method `proceed`.

Using these two basic operators, Reflex user can build higher level operators, such as *wrapping* and *sequence* (cf. Fig. 3.3). The implementation details of such operators is not to be discussed here [20].

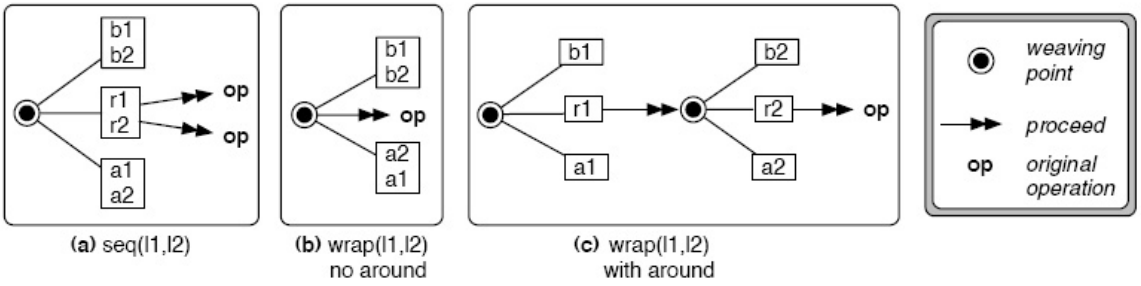

Figure 3.3: Ordering and nesting scenarios

An example of composition of behavioral links is given as follow:

```
BLink timer = ...; BLink synchro = ...;
Rules.declare(new Wrap(timer, synchro));
```

The declared composition implies that the timing aspect measures execution time of methods, including the synchronization cost.

Structural links are treated separately as they cannot be nested. These links are applied in sequence and therefore only ordering is applicable. They are composed using the code:

```
Rules.declarePrecedence(persistency, history);
```

This example states that the link `persistency` has to be applied before the link `history`.

Behavioral and structural links are composed and applied separately and therefore do not need to be ordered in any way.

3.5.3 Hook Generation

When detecting links interactions, Reflex generates a hook skeleton based on the specified composition rules, similarly to Fig. 3.3. The hook skeleton is then used for driving the hook generation process: taking into account how links elements have to be inserted, with the appropriate calls to metaobjects. In order to support nesting of aspects with `proceed`, Reflex adopts a strategy similar to that of AspectJ described in [11], based on the generation of closures.

3.6 Visibility of Structural Changes

As explained ealier, both behavioral and structural links rely on introspection of the structure of a program to define their cut. Yet, SLinks modify this structure, so the question of the visibility of these modifications naturally arise.

Let's consider two links: a SLink `history` adds history to fields while a BLink `persistency` makes fields persistent by monitoring the fields accesses. The issue of whether the field added by the first link in order to record history should be made persistent appears. In fact, `persistency` can only save the state of the fields added by `history` if it actually "sees" them. In some cases, saving this state is relevant while in other it is not, depending on the program and the intentions. Therefore there should be a choice for whether or not added elements are visible to other links.

As said in [20], by default, every structural change is invisible to the cut of other links, thus avoiding unwanted interactions.[1] Still, it is possible to define the *view* of each link so as to *see* the changes made by other SLinks and to take it into account.

The code

```
Rules.augmentViewOf(persistency, history);
```

found in [20] declares that all changes made by `history` are visible to `persistency`, while

```
Rules.addToDefaultView(history);
```

says that all the changes made by `history` are part of the default view, and therefore that they are visible to every other link.

3.7 Structural Links application

In Reflex, structural links are applied when a class is being loaded in the JVM. As shown in Figure 3.4, the structural and behavioral links are applied in different phases: first the structural links, then the behavioral ones.

As for now, the structural links application is done as follow: When the class is being loaded, it enters the Structural Link Application (SLA) phase. The system builds a list of all the structural links present. Each link is being tested on the class to see whether or not

[1] This is possible in the case of additions. The case of modifications is not dealt with in this report.

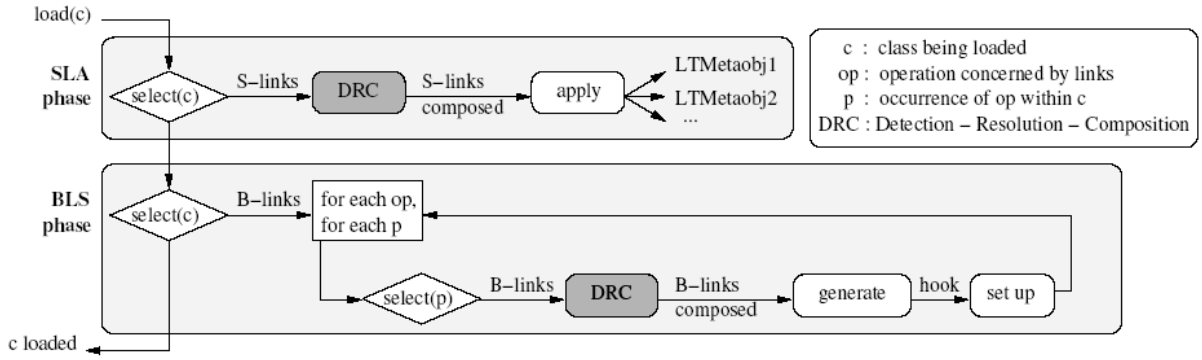

Figure 3.4: Loading of a class in Reflex AOP Kernel

this class is part of its cut. That is the class is given to every link's class selector which will then return a *Yes* or *No*. This is corresponds to the introspection, and to the *select(c)* on the diagram. The links which have this class as part of their cut are kept in a list. Then mutual exclusion rules are applied to this list in order to exclude links. The final list of remaining links is ordered thanks to the ordering rules defined. If no order is defined, SLinks are applied in a arbitrary order. Finally the links are applied to the class, that is to say that their action is performed. This corresponds to the intercession and to the *apply* on the diagram. Afterwards, the class exits the SLA phase and enters the Behavioral Link Set-up (BLS) phase for BLinks application, which will not be explained here.

3.8 Summary

Reflex is a kernel for AOP supporting many languages. It provides a lot of possibilities, especially in terms of composition as it provides fine-grained ordering and nesting. It also tries to supply tools for detection of interactions and conflicts. As for now, these tools are limited, especially in the case of SLinks. Indeed, for SLinks, composition process is very basic and there is hardly any existent interaction detection tools.

In the present contribution, we propose solutions for detection of interactions in the case of SLinks as well as a more advanced and flexible composition process.

Part II

Contributions to Declarative Composition of Aspects in Reflex

Chapter 4

Possible Interactions and their Resolution

In this chapter, we propose a classification of structural interactions, and list the possible resolutions that one would expect from a comprehensive system truly supporting structural aspects [1]. We distinguish conflicts occurring from the interaction of an aspect with the original base code, conflicts between actions of (at least) two aspects, and dependencies between the action of an aspect and the cut of another one. For each category, we give a general description of the syndrome, a few examples, and a list of desired possibilities of resolution. Note that every suggested possibility is a priori always applicable.

In the following, a *structural element* denotes any piece of structure in an OO program, *i.e.* a class, interface, annotation, field, method or constructor. The actions we consider are the *addition* of structural elements to the base program, *e.g.* a new class, a new method to a class, or a new annotation to a field. A *structural container* is an element containing other structural elements; for instance, the VM is a structural container of classes, a class is a structural container of members, and a member is a structural container of its annotations.

[1] This analysis has been published in ECOOP ADI Workshop [13]

4.1 Interactions with base code

Syndrome	An aspect adds a structural element which is already present in the base code.
Examples	Add class `C` but `C` already exists. Add method `m` to class `C` which already has this method (either directly or via inheritance).
Treatments	Skip the action. Combine element to add with existing one (*). Modify the aspect to avoid the clash.

(*) By *combining*, we are referring to a mechanism similar to what is proposed in composition of traits [18]: explicit aliasing of conflicting members and definition of a combination based on the aliased members.

This kind of interactions are reported as conflicts by the compiler as the code cannot be compiled anymore.

The presented treatments will be introduced later in this chapter.

4.2 Interactions between actions

Syndrome	Two aspects add an element with the same signature in the same structural container.
Examples	Aspects `A1` and `A2` add a class `C`. Aspects `A1` and `A2` add a method `m` to class `C` (either directly or via inheritance).
Treatments	Skip one or both of the actions. Combine both elements to add in a single one (*). Modify one or both aspects to avoid the clash.

Same as for interactions with the base code, such interactions are conflicts which prevent the code from compiling.

4.3 Interactions action-cut

Syndrome	An aspect adds an element which belongs to the intensional cut of another aspect.
Examples	Aspect `A1` adds a class `C` to package `p`, and aspect `A2` adds a method `m` to all classes of `p`. Aspect `A1` adds an annotation to all fields of class `C`, and aspect `A2` adds a field to class `C`.
Treatments	Make added element visible or not to (the cut of) other aspects. Control order of application of aspects. Declare mutual exclusion.

These interactions are treacherous as they do not end up in conflicts and are not reported by the compiler. They can bring unwanted and unexpected behavior and therefore should be detected and reported somehow.

4.4 Classification of interactions in Reflex

The present study is limited to the case of structural additions. Modifications like changing the methods modifiers, from `private` to `protected` for instance, or changing the hierarchy of a class, are left for further study. However, this topic is approached briefly in the discussion part, section 9.3.

Yet, additions already provide a lot of possible interactions due to the action-cut model. In this section, we describe the hierarchy of interactions that have been identified and which are detected in Reflex. We also give a terminology for the interactions which will be used in the rest of this report.

4.4.1 Types of interactions

Interactions between structural aspects can be of several kinds. In this chapter, we have identified three categories of interactions: with base code, between actions, action-cut. The first two end up with compilation errors while the third one has to be detected to be noticed. Hence, only action-cut interactions need further work for detection and reporting.

Action-cut interactions involve the cut of one SLink and the action of another one. The cut of SLinks is represented by the *class selector*, as introduced in section 3.2.2. A class selectors can base its decision on any introspectable characteristics of a reified class object, down to the constituents of method bodies (expressions in a method body are reified

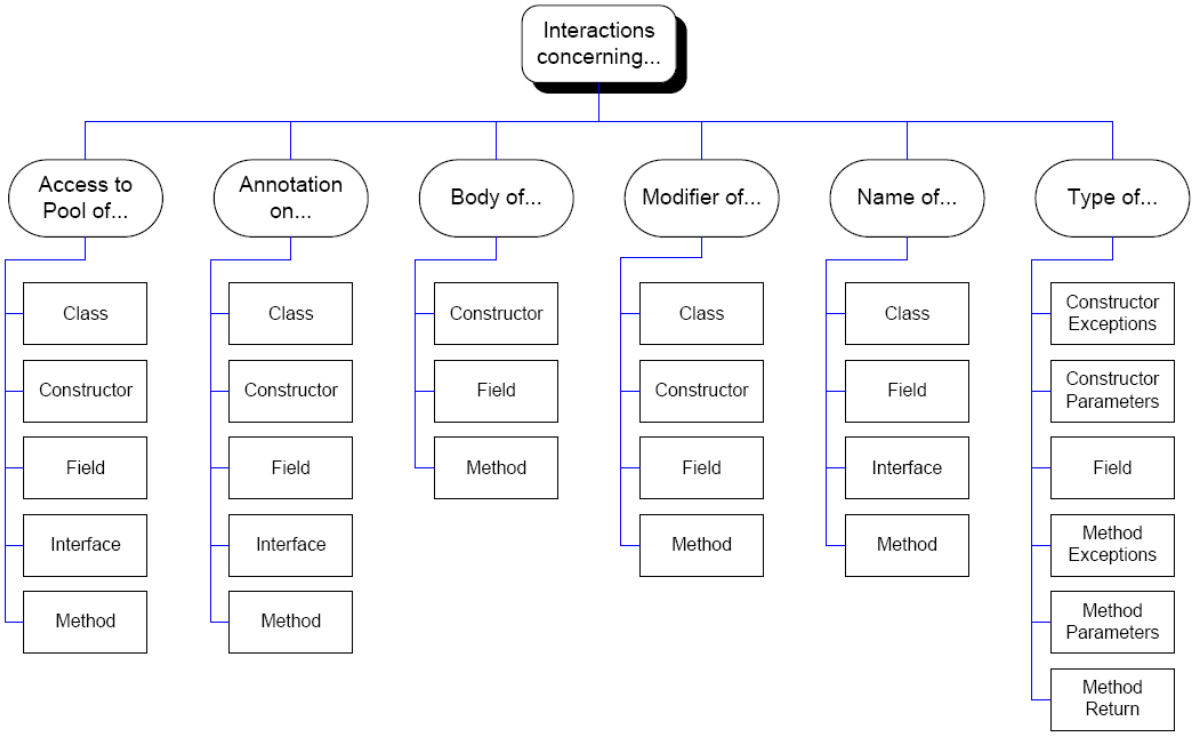

Figure 4.1: Simplified hierarchy of interactions

if needed) [22]. Besides, any kind of structural element can be added, from classes to fields, from annotations to interfaces, etc. Consequently, interactions can happen on any characteristic of structural elements.

Figure 4.1 shows a classifications of the possible points where interactions can happen in Reflex model. The rounded rectangles show the main characteristic of the interaction, what it is about, while the rectangles below show the points where this interaction happen. So for instance, the upper rectangle of the column "Annotation on..." represents the interaction involving the presence of a specific annotation on a class. This interaction happen between a SLink whose class selector accepts classes which have a specific annotation while another SLink adds this annotation on classes. Another example, the rectangle "method" under the rounded-rectangle "Modifier of" designated the interactions concerning the access modifiers of methods. So for instance a SLink's class selector accepts classes which have a `public static` method while another aspect adds a `public static` method.

Of course, as class selectors are not restricted to only one characteristic, interactions can involve several of these points. For instance a class selector can accept classes which implement the interface `Point2D`, which have the annotation `Persistent` and which have at least on method which throws the exception `InvalidPointException`.

4.4.2 Terminology

We have introduced earlier, in chapter 2, a first terminology for interactions. It states that two aspects are said to interact if they apply at a common join point or if one aspect modifies the behavior or the scope of another aspect. Then there are two dimensions: (1) is the interaction correct in a semantic point of view, and (2) has resolution mechanisms been declared for the interaction? For (1) we say that an interaction is non-conflicting if it is correct semantically, and conflicting otherwise. For (2) we say that an interaction is resolved if resolution mechanisms have been declared for it, unresolved otherwise.

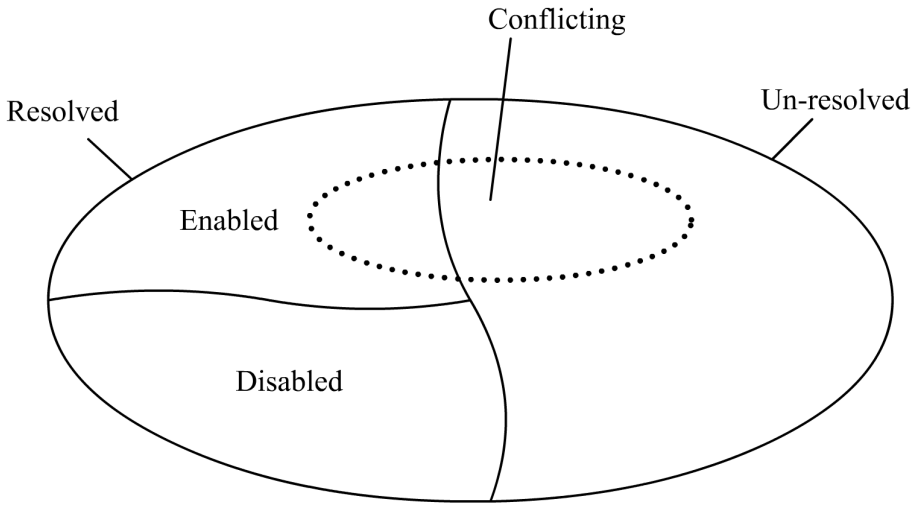

Figure 4.2: Set of possible interactions between structural links

Regarding the new classification of interactions presented in this chapter, it appears that this terminology is not sufficient for action-cut interactions (recall section 4.3). We here define a terminology for structural aspects interactions. Figure 4.2 introduces two new terms in the specific case of action-cut interactions: if the action is placed before the cut, the interaction is said to be *enabled* because the action of the first aspect *does modify* the scope and/or behavior of the second one (cut), while if the cut is placed before the action, the interaction is said to be *disabled* because the second aspect (cut) *is not affected* by the action of the first aspect (action).

Note that the figure 4.2 is only valid for the classification of interactions among structural aspects. For example, between structural aspects in Reflex, an interaction cannot be unresolved and enabled. The reason is that to enable an interaction, one must define rules of ordering for it and therefore the interaction is resolved. This is because of Reflex default semantics that avoids unwanted interactions. However, when involving structural and behavioral aspects together, an interaction can be unresolved and still enabled because behavioral aspects are set up after structural ones are installed (see section 3.7).

Moreover, we say an interaction is effective when the behavior and/or scope of one aspect is modified by another aspect, whereas it is said non-effective if it is not effective but could become effective with composition rules. For instance, the visibility mechanism introduced in section 3.6 hides by default the modifications made by aspects. Therefore, if a first aspect `L1` adds an element which is part of the cut of a second aspect `L2`, `L2` does not see the modifications and thus is not affected: we say that the interaction is non-effective. However, by allowing `L2` to see the modifications made by `L1`, this interaction become effective as the scope of `L2` is modified by `L1`.

So, the visibility of structural modifications controls the effectiveness of the action-cut interactions, while ordering enables or disables them. Still, a non-effective interaction cannot be enabled or disable as it does not really exist.

This terminology will be used for the rest of this document.

4.5 Advanced resolution mechanisms in Reflex

For interactions with base code and between action (see sections 4.1 and 4.2), we have raised the possibilities of skipping an action or combining two methods with the same signature. Skipping an action simply removes the conflict by not adding the conflicting element and therefore keeping the previous implementation of the concerned element. Combining two methods allows to keep and use both behaviors in a single method, which is a combination of the two conflicting ones.

An additional combination method is provided, more as a convenience method, and allows to exclude a class from the class set of a SLink.

Concerning action-cut interactions, resolution mechanisms are base on the control of the visibility of the modifications, on the order of application of the SLinks and on aspect dependencies to declare mutual exclusions.

We here review these mechanisms in more details.

4.5.1 Interactions with base code or between actions

The resolution mechanisms used to cope with interactions with base code or between actions are of three kinds: skipping an action, combining two methods or excluding a class from the cut of a SLink. We here review these three mechanisms.

Class exclusion

Class exclusion can be declared between a SLink an a set of classes (represented by a class selector). It enables to prevent these classes from being part of the cut of the SLink and therefore prevents this SLink from applying any structural modification onto these classes.

To do so, one must use the code

```
Rules.declareExclusion(aLink, aClassSelector);
```

where `aLink` is the SLink concerned by this rule, and `aClassSelector` is a class selector defining a set of classes.

Skipping an action

Skipping an action allows to remove a conflict when for instance an aspect tries to add a structural element which already exists in the class (either from base code or previously added by another SLink). This is done simply by not doing the conflicting action.

Still, there can be two different level of granularity for the term "action". One can either skip the entire action (advice), which is the same as not applying the modifications to the class at all, or skip only the part of the action which is conflicting: the addition of the conflicting element.

Skipping the entire action can be done by using

```
Rules.declareEntireActionSkip(aLink, aClassSelector);
```

which prevents the SLink `aLink` from applying any structural modification to the classes accepted by the class selector `aClassSelector`. It can also be done manually by defining a class exclusion between the link `aLink` and the set of classes defined by the class selector `aClassSelector`. As a matter of fact, skipping the entire action is implemented using the class exclusion mechanism.

Skipping only a part of the action allows to skip only the conflicting addition while keeping the other modifications. It can be declared using

```
Rules.declareConflictingActionSkip(aLink);
```

to define that the addition of conflicting elements added by `aLink` should always be skipped and

```
Rules.declareConflictingActionSkip(aLink, aClass);
```

to define that it should only be skipped on a specific class and conflicts with the other classes shall still be reported. The last rules allows to have a very fine-grained conflict resolution as it allows to resolve the conflict using the action skip in some case while defining other mechanisms for the other cases.

Method combination

To solve the conflicts caused by the addition of an already existing method, another solution has been proposed which consist of the combination of the existing method and the one which is about to be added. These two methods are conflicting because they have a common signature and belong to the same class, which is not tolerated by Java.

Method combination is inspired from the mechanism in traits [18] which allows to combine two methods added using traits into one. The two former methods are aliased and a new method is added, which combines the two aliased methods.

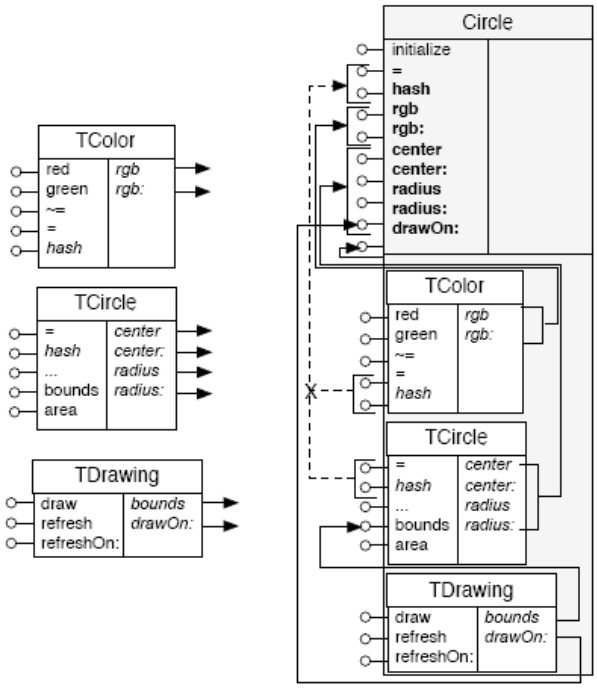

Figure 4.3: Example of a class Circle built using three traits

Figure 4.3 shows a example of use of traits to build a class `Circle`. This class is composed by the three traits `TColor`, `TCircle` and `TDrawing`. `TColor` and `TCirlcle` provide the

methods `hash` and `=` therefore creating a conflict. These methods are combined into one in the class `Circle`: the class defines a method `hash` which uses the two previous methods aliased as `tcolorHash` and `tcircleHash`. Same goes for `=`.

In our case, suppose class `C` has a method `String getName()`. During introspection, a SLink `SL` tries to add a method with the same signature. This is reported as a conflict as it bring compilation errors. To overcome this issue, one can declare a combination of these methods. Then, the existing method is automatically aliased to `getName_Old_9820639816()`, which is a unique generated and non existing name, and the new method is added under the alias of `getName_New_8731378()`. A new method `String getName()` is then created: it defines a combination of the two previously aliased methods. An example of combination is given here:

```
public String getName()
{
   return "My names are " +
        getName_Old_9820639816() +
        " and " +
        getName_New_8731378();
}
```

The body of the combination method is provided under the form of a `String`. The two aliased methods can be referenced using the text `$aliased_existing_method$` for the method already present in the code and `$aliased_new_method$` for the one which is to be added. These references will be replaced in the body by the actual names of the aliased methods.

The methods

```
declareMethodCombination(SLink aLink,
                         RClass aReturnClass,
                         String aMethodName,
                         RClass[] aParameters,
                         RClass[] aExceptions,
                         String aBody);

declareMethodCombination(SLink aLink1, SLink aLink2,
                         RClass aReturnClass,
                         String aMethodName,
                         RClass[] aParameters,
                         RClass[] aExceptions,
                         String aBody);
```

allows to declare a combination for the method with the given signature (return type, method name and parameters), with the body given by `String aBody`. The first method is used in the case of interaction with base code and defines the SLink involved, while the second one is used in case of interaction between actions thus providing the two interacting SLinks.

For some reasons, mostly technical, it is not the conflicting method which is given in parameters as a `RMethod` but only what defines its signature. As a matter of fact, it is not always possible to give a reference to the method which is involved in the conflict and which is to be added as this method may be generated during the action. Providing the existing method involved in the conflict is not a good choice as (1) it leaves the conflict between actions unsolved, (2) it forces the programmer to declare such rule for each class where this method is conflicting. Suppose the method `String getName()` appears in several classes and the link `SL` adds a generated method `String getName()` in several of them, then the programmer would have to declare a rule for each case. It appears then that the most flexible solution is to provide the elements which define the signature of the methods as this solution works in all cases.

An example of code used to declare the previous combination in case of interaction with base code is given as follow:

```
Slink theLink = = addSLink(...);

String theBody = "return \"My names are \"" +
                 "$aliased_new_method$()" +
                 "\" and \"" +
                 "$aliased_existing_method$();";

Rules.declareMethodCombination(
         theLink,
         API.getClassPool().get("java.lang.String"),
         "getName",
         new RClass[] {},
         new RClass[] {},
         theBody);
```

4.5.2 Action-cut interactions

Two cope with action-cut interactions, we can either control the visibility of the structural modifications, control the order of application of the SLinks or declare mutual exclusions. We here review how these mechanisms help to resolve interactions.

Visibility

The ability to control the visibility of structural elements has been introduced in section 3.6. It allows to show or hide the structural modifications made by the SLink to the cut of other links. It then appears that hidding these modifications prevents action-cut interactions.

Ordering

Action-cut interactions happen between two SLinks when the action of a SLink `L1` adds an elements which is part of the cut of a SLink `L2`, thus modifying its behavior and/or scope.

The point of ordering as a resolution mechanism is that if `L1` is applied before `L2`, then `L2` can see the modifications done (assuming sufficient visibility) and therefore react upon it: the interaction is enabled. On the contrary, if `L2` is applied after `L1`, then `L1` did not see the modifications made by `L2` and therefore is not affected by it: the interaction is disabled.

Hence, ordering is used to resolve action-cut interactions as it can enable or disable them.

Aspect dependencies

As said before, action-cut interactions happen between two SLinks. The idea of using aspect dependencies to solve these interactions is that if these two links are in mutual exclusion, they cannot interact. Indeed, in that case, the two SLinks are never both applicable on a common class and so the modifications of one SLink cannot affect the cut of the other. Aspect dependencies can then be used to prevent action-cut interactions.

4.6 Summary

In this chapter we have studied the possible interactions between structural aspects and discussed possible resolution mechanisms that would be applicable. We have seen that apart from ordering and visibility we would like to use two other mechanisms: the possibility to skip the conflicting actions purely and simply by skipping the conflicting actions, and the possibility to replace the conflicting methods by a new one which combines both behaviors.

Now that we have identified all possible interactions and some resolution mechanisms to deal with them, we wish to know how the composition process works for structural links. This matter is to be discussed in the next chapter.

Chapter 5

An iterative composition process

In the previous chapter we have defined what interactions between structural aspects are and ways to resolve them. But in order to resolve the interactions, one must first be aware of their existence and know where they happen. We therefore require a mechanism to detect and report the interactions between links.

As introduced earlier, interactions with base code (section 4.1) and between action (section 4.2) terminate in compilation errors and therefore are reported by the compiler itself; only action-cut interactions (section 4.3) need further consideration.

In our proposal, the detection mechanism is integrated in a wider process for links composition. This process aims at presenting the detected interactions according to the specifications of the programmer. The programmer can specify rules of ordering, visibility, dependencies (*e.g.* mutual exclusion), etc., in the line of what is introduced so far. Then, by rerunning the detection mechanism, he can see the effect of these rules and therefore modify them or add some more if needed.

We first present the global view of this process and then review all the parts in more details in further sections. We also have a word on the reporting tools in the last section.

5.1 Global view of the process

In [7], Douence *et al* introduce a process for aspect composition. We reuse the idea and build an iterative process out of it: interactions are detected during a *conflict analysis* and reported to the programmer who then has to resolve them using composition tools and who can then see the effects by returning to the conflict analysis phase. The Reflex composition process is based on the same idea. Figure 5.1 shows a global view of the

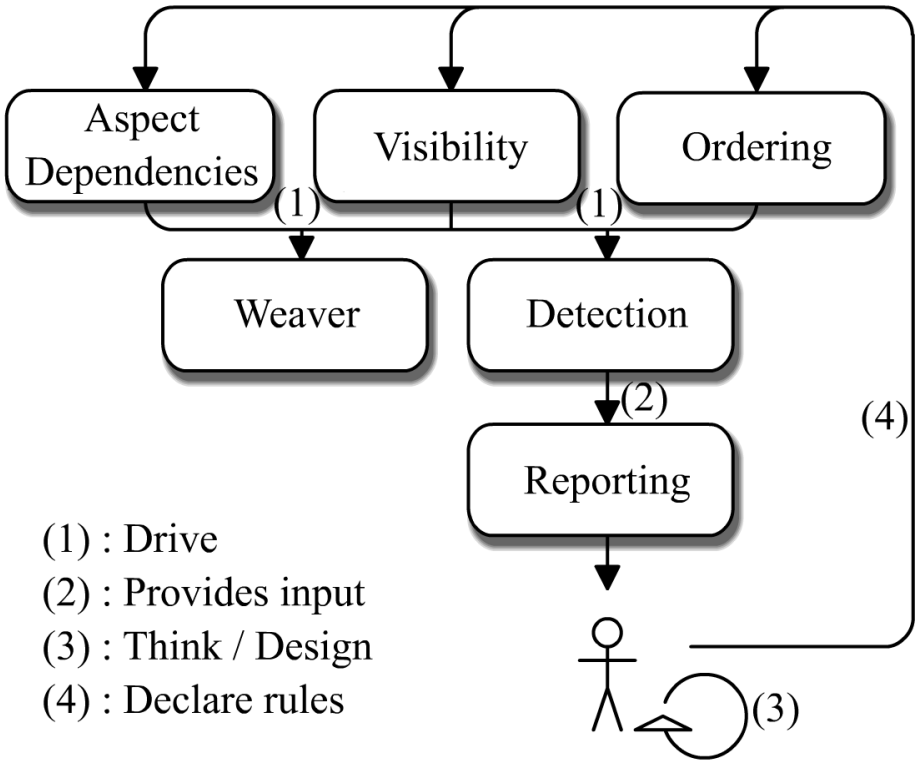

Figure 5.1: An iterative composition process

process. Composition tools (top of the figure) have an influence on the interactions and therefore on the detection tools which provide data to the reporting tools which presents the found interactions to the programmer (bottom of the figure). The latter can then define new composition rules, or modify the existing ones, and see their effects over a new iteration of the process.

The composition rules defined have an impact on the interactions and thus on the detection: an interaction which has been resolved shall still be reported though differently than the unresolved ones. How this is presented is the matter of the reporting tools which are not in the scope of the present study.

The effects of the three composition tools, namely *visibility*, *ordering* and *aspects dependencies*, on the detection tools are studied in the following sections.

5.2 Visibility

Section 3.6 introduces the concept of visibility of the structural modifications brought by the structural links (SLinks). The point is that the visibility of these modifications should be controllable so as to prevent unwanted interactions. By default, these modifications are invisible to other links thus preventing any interaction. This visibility has to be extended over the default one (see section 3.6 to see how) for SLinks to interact.

It appears easily that visibility rules do affect the interactions detection tools as it can make any action-cut interaction non-effective by preventing the modifications of the action to affect the cut of other links. On the contrary, extending the visibility to other links does not necessarily enable an interaction: ordering is still needed to enable it.

Finally, detection tools are able to detect non-effecitve interactions, that is to say interactions that lack sufficient visibility to be effective. For instance SLink `L1` adds the annotation `Persistent` to some classes while the cut of `L2` contains classes with the annotation `Persistent`. By default, the annotations added by `L1` are invisible to the cut of `L2` and so no interaction exists between these two SLinks. However, if the visibility of the modifications done by `L1` was extended to `L2`, then an interaction would be detected as it would be effective. These interactions are still detected and reported as any interactions, but marked as lacking visibility.

5.3 Ordering

Over the iterative process, action-cut interactions are detected and resolved by the programmer. One way to resolve this kind of interactions is the ordering rules. The idea behind that is that if the cut is done before the action, then the interaction is disabled. On the contrary, if the action is done before the cut, then the interaction is enabled and so the behavior and/or the scope of the second aspect, the one whose cut is involved, are modified.

Once the programmer has defined an ordering rule for one interaction, it means that the interaction is resolved and so the warning message for the interaction has to be different. The goal is to make a clear difference between the interaction that are still to be coped with and those that have already been. To do this, the detection mechanism needs to make a difference between these two situations and so be aware of the ordering rules defined.

5.4 Aspects Dependencies

Aspect dependency rules (see section 3.4) act like a filter which adds/removes links from the set of links which should normally be applied on a class. Their effect on the detection is not the same whether they add or remove links. Normally, what happens on each class is:

Mutual exception. In a system which reasons on the cuts and the actions of links so as to find possible interactions, problems occur if a link first introspects a class and is removed by mutual exclusion afterwards. The reason is that the introspection of a SLink over a class has already given the detection mechanism some information about the cut of the SLink and therefore interactions can be detected although the mutual exclusion prevents it. The detection mechanism thus needs to be aware of exclusions so as to report only relevant interactions.

Implicit cut. As for now, implicit cut is done by defining that two links have the same cut. So the detection is not affected as all phases (introspection and intercession) are done normally.

Interaction Selectors. More generally, Reflex provides ways to create advanced composition rules the same way JAsCo provides the Combination Strategies. Mutual Exception and Implicit Cut can be programmed using these *interaction selectors* and based on these two examples we can conjecture that the only case that arise problems is when a SLink is removed from the set of selected SLinks because interactions are still reported although they should not. The detection mechanism shall therefore be aware of all exclusions.

5.5 The reporting tools

Once the interactions are detected, they have to be reported to the programmer so that he can deal with them. These tools are out of the scope of the present study but have to be built on top of it and so we provide the necessary functions for an easy implementation.

Ideally, these tools would be a graphical interface which allows the programmer to browse and sort the found interactions and which provides as much information about the interactions as possible, so that the programmer can resolve them easily. If he finds that an interaction is spurious or if he simply does not want to deal with it, he can tell the reporting tools to ignore this interaction. To do so, he simply have to declare an ignore rule between the two concerned SLinks as follow:

```
Rules.declareIgnore(aLink1, aLink2);
```

5.6 Summary

An iterative process enables the programmer to detect and resolved interactions. These interactions are detected by automatic detection tools and then reported to the programmer thanks to reporting tools. The programmer can then define composition rules to resolve the interactions and see their effect by querying again the detection tools. This iterative process thus helps the programmer to compose interacting aspects by providing better visibility on the interactions and on the effects of the composition rules.

In the next chapter, we describe the detection tools and see how we can use a logic engine to detect the interactions.

Chapter 6

Automatic Detection of Structural Interactions

Admittedly, a crucial part in aspect composition lies in the *detection* of the interactions [7]. As a matter of fact, aspects typically use *intensional* definitions of their cuts, so the developer can have trouble foreseeing the possible interactions between a given aspect and the base code, and between several aspects. It is therefore crucial that the AO system *detects* and *reports* on interactions.

Some interactions (with base code and between actions in our classification) are easy to detect since they result in compilation errors (*e.g.* a class with two methods of the same signature). AspectJ for instance reports on this class of conflicts, since the compilation process cannot go any further.

More subtle is the case of action-cut interactions, because there is no compilation error implied, nor are the dependencies between aspects easy to see – in particular if the cut and action languages are Turing-complete. In order to detect these interactions, we must be able to reason from two sets of pieces of information: which aspects *look at* which structural elements for their cut and action? which aspects *change* which structural elements as a result of their action?

6.1 A logic-based approach

Our approach consist in the use of a *logic engine* connected to Reflex. This is feasible and interesting to do, as explained in [6], because this part of the system is strongly logic-oriented.

Moreover we could have programmed the detection tools in plain Java, as the rest of Reflex is, but it seemed to us that it would be so much simple and easier to extends with new rules with a Prolog interface. Indeed, the actual rules only take into account the additions of structural elements and not modifications, which have been left for further consideration. Extending the actual system of rules to this problem is quite simple as it is just few rules to add or modify. In Java, such modification, and therefore the maintenance and extension of the mechanism, would have been much more difficult.

Finally the current implementation of detection and resolution of interactions between behavioral aspects in Reflex is purely Java-based, and the obstacles encountered strongly motivated this change of implementation approach.

6.1.1 Fact generation

In order to reason about the facts concerning the added elements and the observed ones, we need to generate them. This generation happens at several level of the system:

- Upon introspection, entities from our structural model (classes, methods, fields, etc.) generate logic facts indicating that they are being observed by a given link. For instance, link `l1` cuts the classes which have a field with the annotation `@Persistent`. It first accesses the pool[1] of fields of the class and then, on each field it accesses the pool of annotations of the field and finally, on each annotation, it reads its name. Each structural element generates a fact when it is being accessed and therefore the class `C` generates the fact that `l1` reads its pool of fields (`readFields('l1','C').`[2]), each field generates the fact that `l1` reads its pool of annotations (`readFieldAnnotations('l1','C').`) and each annotation generates the fact that `l1` reads its name (`readFieldAnnotationName('l1','C').`) We do not need to keep additional information about what is being accessed as only the intentional cut is important and useful.
- Upon intercession (*i.e.* structural changes), structural elements generate logic facts indicating the changes being made. For instance, class `C` generates the fact that link `l2`, applied on class `C2`, adds the annotation `@Persistent` on the field named `f`, in class `C` (`C` and `C2` may not necessarily be the same). That is `addAnnotationToField('l2','C2','C','f','Persistent').`

Besides, upon specification of rules of ordering, visibility or dependency, additional facts are generated. These facts are not directly involved in the interactions, but they influence the

[1] We say that each class has a *pool* of fields, a pool of methods, a pool of constructors, etc. Similarly, other structural elements may have pools of other elements, such as annotations.

[2] This is a Prolog fact, thus respecting Prolog syntax: Strings are surrounded by ' '

detection as they determine the scope of the aspects and the feasibility of the interactions.

Visibility

As reasoning on the sets of facts generated in order to find interactions, the logic engine needs additional data concerning the effectiveness of the interactions. These data are related to the visibility of the structural modifications made by SLinks. So, we need a way to tell the logic engine whether or not the modifications are visible, or what links can see the modifications. For example, the code

```
Rules.augmentViewOf(persistency, history);
```

found in section 3.6 declares that `persistency` sees all changes made by `history`, and therefore generates the fact

```
visible('history','persistency').
```

which means that changes made by `history` are visible to `persistency`. In the same way, the code

```
Rules.addToDefaultView(history);
```

says that all the changes made by `history` are part of the default view, and therefore that they are visible to every other link. Thus it could generate the fact

```
visible('history',_).
```

which says that `history` is visible to any other link.

In such way, the logic engine, with the appropriate rules, can make the difference between interactions disabled due to insufficient visibility from the one partially enabled by sufficient visibility.

Ordering

Once a precedence rule has been declared between two links, we consider that the interaction has been coped with (resolved). The rule can either disable or enable the interactions

(see section 4.4.2). In any case, the important information is that the interaction has been resolved and therefore shall not be reported to the programmer, or at least not the same way.

Defining an ordering rule thus generates the fact

```
ord('link1','link2').
```

where `link1` is the SLink which is applied first and `link2` the one applied in second. Then the detection understands that `link1` and `link2` are ordered if there is either `ord(link1,link2)` or `ord(link2,link1)`, but still knows whether the interaction is enabled or disabled.

Aspects Dependencies

We have seen that the only thing the detection mechanism needs to be aware of is when a SLink is being removed from the set of links which are to be applied to class (see sections 4.5.1, 4.5.1 and 5.4), either by class exclusion or mutual exclusion. Therefore, when resolving the the exclusions, a fact

```
remove('link','class').
```

is emitted each time a link is removed from the set of links that should be applied to a class.

In such way, we are able to make the difference between disabled resolved interactions from the others.

6.1.2 Rules

Along with the facts generated concerning the added elements, the observed elements and the composition rules, we define rules for detecting interactions. These rules follow the principle of section 4.3: they look for actions interacting with cuts. Below is an example of the minimal rule which detects all interactions between links that adds an interface to a class and links whose cuts contain classes with an interface:

```
interactInterfaceName(Link1,Link2,Class1,Class2,InterfaceName)
     :- readClassInterfaceName(Link1,Class2),
        addInterface(Link2,Class1,Class2,InterfaceName).
```

`Class1` is the class on which `Link2` is applied and `Class2` is the class where `Link2` adds the interface with name `InterfaceName`. The result of the detection using this rule is a tuple which is to be understood as: *"`Link2` interacts with `Link1` because `Link2`, applied to class `Class1`, adds the interface named `InterfaceName` to class `Class2`, while `Link1` is looking for interfaces on `Class2` in its intentional cut."*

As such, this rule is not complete as it does not take into account what was discussed earlier. It should indeed detect the state of the visibility, the ordering and whether or not `Link2` was really applied and not excluded. The correct rule which takes all that in account is as follow:

```
interactInterfaceName(Link1,Link2,Class1,Class2,InterfaceName,View,Order,Mutex)
      :- readClassInterfaceName(Link1,Class2),
         addInterface(Link2,Class1,Class2,InterfaceName),
         visible(Link2,Link1,View),
         ordered(Link1,Link2,Order),
         remove(Link1,Class2,Mutex).
```

The above rule calls three additional predicates which check what composition rules were declared to solve the interaction, if any, an return informations about the composition thanks to the variables `View`, `Order` and `Mutex`

6.2 Summary

In this chapter we have seen how the interactions are detected. A logic engine has been added to the Reflex kernel and is used for detection. Fact describing the behavior of the SLinks, that is, what they look at and what they modify, are generated during introspection and intercession so as to give food for thought to the engine. Additionally, several rules have been defined in order to detect each specific interaction. Finally, the state of the interaction is reported as well during the reasoning: each interaction is marked with meta-data concerning the visibility and the ordering between the two concerned links, and concerning actual exclusion or not of the modifying SLink.

We now turn to the resolution mechanisms. In the next chapter, we study how interactions are resolved thanks to the rules defined by the user.

Chapter 7

Interaction resolution mechanisms

In the previous chapter, we have discussed the detection mechanism and the effect of the various composition rules upon it. Now that the interactions are detected, they shall be resolved, if necessary.

The aim here is to take advantage of the precedence rules, of the visibility mechanism and enable or disable action-cut interactions. Considering the structural links (SLinks) installation described in section 3.7, it is impossible to enable interactions because all the introspections are done before all the actions and because the default visibility hides the modifications. Therefore the composition mechanism of SLinks has to be reviewed.

In this chapter, we first review the interaction resolution possibilities and see how they are limited by the previous SLink application mechanism. We then review the previous SLinks application mechanism and study its limits before introducing the new mechanism implemented.

7.1 Interaction resolution mechanisms

We have defined in chapter 4 some resolution mechanisms depending on the kind of interactions (*i.e.* with base code, between actions or action-cut).

We have already reviewed several mechanisms used to resolve the interactions with base code and between actions in section 4.5, namely skipping an action and the combining methods. These mechanisms are involved during introspection, while adding some structural elements.

We here review the resolution mechanisms used for resolving action-cut interactions, namely

the precedence rules (ordering) and the visibility rules. These two resolution mechanisms are directly involved in a higher perspective: the SLink installation process.

7.1.1 The visibility mechanism

The visibility mechanism enables to control the visibility of added structural elements so that it is possible to control how these modifications will affect the scope and behavior of other links. This mechanism is therefore used as a tool for resolving interactions as invisibility of modifications make the action-cut interactions non-effective.

To use it, one can extend the visibility of the modifications done by a SLink `aSLink` to another link `aLink` (not necessarily SLink) using the code:

```
Rules.augmentViewOf(aLink, aSLink);
```

It is also possible to promote the modifications done by a SLink to every other links by using:

```
Rules.addToDefaultView(aSLink);
```

However, when a the structural modifications made by a SLink are defined to be part of the default view (*i.e.* visible to every other link), it is legitimate to think that they should be applied before any other link does anything on the class (either introspection or modifications). That way, these modifications can be promoted to the rest of the links. This introduces the concept of *absolute* precedence: the link is applied before everything else.

There are then at least two levels of ordering, the absolute level which involves SLinks which are to be applied before any other, and a second level which contains all the other SLinks. The study of the latter level is continued in the following section.

7.1.2 The precedence rules

Precedence rules are used to enable or disable an action-cut interaction as introduced earlier (see section 4.4.2). The definition of such rules, in the case of SLinks, is to order two SLinks one before the other. This introduces a concept of *relative* precedences.

In the absolute level of ordering introduced in the previous section, that is to say the level containing the SLinks which shall be installed before the others so as to promote

the modifications as part of the default view, when several SLinks co-exist in that level then they shall be ordered as well. Indeed, if one SLink should be applied before all the other and if another SLink should do the same, then we have an unsolvable situation. The solution we chose is to ask the programmer to define an ordering for the SLinks in that situation.

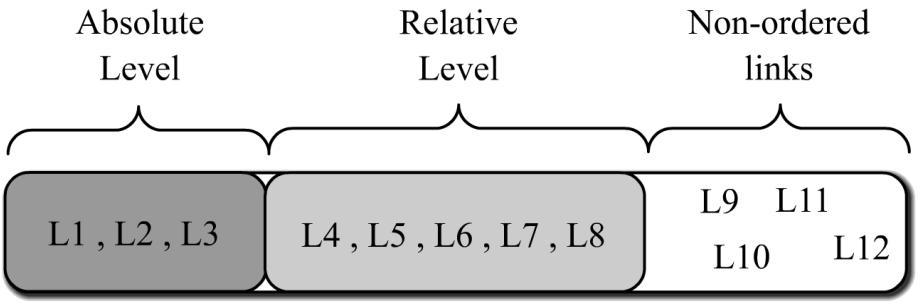

Figure 7.1: Three different levels of ordering

We can then identify three levels of ordering as shown in figure 7.1. The absolute level contains the SLinks concerned by the absolute precedences. The relative level contains the SLinks concerned by relative precedences which are not part of the absolute level. The third level contains all the other SLinks, which have not been ordered. Note that it is incoherent to say that a SLink from the relative level is to be applied before a SLink of the absolute level. Either this link should be in the absolute level as well or no order shall be declared.

Rules have been defined to declare these precedences:

```
(1)  Rules.addToDefaultView(SLink aLink);
(2)  Rules.declareAbsolutePrecedence(SLink aLink);
(3)  Rules.declareRelativePrecedence(SLink aLink1, SLink aLink2);
```

To declare that a SLink shall be applied before the others, one must declare it as part of the default view (1) or simply declare it as part of the absolute level (2) although we do not recommend this method. Then, one must order the SLinks in the absolute level among themselves using simple relative precedences (3). To order the other SLinks, one should use (3).

While declaring orders of application, one can define entire ordering chains. For instance, if `A1` should be applied before `A2`, which in turn should be applied before `A3`, then a an ordering chain `A1<A2<A3` is therefore defined. In absolute and relative levels, such chains can be defined. Chains can be very long depending on the precedence rules declared, or on the contrary very short. The shortest involve only two elements and is defined by any precedence rule: saying that `A1` is to be applied before `A2` defines the chain `A1<A2`.

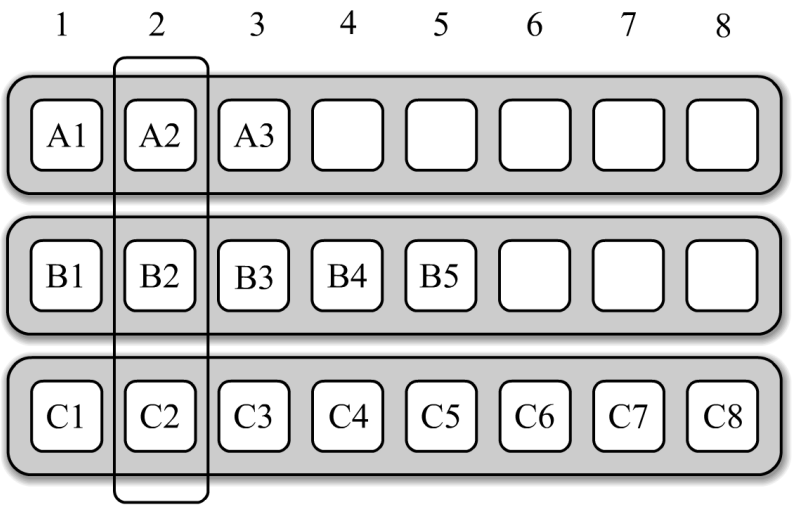

Figure 7.2: An example of Precedence Chain aggregating three ordering chains.

Moreover, when several chains exist, the links of one chain are supposed not to interact with any link of another chain. Otherwise, interacting SLinks should be ordered and therefore belong to the same ordering chains. Therefore, applying two links of two different chains should not create any problem. So we can build a structure which aggregates all the ordering chains of a same level of ordering and which enables to manipulate groups of non-interacting links for application. We shall refer to this structure as a *precedence chain* later on in this document.

An example of precedence chain is shown in figure 7.2. There are three different ordering chains: `A1` to `A3`, `B1` to `B5` and `C1` to `C8`. In a same column, all links are supposed to be non-interacting as no ordering rule have been defined for them.

Having defined all that, it appears that the SLinks installation process described in section 3.7 is insufficient to support ordering. In the following section, we come back on this process and review its limits regarding the desired interaction composition mechanisms. Later on we will introduce a new design for this process which is better suited for these composition mechanisms

7.2 Limits of the previous system

In section 3.7 we have described the previously implemented Structural Link Application (SLA) process. We have seen that when a class is loaded in the JVM, it enters two phases: the SLA phase and the Behavioral Link Set-up (BLS) phase. The one that is of some interest here is the SLA.

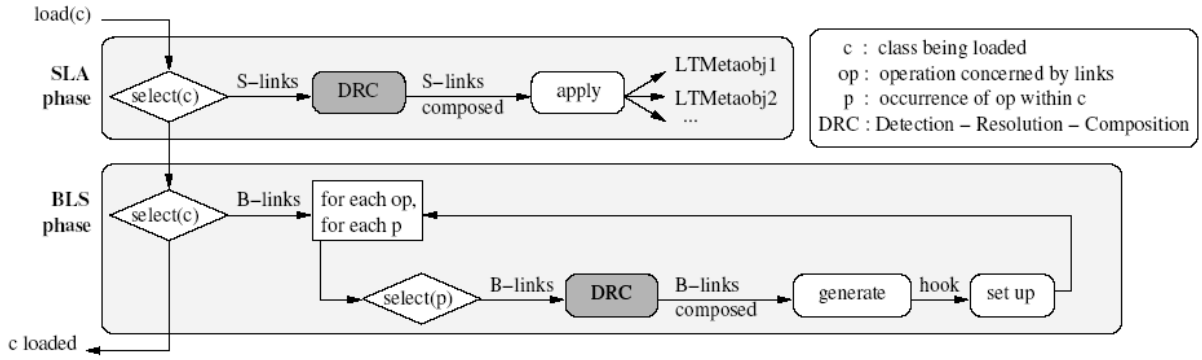

Figure 7.3: Loading of a class in Reflex AOP Kernel

The SLA phase is composed of only one main step containing three inner steps. First a *selection step* (a diamond in Fig. 7.3) determines the set of SLinks that apply to the class being loaded. This is done by introspection on the class. Then the selected links are composed according to user specifications. Due to exclusion rules, some SLinks may be removed from the set of selected links. And finally the remaining SLink are applied, in order if they have been ordered, or in arbitrary order otherwise, bringing structural modifications to the class. This is intercession.

One could notice that with such process, action-cut interactions are inexistent among SLinks. The modifications made by a SLink can still modify the scope or the behavior of a behavioral link, but this is out of the scope of this study.

Consequently, defining ordering among SLinks in order to enable action-cut interactions is not working as it does not help to have the introspection of the affected SLink after the modifications done by the other SLink involved in the interaction.

It appears then that this process needs to be modified so as to be able to truly enable action-cut interactions among SLinks. This is the leitmotiv which conducted the implementation of the new process, which is presented in the next section.

7.3 A new link installation process

A new process has been designed to install the links. As BLinks are out of the scope of this study, only what deals with SLinks has been modified, that is to say the SLA phase.

The guide lines when designing this new process were:

- We want to support every kind of interactions we have identified, especially action-cut

interactions (See chapter 4).

- We want to be able to enable or disable action-cut interactions.
- We want to leave little space to unknown as it can bring unexpected behavior.
- We want to fully use the composition possibilities given by the ordering and the visibility mechanisms.

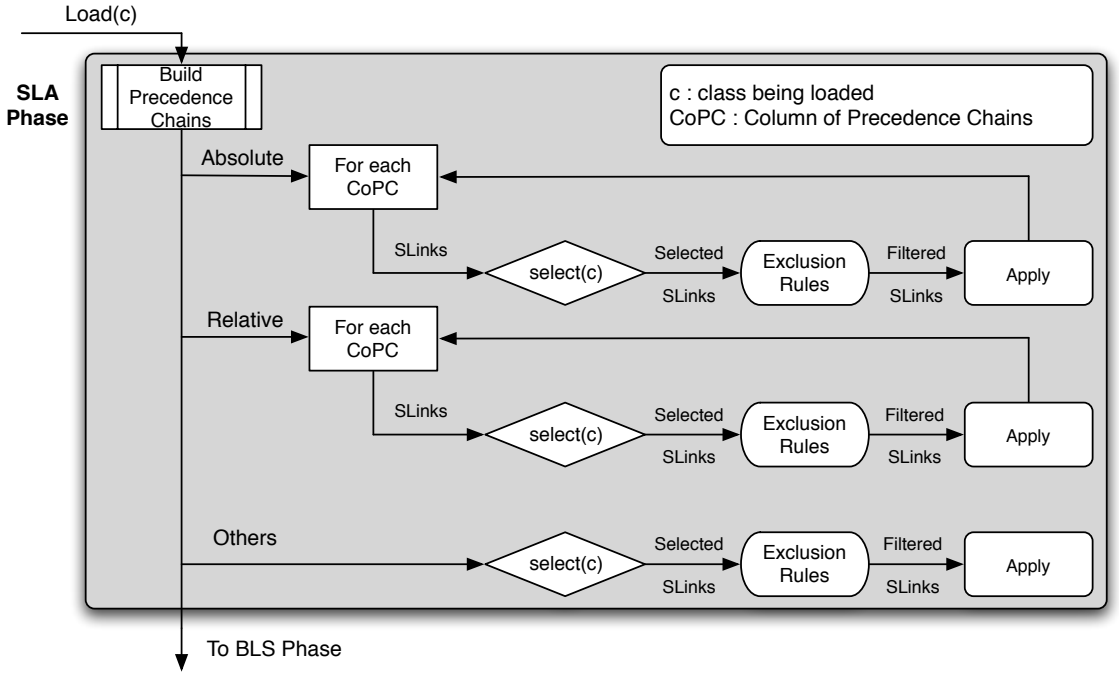

Figure 7.4: Modified SLA phase

Figure 7.4 shows the new process as it is currently implemented. The process is divided in three main steps, related to the three levels of ordering introduced in section 7.1.2 and figure 7.1. The first step relates to the links of the absolute level, while the second step relates to the links of the relative level and the last step to all the other links.

The first two steps work the same way, only a difference is made between the absolute level and the relative level of ordering. At first, precedence chains are build for the absolute and relative levels and are aggregate into the two different precedence chains. Then, for each column of the precedence chain concerned (see Fig. 7.2), a selection test occurs (a diamond shape in the figure). The SLinks selected are filtered using exclusion rules and the remaining SLinks are applied. In the last step, as no order has been specified, all links are tested at the same time and the selected links are applied in a arbitrary order. However, no iteration is done on this step.

This design allows to have action-cut interactions has two ordered links are tested and applied one after the other. It also allows to promote the modifications which should be part of the default view to every other link by installing these SLink at the very first stage of the process. And finally, as there is no iteration during the last step, there can be no "surprises". Indeed, the only way to have an action-cut interaction is to enable it both with visibility and ordering rules. This decision gives an essential role to the detection tools which have to tell the programmer what he can enable and how.

7.4 Summary

In this chapter we have reviewed the interaction resolution mechanisms used for action-cut interactions. We have seen that promoting some modifications to all the other links and adding it to the default view implies that these modifications shall be done at the very beginning, thus leading to the concept of absolute precedence. We have also seen that the ordering rules define another concept of relative precedences and that the absolute level of precedences and the relative one shall be treated separately. We have also seen that the previous SLA phase was no adapted for such resolution mechanisms and that a new process was needed. We therefore have introduced a new design for this SLA phase. This phase is now divided in three steps, regarding the three levels of ordering (absolute, relative and the rest), and thanks to a structure of *precedence chains* and a succession of introspection-intercession phases we are able to apply the SLinks according to the ordering rules defined.

Chapter 8

The technique in details

In the previous chapters, we have introduced the design and ideas of the detection tools and composition mechanisms. In this chapter, we describe in more details how these mechanisms and tools work. In the first section, we study the detection tools, how the link with Prolog is done and how the facts are generated. Then in a second part we come back on the composition process and more particularly on the composition algorithm, and then see how we can use Prolog to order the links according to the rules defined by the developer.

8.1 Detection

We have seen that the detection mechanism is based on a logic engine which reason thanks to detection rules previously defined and facts generated by the elements being introspected and modified. In this section, we will review how the detection mechanism works in details, how the link has been made between Prolog and Reflex, and how facts are generated.

8.1.1 Switching on the detection mechanism

By default, the interaction detection mechanism is disabled. In order to enable it, the programmer has to add a simple parameter to the command line when launching Reflex. The parameter chosen is `--detect:on`.

To do so, another `ArgumentHandler` has been added to Reflex so as to take this parameter into account. Moreover, a global `DetectionLevel` has been added, which gives the actual state of the detection mechanism. By default, the state is `OFF`, but when the parameter is

added, it is simply switched to `ON`. Other systems used for the detection refer to this state so as to adapt their behavior.

8.1.2 From Reflex to Prolog

As introduced earlier, Reflex now uses a logic engine to reason about the interactions between structural links (SLinks). This logic engine is basically a Prolog engine attached to Reflex. There are several different implementations which allow to make a link between Java and Prolog: some based on a real Prolog environment and mechanisms to communicate with it using Java, and others which provide a Java implementation of the Prolog engine.

In order to keep Reflex independent from other softwares, it has been decided to use a Java implementation of Prolog. Still, we have made great efforts to keep the link between Reflex and Prolog as generic and modular as possible so as to be able to change the Prolog engine at will.

The gateway between Reflex and Prolog is divided into two parts, which are two levels of abstraction going from Prolog to Reflex. A first part abstract the Prolog engine into an abstract logic engine, thus providing a logic interface independent from the chosen Prolog implementation, and a second part links this logic engine abstraction to Reflex. However, as the gateway is seen from Reflex to Prolog and not the opposite, the names are in the opposite order: from Reflex to the logic abstraction and from the logic abstraction to Prolog.

In the overall, it gives a system as presented on figure 8.1. In order to change the Prolog engine, one just have to change the Logic to Prolog gateway by another adapted to the new engine.

Logic to Prolog

The Relfex to Prolog gateway is divided in two parts, as shown on figure 8.1. This section presents the Logic to Prolog part.

The Logic to Prolog gateway is represented by an interface `LogicToPrologGateway` which provides a number of low level, yet generic, methods used to communicate with the chosen Prolog engine. Implementations of this interface build a link between these methods and a particular Prolog implementation.

These methods are generic and used to manage the rules and facts of the knowledge base and to send queries to the engine. They are independent from any particular implementa-

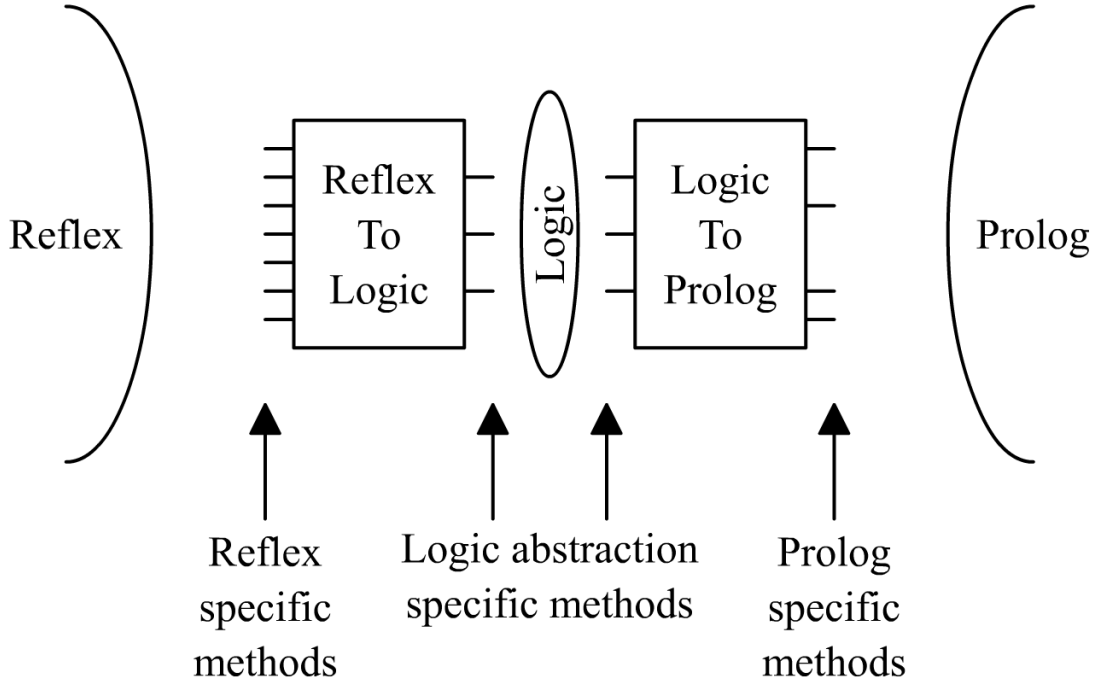

Figure 8.1: Design and decomposition of the Reflex to Prolog gateway

tion.

So far, there are two different implementations in Reflex. The first one is embedded and used as default as it relies on a Java implementation of Prolog, named *JLog* [12] and therefore does not bring external dependencies. The second one is based on the *Prolog Development Tools* developed at the university of Bonn [16]. This *PDT* relies on the program *SWI-Prolog* which is a multi-platform Prolog engine, thus bringing external dependencies.

If one want to change the Prolog engine, all he has to do is replace the Logic to Prolog gateway by another one building a link to the chosen engine. This ensures that Reflex is kept independent from any Prolog implementation.

Any implementation of this interface should store a list of facts and rules, as they are added through fact generation. It is also generally needed to write the rules and facts in a text file and load this file in the Prolog engine before making queries.

The second part of the Reflex to Prolog gateway, the Reflex to Logic gateway, is presented in the next section.

Reflex to logic

This part of the link is represented by a single interface `ReflexToLogicGateway` which make the link between Reflex and any `LogicToPrologGateway`. This interface provides all the methods used to generate the facts during introspection and intercession and a simple query method.

It has been studied what facts were needed and possibly generated and each fact has a method dedicated in this interface. During introspection, the facts concern *reading* of the existent elements in classes, therefore leading to a list of *reading* facts. Thus this interface provides methods that are to be called when accesses are made to class elements during introspection. For example, if a class selector reads the name of a field in a class, a call to the method `readFieldName(Slink aLink)` is to be made, which will generate the proper fact and send it to the `LogicToPrologGateway` for storage. How this call is made will be discussed in the section 8.1.3

Furthermore, there are two different instances of this `ReflexToLogicGateway`. One is dedicated to the detection mechanism while the second is used for ordering the SLinks before intercession. The separation between the two is mostly just a matter of convenience and optimization of the performances. Both instances can be summoned using the methods (1) for ordering and (2) for detection:

```
(1)  API.getOrderingPrologGateway();
(2)  API.getDetectionPrologGateway();
```

A `PrologGatewaysFactory`, based on the design pattern of the Abstract Factory, manages the creation and access to the instances depending on the detection level introduced in section 8.1.1, and on the kind of gateway desired (for ordering or for detection).

8.1.3 Fact generation

During introspection and intercession, as elements are being observed or added, facts have to be generated to support the detection of interaction. As introduced before, these facts are generated through calls to methods of the `ReflexToLogicGateway`. The question is: who makes these calls?

A first alternative would to define a small Domain Specific Language (DSL) to describe the intentional cut of the links. This DSL would then call the appropriate methods of the gateway according to what has been declared.

A second alternative would be to change the pointcut language and adopt a language closer to the logic one, as it is done in Compose* for instance. This solution allows to generate

accurate facts concerning the cut of the links but greatly reduces the expressiveness of the pointcut language as it is based on predefined predicates.

A last alternative is to let the structural elements generate facts as they are being observed. This solution is less precise but is automatic and does not require additional work from the programmer.

Concerning the modifications being done, letting the modified elements handle the calls to the appropriate methods of the gateway seems a good solution. Indeed, we know exactly what is being done, by who and on what and therefore the facts generated are accurate.

So the most relevant answer seems to let the elements being observed and/or modified call the appropriate methods of the gateway so as to generate the facts for detection of interactions. These elements are the structural elements of the Reflex API, representations of the Java elements manipulated. These elements are presented in the class diagram in figure 8.2.

Figure 8.2 shows the hierarchy of the structural elements in Reflex API. The methods and fields have been omitted in the class diagram as they are not relevant for the coming explanation and make the diagram heavier.

Leaf elements, such as `RClass` and `RMethod`, have a bunch of methods to access their properties, such as signature, parameters etc., and to modify the elements, mostly by additions of other elements. Some methods allow to rename the elements or to change the hierarchy of the class, but such modifications are not taken into account in the present study; they shall be studied later on. For fact generation, leaf classes of the tree have been subclassed by classes with name ending with *Logic*. A mechanism based on the *Abstract Factory* pattern has been set to choose between the simple `R`*`Element`*`Impl` or `R`*`Element`*`ImplLogic` depending on the state of the `DetectionLevel` (cf. 8.1.1.) When the detection is `OFF`, simple implementations are instantiated, but when the detection is `ON`, *logic* implementation are instantiated.

These logic implementations override every methods of their superclass and manage the calls to the methods of the `ReflexToLogicGateway` for fact generation. Concretely, every method of such implementation simply generates the proper fact by calling the proper method of the gateway, but only if the phase is correct (introspection/intercession/other), and then forwards the call to its superclass. For example, the method `getName` in the logic implementation would follow the following pattern:

```
public String getName()
{
   if the phase is introspection
      then call the method readName(...) of the ReflexToLogicGateway;
   return super.getName();
```

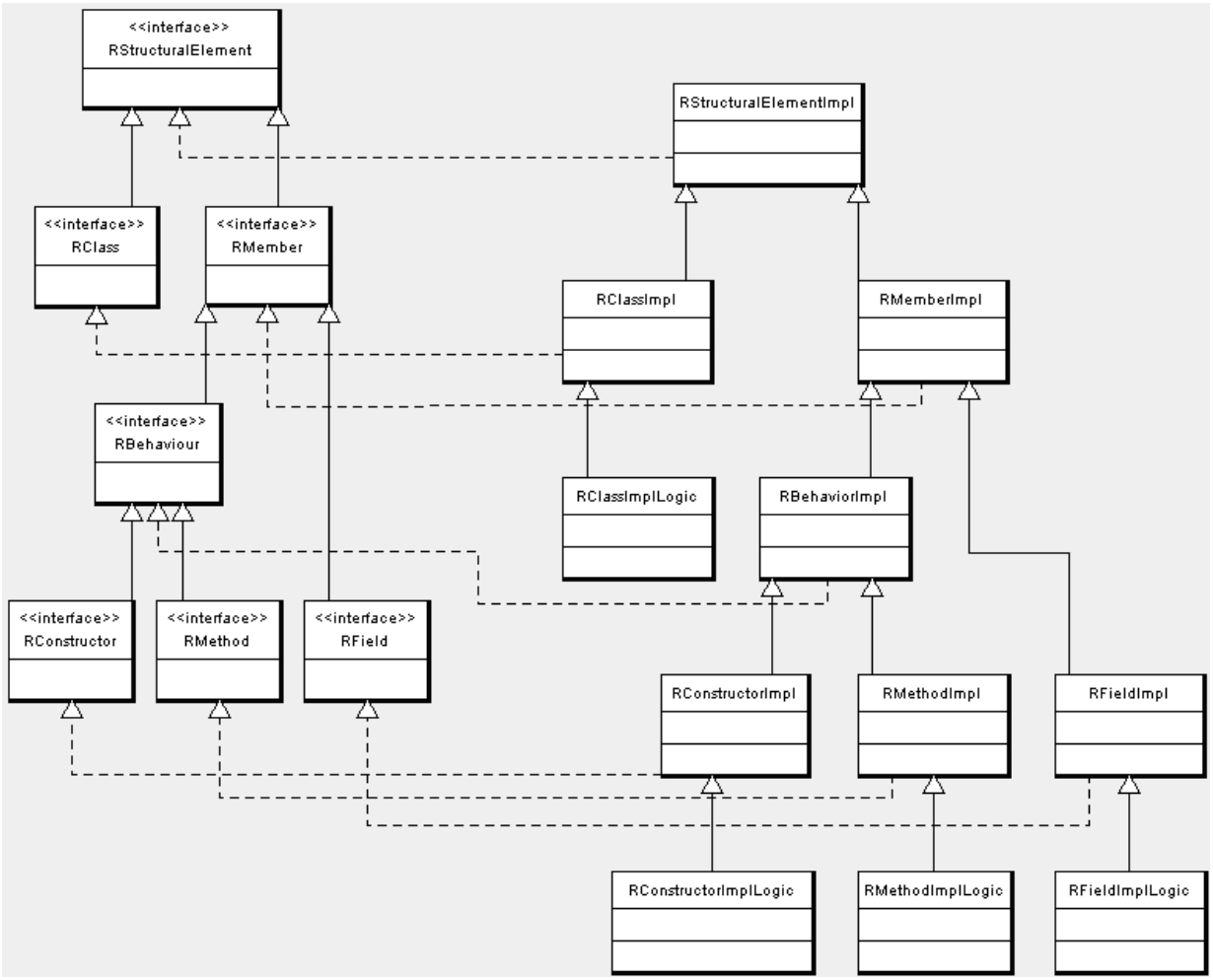

Figure 8.2: Hierarchy of Structural Elements

```
}
```

8.1.4 Interactions reporting

Detecting the interactions, and therefore the possible conflicts, is not all. Without a good reporting to the user, detection doesn't bring much. And as the matter of interactions is complex, there is a need for an advanced reporting tool, such as, for instance, a graphical interface which allows to browse the detected interactions and to sort them by various categories. The design and implementation of this graphical interface is out of the scope of this study and so has not been done.

This reporter has to be aware of all kinds of interactions that exist

About the interactions

Interactions happen between two links as explained in the model of action-cut interactions. In case of bigger interactions involving more then two links, then several interactions are reported.

The two links are not at the same level: one is the *modifying* link (*i.e.* the action) while the other one is the *reading* link (*i.e.* the cut). An interaction stores both and can be sorted according to both kinds of links.

Furthermore, there are two classes involved in the interaction. The *modifying* link is applied to a class but may do the modifications on another class. For instance, link `L` may say that if class `C` has a specific annotation and some specific methods, then a new class shall be created. This new class is called `VisitorC`, implements the interface `Visitor` and provides a particular method which is generated according to the elements of `C`. Therefore, in this example, the class on which the link is applied and the class on which the modifications are done are not the same. In consequence, an interaction stores both classes as the *application* class and the *destination* class. The *reading* link is only concerned by the *destination* class, which is the class being modified.

Moreover, a interaction can be detected even if it is logically impossible. For example, the detection mechanism detects possible interactions even if the visibility of the modifications disables interaction, but the interaction is marked as insufficient visibility. Still, the programmer can see it in the report and therefore may choose to extend the visibility so as to enable the interaction. The same goes for rules of ordering and exclusion. Interactions are still detected even if there is an ordering rule coping with it, but the found interaction will be marked as already resolved. Thus the reporting message will be different as it will show that the composition process has already been applied here. In the same line, mutual

exclusion are taken into account: the possible interaction is detected but reported to the programmer as non valid as the reading link has been excluded. This way, the programmer can know everything and modify the three kinds of rules at will, knowing what should be the result.

In the case of ordering rules, the detection mechanism reports that the two involved links are either not ordered, or ordered so as to enable the interaction (*i.e.*the action is before the cut), or ordered so as to disable it (*i.e.*the cut is before the action).

Finally, interactions are of different kinds, as explained in section 4.4.1. Some interactions are about the name of some elements, other about the existence of elements, other again about the types of some elements. etc. In order to store this information as well, several kinds of interactions have been created. The interface `Interaction` defines the methods that any interaction should implement. This includes methods for accessing the links and the classes concerned, as well as the description of the interaction, a conflict report generation method and queries about the visibility, ordering and mutual exclusion. Then there is a whole hierarchy of interactions depending on their type. Figure 4.1 show a simplified hierarchy of these interactions. The explanation of this schema has already been carried out in section 4.4.1.

Interactions reporter

An interactions reporter is an object which can be queried about the interactions found. It is in charge of querying the Prolog engine to detect the interactions which it stores for further reuse.

An interactions reporter implements the interface `ConflictReporter` which declares basic methods for manipulation of found interactions, for example methods for retrieving the interactions which concern a specific link or a specific class. Using this, any graphical reporting tool can make queries to the base of conflicts.

Note that before showing the found interactions, one must ask the `ConflictReporter` to search for them, that is to say to query the Prolog engine for all kinds of interactions. This is not done automatically as it burdens the performances a lot and as detection is not always required.

8.2 Resolution

In chapter 7 we have introduced the design of the composition process. We have seen that we need to have an iterative process composed of several phases of introspection and

intercession in order to have action-cut interactions. In this section, we review it in details and see the influences of the ordering mechanism on it.

But first of all, we come back on the matter of the two resolution mechanisms introduced in section 4.5: skipping an action and methods combination. These two mechanisms interact as they apply at a common join point and therefore shall be composed the same way we compose aspects.

8.2.1 Interactions between the Action Skip and Methods Combination

Skipping an action and combining methods work at a common join point of the system and therefore interact. Indeed, both system first try to add the conflicting element and react when an exception is raised. Therefore they have to be composed in the right order and way so as to provide the desired behavior.

When a method is added although there is already a method with the same signature in the code, either from the base code or previously added by another SLink, the compiler raises an exception saying it cannot compile the new code.

The action skip mechanism provide means to skip the addition of a conflicting element. It is in fact implemented using fault tolerance: it first tries to add the element and when an exception is raised saying the addition failed then it silently catches the exception and continue. In normal cases, SLinks are not tolerant to faults and therefore the exception is normally thrown.

The Method Combination mechanism acts the same way: it reacts on a failed attempt to add an element. It first catches the exception, checks for method combination rule and if not found throws back the exception, but if found applied the combination mechanism. We assume that when both rules are defined and applicable for a same method, the combination prevails and therefore shall be applied first.

The final mechanism is then:

1. First try to add the method.
2. If a error occurs (exception raised), then try:
    - If a method combination has been declared for this method then do it.
    - Otherwise if the link is *tolerant* to faults, then do nothing.
    - Otherwise report the error (throw the exception back).

3. Continue normally.

In the case of other structural element, this problem doesn't exist as there is only the Action Skip mechanism.

8.2.2 Composing the SLinks

In sections 7.1.1 and 7.1.2, we have introduced two concepts in the ordering of SLinks: the *absolute* precedence and the *relative* precedences. We have also seen that there are some relative precedences among the SLinks of the absolute level.

This leads to two different levels of ordering, two groups, which have to be treated separately. Still, each group have to be processed and the algorithm to do so are the same, as we will see further in the explanation.

We will first study the mechanisms used to order the SLinks according to the rules defined and what is the structure used, then we will come back to the composition process.

Applying the links in order

During the iterative process introduced in chapter 5 the developer defines ordering rules to deal with the various interactions. These relative precedences build chains of SLinks defining the application order. Let us study an example. Suppose there are tree ordering chains:

```
La<Lb<Lc<Ld<Le<Lk<Ll
Lf<Ld<Lg<Lm<Ln<Ll
Lh<Li<Lj
```

(Where `La<Lb` stands for *"La is to be applied before Lb"*.)

We can notice that `Ld` is a crosspoint between two chains, and so is `Ll`. Now, a SLink shall be applied only once in the composition process. Therefore, the two instances of this link shall be treated as one.

We can assume that since there is no ordering rules between `La` and `Lh`, they do not interact and therefore can be applied at the same time. So during the composition process, we can apply all the links of the same column at the same time. By applying the links column after column, we are sure to respect the order defined and we process all the different chains at once.

Moreover, each SLink shall be applied only once. If one link is involved in several chains, then it has to be placed in a single column so as to be used only once.

Considering these remarks, we can come up with a design for the precedence chains, given as follow:

```
La | Lb | Lc | Ld | Le | Lk | __ | Ll
__ | __ | Lf | Ld | Lg | Lm | Ln | Ll
Lh | Li | Lj
```

All the SLinks are placed in rows, according to their precedence chains, and every time a SLink is a crosspoint between several chains, then it is placed in a single columns. Then, over the iterations of the composition process (See section 7.3), the links will be applied by column.

A possible simplification.
This is the optimal way to order the SLinks concerned by the precedence rules, but it is not the easiest to build... Crosspoints have to be identified, localized, and empty cells have to be placed in the rows to ensure the crosspoints are in single columns.

An easier alternative is to have only one chain containing all the links involved and ordered according to the precedence rules. So for instance:

```
[La,Lb,Lc,Lf,Lh,Ld,Le,Lk,Li,Lg,Lm,Ln,Ll,Lj]
```

The construction of such chain is always possible, assuming that there are no circular declarations. This chain has the advantage to be easier to build, but on the other hand can slow the process as there will be as many iterations as there are links in the chain. This is especially true if there are many ordering chains involving only few links. In this case, instead of having three loops involving each twenty links, there would be sixty loops involving each only one link. Fortunately, this case is not frequent as there are rarely many SLinks defined and interacting.

Improvements and optimizations.
The first solution with the two dimensions array containing the links to be applied in order is the optimal solution, yet more complex. The solution proposed is a simple and convenient solution, yet not very optimized. In further optimization, one can think of a two dimensions array containing several lists such as the second solution. To do so, we just need to separate the links in several lists, putting together the links that are somehow connected and therefore separating the links that are not comparable at all. This solution would reduce slightly the length of the final list in most cases, and would come very close to the optimal solutions in some cases. For instance, the first given example would become:

```
La | Lb | Lc | Lf | Ld | Le | Lk | Lm | Ln | Ll
Lh | Li | Lj
```

In the case of numerous small ordering chains, this solution will be far better than putting all this chains in a single list.

This is the actual used solution. A pre-sorting as been set, separating the initial list into several ones, each list containing the SLinks that are somehow connected. Then each sublist is sorted and checked, and we get a list of lists as a result.

The sorting of each sublist is based on the sorting algorithm usually known as *insert-sort*. It is possible to use it because a single sublist contains only links that are somehow connected and therefore comparable. However, there are some cases where comparison is not possible: `La` and `Lf` are not comparable. Indeed, they are both before `Ld` but we still cannot compare them. Therefore the *insert-sort* algorithm has to be modified to take into account the case when no order is known between two elements.

This algorithm has been programmed in Prolog, using the existing Reflex-Prolog interface, because it is more convenient when it comes to such reasoning and programming. The code is as follow:

```
insert_sort(List,Sorted) :- i_sort(List,[],Sorted).

i_sort([],Acc,Acc).
i_sort([H|T],Acc,Sorted) :-
    insert(H,Acc,NAcc),
    i_sort(T,NAcc,Sorted).
```

This is the basic code made to make the use of the algorithm more user-friendly. The user calls the function, giving an unsorted list as `List`, and receives the sorted list as `Sorted`. This calls the recursive function `i_sort` which will add the element one by one at a correct position. The sorting part of the algorithm is in the predicate `insert/3`.

```
before(X,Y) :- before(X,Y,[Y]).

before(X,Y,_) :- ord(X,Y).
before(X,Y,Memo) :-
    ord(X,Z),
    acceptable(Z,Memo),
    before(Z,Y,[Z|Memo]).
```

```
acceptable(_,[]).
acceptable(X,[Y|Xs]) :-
    X \== Y,
    acceptable(X,Xs).
```

This is the comparison predicate: we say an element `X` is before another one `Y` either when it has been defined as such (`ord(X,Y).`) or when `X` is before another element `Z` and `Z` is before `Y`. The use of the predicate `acceptable/2` is to avoid the circular declarations, if any, as this kind of reasoning can easily go into an infinite loop. It ensures that the `before/3` predicate does not pass by the same element twice.

```
insert(X,[],[X]).

insert(X,[Y|Ys],[Y|Zs]):-
    before(Y,X),
    not(before(X,Y)),
    insert(X,Ys,Zs).

insert(X,[Y|Ys],[X,Y|Ys]):-
    before(X,Y),
    not(before(Y,X)).
```

This is the exact translation of the classic *insert-sort* algorithm:

- Inserting an element in an empty list is a list with the element.
- Inserting an element X in a sorted list which first element is Y, having X greater than Y, is inserting X in the tail of the sorted list.
- Inserting an element X in a sorted list which first element is Y, having Y greater than X, is inserting X in the first position of the sorted list.

The `not(before(X,Y))` and `not(before(Y,X))` are just to make sure that there are not circular declarations between two elements.

```
insert(X,[Y|Ys],[Y|Zs]) :-
    not(before(X,Y)),
    not(before(Y,X)),
    insert(X,Ys,Zs).
```

This part is needed as all pairs of elements cannot be compared. This states that if no comparison is possible, then it should try to insert the element in the tail of the sorted list.

This algorithm deals with the problem of circular declarations in several ways. First by ensuring that `before/2` does not loop because of them, and secondly by ensuring, in insert-sort algorithm, that if a link is before another link, then it is not after at the same time. In case of circular declaration, the algorithm fails and no result is returned.

This algorithm is used to construct the precedence chain by giving it a list containing all the links involved in precedence rules and by retrieving a list of lists. This list of lists contains in fact sorted sublists. Each sublist is an ordering chain which has been sorted. If the result is empty, it means that there are circular declarations and the programmer is to be informed about it.

The composition algorithm

In section 7.3, a new process has been designed for SLinks installation. This process works in several consecutive loops. Each loop is composed of a introspection phase and a intercession phase. At the very first stage of the composition process, the ordering rules are examined and combined so as to build the precedence chains. All the links mentioned by theses rules are kept aside and treated separately, in two lists of links called L-ordering-absolute for the links of the *absolute* level and L-Ordering-relative for the links of the *relative* level (See section 7.1.2), depending on the kind of precedence rule. Let S-applied be the set of links that have been applied to the class being loaded, S-rejected be the set of link rejected by exclusion rules, S-selected be the set of selected links for this class, during one loop, and S-pool be the general pool of links to be tested. The composition algorithm is as follow:

1. Build precedence chains using defined ordering rules: L-ordering-absolute and L-ordering-relative.
2. From the pool of links of Reflex, place in S-pool all the links except those concerned by ordering rules, that is to say those that are in L-ordering-X.
3. Test and Apply links of L-ordering-absolute:
   (a) For each link of L-ordering-absolute, in order:
      i. Introspection on the current class being loaded (*).
      ii. If the class is accepted, apply the exclusion rules.
      iii. If the link is excluded, add it to S-rejected, then stop.
      iv. Intercession of the link on the class. Add the link to S-applied.

4. Test and Apply links of L-ordering-relative:
   (a) For each link of L-ordering-relative, in order:
      i. Introspection on the current class being loaded.
      ii. If the class is accepted, apply the exclusion rules (*).
      iii. If the link is excluded, add it to S-rejected, stop.
      iv. Intercession of the link on the class. Add the link to S-applied.
5. Test and Apply all other links *i.e.*links of S-pool.
   (a) Empty/flush S-selected.
   (b) Introspection on each link of S-pool. Place in S-selected the accepted links.
   (c) Apply the exclusion rules on S-selected (*). Add the excluded links to S-rejected and remove them from S-selected and S-pool.
   (d) Intercession on each link of S-selected. Add the applied links to S-applied and remove them from S-pool.
6. Verify all the dependency rules on S-applied in order to search for conflicts. For each conflict found, generate an error message for the programmer.

(*) Exclusion rules take into account the currently selected SLink as well as the SLink which have already been applied (S-applied).

(**) Modifications have been done, other links may then apply...

With such algorithm, the links concerned by precedence rules are tested only once but in a precise order. The others are tested several times as long as they still don't cut the class, while those which cut the class are applied and removed.

Problems can occur due to conflicts between mutual exclusion rules and ordering rules. For instance one can declare that `L1` is to be applied before `L2` and that if `L2` is applied than `L1` shall be excluded. This situation clearly conflicts and therefore shall be reported to the programmer as a warning.

Figure 7.4 illustrates the previous algorithm. The figure 3.4 is then obsolete as the SLA phase is replaced by this one. The BLS phase is kept unchanged.

Handling the composition rules

During the iterative process for SLink composition (cf. chapter 5), composition rules are defined. In this section we review how these rules are handles, and more precisely by what.

We have seen that rules are always defined using the `Rules` class, like for example:

```
Rules.declareConflictingActionSkip(aLink);
```

In fact, this object dispatches the messages to processor objects dedicated to these rules. Indeed, there is one processor for each kind of rule, and it is used to centralize them, to store them and to interpret them. We identify the following rule processors:

- a `ClassExclusionProcessor` which handles the rules excluding a class from the cut of a SLink
- a `FaultToleranceProcessor` which is used to implement the Action Skip mechanism
- a `MethodCombinationProcessor` which handles the combinations of method
- a `PrecedenceProcessor` which handles the precedence rules for SLinks and builds the precedence chains
- a `StructuralMutexProcessor` as mutual exclusion rules are treated separately between SLinks and BLinks
- a `VisibilityManager` which manages the access rights of the links

Each processor is based on the singleton design pattern and provides a method `process`, with or without arguments depending on the needs. Then the effect and the method return is different depending on the processor. For example, the structural mutex processor takes a set of SLink in parameters, representing the SLinks which are about to be applied in intercession phase, and returns a set of SLinks which has been filtered using the mutual exclusion rules. The precedence processor queries the Prolog engine in order to get the two precedence chains for the absolute and relative levels of ordering and then provides access methods to these precedence chains.

These objects allow to have a separated treatment of the rules from the rest of the code and therefore help to have a maintainable code.

Part III

Conclusions

Chapter 9

Discussion and evaluation

We have seen, in the previous chapters, how the detection and composition mechanism work. We have seen that the programmers can define rules to compose the structural links (SLinks), such as rules for ordering, visibility of the added elements, aspect dependencies, etc. The point is that these rules are global. In other words, they are always valid, in all cases. One could wonder about the use of local rules, which would be valid and used only at some particular points.

We have also seen that the detection mechanism detects a lot of interactions, and not only those that are actually happening. There are pro and cons for both visions and this is open for discussion.

Furthermore, this contribution deals only with the case of structural aspects *adding* elements. It does not deal with *modifications*, like an aspect changing the hierarchy of a class or changing the access rights of methods (from `private` to `protected` for instance). These aspects are more tricky as they can help a lot but also bring a lot of troubles since they can be destructive (renaming a method bring compilation errors in most cases). Further work shall be done on this matter.

In this chapter, we will discuss briefly each topic. We do not intend to bring any solution or definitive opinion on these questions, but instead bring some preliminary thoughts.

9.1 Local rules

The question arise concerning the use of local composition rules, that is to say rules that are only valid in certain cases or places. The question is still open and shall not be answered in this paper. Still, here is some food for thoughts.

9.1.1 Ordering

In [10], Havinga *et al.* propose an algorithm to deal with circular interactions of type action-cut. These case happen when several aspects depend on the action of another aspect and form a circular chain of dependencies. They resolve it by iterating over the different orders possible until they reach a fix point, as explained in section 2.2.3.

In Reflex, the approach is always defensive: whenever a problem is detected, it is the duty of the programmers to solve it using all the possibilities given. In the same line, the case of circular dependencies is dealt with using precedence declarations. The programmers have to define a suitable order for these interactions so as to have the desired behavior.

As for now, precedence declarations are global, that is to say that the order declared is to be used in all the cases. With circular dependencies, one may wish to have a specific order on one class, another order on another class, and a third order on the rest of the classes.

For instance, using the example given in [10]: two aspects adding annotations, the first one `A1` adds the annotation `WebService` to any class which have a public method with the annotation `WebMethod`, and the second one `A2` adds the annotation `WebMethod` to every public methods of classes which have the annotation `WebService`. Imagine we have two classes `C1` and `C2`: `C1` has the annotation `WebService` and public methods without any annotations, and `C2` does not have annotations but a method with the annotation `WebMethod`. We obviously want that `A1` is before `A2` on `C2` and the opposite order on `C1`.

The use of local ordering rules is then interesting, especially for resolving circular dependencies, due to the position of Reflex for interaction resolution. These rules are not supported in the current state of Reflex but could be added with a little refactoring.

9.1.2 Visibility

In the same line, local visibility rules may be wanted. Ordering rules and visibility rules are closely tight as without extended visibility ordering rules are useless. In order to fine tune the behavior of the structural links, one may want to adjust precisely the visibility of the modifications as well as the precise ordering. Though, this is quite redundant with the local ordering rules in most cases and therefore is just a second way to do the same thing. Indeed, the use a local visibility rule is to enable or disable an interaction on some classes among all those concerned by the same interaction between the same links. But that is just the purpose of ordering rules as well. The only case where this is not possible is the case where the programmer wants to disable an interaction in the case of circular dependency while he wants to enable it on other classes. Therefore, in the case of two links in circular dependency, applying on several classes, if the programmer wants to disable the interaction on some classes, allow it on others and define two different orders

for modification application, then he needs both local visibility and local ordering rules.

9.1.3 Aspects Dependencies

The question arise whether or not it is relevant to define local combination rules. Is it relevant to say that two links are in mutual exclusion only on some classes but not other? It does not appear to be relevant at first sight. However, it is still possible to do so manually by excluding one SLink from one class, as explained earlier.

9.1.4 Action skip

The skip-action rule defines in fact the fault tolerance of a structural link regarding the additions it makes. In normal case, a link reports every conflict (*i.e.* error) when the addition of an element has failed. With such rule, no conflict is reported as the addition is virtually skipped. The point of a local version of such rule would be to skip conflicting additions in certain cases, on certain classes, and to report the error in every other case. This would be useful for the programmer who wants to skip actions which he knows are conflicting but still be warned if new conflicts arise while developing.

This is actually supported in the current version of Reflex.

9.1.5 Method Combination

Combining methods enables to keep the two behaviors in a single method. In the current state, the method combination is global and the combination behavior is the same every time the methods are combined. But one could wish to combine a specific method differently depending on the implementation of the existing method, that is to say depending on the class where the conflict takes place. Therefore providing the possibility to define different behaviors (*i.e.* different bodies) depending on the class could be useful.

This also is not supported in the current state of Reflex but could be added with few refactoring.

9.2 The detection mechanism

There are several ways to see what the detection mechanism should do. Of course it should be able to detect the effective interactions so that the programmer is aware of it. But the

question arise of whether it should detect only that or also detect possible interactions that could arise if the programmer helps it.

The fact is that in Reflex, by default, thanks to the visibility of the structural elements, no interaction is possible between aspects of any kind. The reason is simple: by default, the structural elements added are invisible to the cut of other aspects. Therefore a tool which detects only the effective interactions would not be of a great use at first. The visibility has to be extended before interactions can be effective.

Being able to detect interactions that could arise if the visibility was enhanced and good ordering was declared is a great help to the programmer who has to deal with several aspects which should interact. With the current approach, the programmer can know what he has to declare so as to enable the interaction he wants and not the other ones. Therefore he can fine-tune the visibility of each aspect, define a precise order among them and have the result he wants without having to cope with unexpected interactions.

The other alternative, less elegant, is to disable the visibility mechanism, like in other aspect-oriented approaches, detect only the existing interactions and deal with each of them. This approach is not very subtle and starts from a conflicting situation in order to go to a suitable situation. Our approach, with visibility, starts from a basic, but not conflicting situation, and goes toward a suitable one involving interactions.

On top of that, another dimension of the problem is to be taken into account. This dimension is discussed in the following section.

9.2.1 Accuracy of detection vs. expressiveness of the cut language.

Regarding all the facts generated during the introspection (see section 6.1.1), the question of the accuracy of the detection naturally arises. In fact, there is trade off between expressiveness of the cut language and the accuracy of the detection[1]. In Compose* for instance, the cut language has limited expressiveness (it is not Turing-complete) and greatly resembles a logic language. Hence with such a language it would be straightforward to ensure that only appropriate facts are generated. On the other hand, this means not being able to express advanced selection criteria, *e.g.*: cut every class which has a method with a specific annotation, but no more than three annotations, exactly four parameters and two of type `int`.

The path we choose is to keep the expressiveness given by our reflection-based model

[1]The action language does not cause any problem as, in Reflex, we precisely know what changes are effectively done since structural entities know what changes are performed upon them, as well as by which link. So the facts generated for structural changes are accurate and always correct.

(*i.e.* cut and action are expressed as Java methods manipulating reifications of the structural elements). Then there are two alternatives: *(a)* the structural elements automatically generate logic facts as they are being observed, *(b)* the user explicitly specifies (*e.g.* as annotations or using an embedded DSL) what the cut does.

In the first case, there is a possibility that too many facts are generated. For instance, in the example introduced in section 6.1.1, only the fact that `l1` reads the name of the annotation on a field is relevant as it was the intention of the programmer. The two other facts may lead to non-existent interaction detection as they get involved with other rules. Therefore we detect spurious conflicts, but this is arguably better than missing effective conflicts. In the second case, an explicit contract is actually expressed by the aspect programmer, and hence we can generate the facts that precisely correspond to the intention of the programmer. On the other hand, it is the burden of the programmer to declare this contract. We have finally chosen the first alternative.

9.3 Structural modifications

The present contribution is limited to the case of structural additions and does not cover the question of modifications or removal.

Removing an element using aspects is dangerous, as anyone could have figured: the code may not be compilable anymore as elements are missing. Moreover, this kind of modifications does not make sense anyway and that's why it has never been covered by any aspect-oriented approach (to our knowledge.)

Modifying the hierarchy of a class is supported and used in other approaches, such as AspectJ for instance. It allows to add some behavior and some characteristics to a whole tree of classes at the same time. AspectJ supports it with restrictions: the new superclass must extends the previous superclass so as to keep the original inheritances.

Action-cut interactions involving hierarchy changes are possible. For example: an aspect (1) cuts every class which extends a class `SC` and a second aspect (2) changes declares a class as extending `SC`. This case is detectable using the model we have designed and so our model could be extended to that matter in future work.

Apart from hierarchy modifications, there are other modifications possible. Some of them are useful and relevant whereas other, as removal, do not make sense as such. For example, renaming an element is possible in Reflex but it is dangerous to use it. Indeed, renaming an element breaks the code of every other elements which were using it. So it cannot be used as such. However, it can be used in combination with a copy procedure so as to *clone* elements or classes. The element is then copied and renamed before it is added somewhere. Cloning facilities can easily be involved in action-cut interactions and therefore should be

detected, but the study of the matter is left for further consideration.

Still, one other modification makes sense: changing the *modifier* of an element. For example declaring that a `private` method is now `protected` so that it can be used by the children classes. This is interesting in combination with an aspect which adds a class which extends an existing one with `private` methods. Still, only increasing the access permissions should be possible as restricting the permissions could lead to compilation errors. This is again subject to action-cut interactions as an aspect could search for methods with a specific kind of modifiers. It shall therefore be detected and reported.

9.4 Interactions with behavioral aspects

In the present study, only the interactions among structural aspects has been studied. However, if this approach of detection and resolution using a logic engine turns out to be a valid approach, then it may be considered to extend it to the behavioral part later on.

At first, the bridge between the two "worlds" could be done by detecting the interactions between the behavioral links (BLinks) and the structural modifications made by the SLinks. These interactions are obvious as structural elements are added in order to be used by BLinks at some point during the execution. The point would then be to detect interactions which were not meant in the design of the links. Again, thanks to the visibility mechanism, these interactions are limited or non-existent but one could wish to take advantage of an interaction although it was not designed to be done at first.

Then the matter of BLink interactions is to be studied further to see if it can be detected using logic, and if this approach is the best to be used. Actually, the BLink interaction detection is done in plain Java, and the problems encountered strongly push to search for another alternative.

9.5 Evaluation

The proposed approach detects interactions between two SLinks but do not detect semantic conflicts. Indeed, it is not possible to define what a semantic conflict is, as Compose* does [8, 9]. Hence, every interaction is reported to the programmer which has to decide himself if the two aspects are conflicting or not and so define composition rules. This approach therefore puts more burden on the programmer.

Moreover, as discussed in this chapter, due to the model chosen, the detection is not very accurate as spurious interactions can be detected. This comes mainly from the fact that

the facts generated during introspection and intercession are not linked together. For instance, if a SLink `L1` defines its class selector as accepting classes which implement a specific interface and which have a specific annotation, two facts will be generated: (1) the cut is about a specific interface and (2) the cut is about a specific annotation. Then if another SLink adds an interface to classes, the two SLink will be declared as interacting because of point (1). In fact, in most cases, they are not as the added elements are not sufficient: the annotation is not added. However, in the case of classes which already have the desired annotation, adding the expected interface is sufficient to modify the scope of `L1` and therefore the interaction is not spurious. So finally, detecting spurious interactions is not that bad as long as no effective interactions are missed.

Chapter 10

Conclusion

In the present study we have proposed a solution for automatic detection and declarative resolution of interactions between structural aspect using a logic engine attached to Reflex.

Automatic interactions detection tools are particularly needed to help the programmers to detect interactions when building big programs involving numerous aspects developed by different persons. In this extreme case, detecting interactions manually is almost impossible and therefore automatic tools are needed. In the present study, we propose a solution based on a logic engine attached to the aspect language. We use generated facts regarding the observed and modified structural elements to reason about the cut and the action of aspects. Then, using detection rules previously defined to detect interactions, we are able to combine these facts and detect interactions between structural aspects. In the one hand this solution is not very accurate as spurious interactions may be detected, but on the other hand it allows to detect all effective interactions plus also non-effective interactions which can help the programmer better understanding what is happening in his application and what may happen if he modifies the aspect composition rules.

Declarative resolution mechanism allows one programmer to resolve the interactions detected and to solve the conflicting interactions if any. In this study, we have identified three kinds of interactions and defined wanted resolution mechanisms for each of them. Designing and supporting these resolution mechanisms has forced the structural aspect composition process to be revised entirely. The new process now supports fully all the provided resolution tools and allows the programmer to compose aspects in a very fine-grained manner. Still, these resolution mechanisms could be even more fine-grained by defining local rules, that is to say rules of composition which are only valid at some points of the program.

Furthermore, this study was limited to the case of structural aspects which add structural elements and further work shall be done on those which modify existing elements and on

interactions between structural and behavioral aspects.

And finally, some work is still needed to build the reporting tools, that is to say a complete graphical interface to sort and browse the found interactions. This interface would have to get its data from the detection tools and present them in a good way for the user to understand easily where are the interactions and how they can be resolved.

Bibliography

[1] D. Batory, C. Consel, and W. Taha, editors. *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, Pittsburgh, PA, USA, Oct. 2002. Springer-Verlag.

[2] L. Bergmans. The composition filters object model. Technical report, Dept. of Computer Science, University of Twente, 1994.

[3] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In Batory et al. [1], pages 110–127.

[4] S. Chiba. Load-time structural reflection in Java. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[5] Condor. http://roots.iai.uni-bonn.de/research/condor/.

[6] S. Dawson, C. R. Ramakrishnan, and D. S. Warren. Practical program analysis using general purpose logic programming systems – A case study. *ACM SIGPLAN Notices*, 31(5):117–126, May 1996.

[7] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In Batory et al. [1], pages 173–188.

[8] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In R.Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, Lancaster University, July 2006.

[9] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *2nd European Interactive Workshop on Aspects in Software (EIWAS 2005)*, Brussels, Belgium, Sept. 2005.

[10] W. Havinga, I. Nagy, L. Bergmans, and M. Aksit. Detecting and resolving ambiguities caused by inter-dependent introductions. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 214–225, Bonn, Germany, Mar. 2006. ACM Press.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In K. Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 26–35, Lancaster, UK, Mar. 2004. ACM Press.

[12] JLog. http://jlogic.sourceforge.net/.

[13] B. Kessler and É. Tanter. Analyzing interactions of structural aspects. In *Proceedings of ECOOP Workshop on Aspects, Dependencies and Interactions*, Nantes, France, July 2006.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[15] LogicAJ. http://roots.iai.uni-bonn.de/research/logicaj/.

[16] PrologDT. http://roots.iai.uni-bonn.de/research/pdt/.

[17] T. W. (Rho). Logicaj - eine erweiterung von aspectj um logische metaprogrammierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, Aug 2003.

[18] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 248–274, Darmstadt, Germany, July 2003. Springer-Verlag.

[19] D. Suvee, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In M. Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 21–29, Boston, MA, USA, Mar. 2003. ACM Press.

[20] É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, Lecture Notes in Computer Science, Vienna, Austria, Mar. 2006. Springer-Verlag. To appear.

[21] É. Tanter and J. Noyé. Motivation and requirements for a versatile AOP kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, Sept. 2004.

[22] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia, Sept./Oct. 2005. Springer-Verlag.

[23] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In R. Crocker and G. L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, Oct. 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[24] W. Vanderperren, D. Suvée, D. D. Fraine, and V. Jonckers. Aspect-oriented programming using JAsCo, 2005.